

成都理工大学软件工程专业卓越工程师培养系列教材

计算机图形学实验指导书

成都理工大学软件工程系

2014 年 12 月

目 录

计算机图形学实验介绍.....	1
本课程实验的作用与任务.....	1
本课程实验的基础知识.....	1
实验教学项目及教学要求.....	1
实验 1 GLUT 的 Hello World 程序.....	3
1.1 实验目的.....	3
1.2 实验原理.....	3
1.3 主要实验步骤.....	4
实验 2 交互直线绘制.....	13
2.1 实验目的.....	13
2.2 实验原理.....	13
2.3 主要实验步骤.....	14
实验 3 多边形填充.....	20
3.1 实验目的.....	20
3.2 实验原理.....	20
3.3 主要实验步骤.....	23
实验 4 直线和多边形裁剪.....	31
4.1 实验目的.....	31
4.2 实验原理.....	31
4.2.1 直线编码裁剪原理.....	31
4.2.2 多边形逐边裁剪原理.....	33
4.3 主要实验步骤.....	35
4.3.1 直线编码裁剪主要步骤.....	35
4.3.2 多边形逐边裁剪主要步骤.....	37
实验 5 OpenGL 三维图形程序设计.....	41
5.1 实验目的.....	41
5.2 实验原理.....	41
5.3 实验主要实现步骤.....	44
实验 6 鼠标追踪球旋转.....	52

6.1 实验目的.....	52
6.2 实验原理.....	52
6.3 实验主要实现步骤.....	54
实验 7 光照与纹理	59
7.1 实验目的.....	59
7.2 实验原理.....	59
7.3 实验主要实现步骤.....	62
实验 8 三维 B 样条曲面绘制	70
8.1 实验目的.....	70
8.2 实验原理.....	70
8.3 主要实验步骤.....	71

计算机图形学实验介绍

本课程实验的作用与任务

计算机图形学是研究利用计算机来处理图形的原理、方法和技术的学科。图形的处理包括了图形生成、图形描述、图形存储、图形变换、图形绘制、图形输出等等。计算机图形学与计算机图形处理技术是许多重要应用领域的基础，图形技术已经交叉渗透到各个应用学科中。

图形学是一门含有较多算法和原理的课程，通过编程实践，可以让学生进一步理解和掌握计算机图形技术的知识，熟悉常用计算机图形应用软件开发接口，培养计算机图形处理的综合能力。本实验指导书依据《计算机图形学》课程教学大纲编写，课程的任务是让学生学会如何把书本上学到的知识用于解决实际问题，培养软件工作所需要的动手能力；另一方面，能使书本上的知识变“活”，起到深化理解和灵活掌握教学内容的目的。

本课程实验的基础知识

本课程的主要讲授计算机图形学的基本概念、研究内容和应用领域；图形信息的计算机处理；图形系统的组成和图形设备；基本图元的生成技术；图形的填充与剪裁；二/三维图形变换技术；消隐及真实感图形生成技术，计算机图形标准、图形数据结构和 OpenGL 图形程序设计等。

实验教学项目及教学要求

序号	实验项目	学时	教学目标及要求
1	GLUT 的 Hello World 程序	2	通过一个简单的二维 GLUT 图形程序掌握 VC 平台下 GLUT 开发环境的搭建；基本的 GLUT 程序结构；OpenGL 基本的视窗变换和二维交互
2	交互直线绘制	2	实现一个简单的通过鼠标交互绘制直线的程序，理解 Bresenham 直线绘制算法；掌握基本的二维

			交互程序设计方法
3	多边形填充	2	设计并实现一个多边形交互编辑和填充程序，理解多边形扫描线和改进的扫描线填充算法；实现算法程序
4	直线和多边形裁剪	2	完成一个直线裁剪和一个多边形裁剪程序，理解直线编码裁剪和多边形逐边裁剪算法；实现算法程序
5	OpenGL 三维图形程序设计	2	实现一个简单的三维 OpenGL 程序并测试几种投影矩阵，理解三维投影变换算法；掌握三维 OpenGL 程序的基本结构；OpenGL 三维投影变换、视窗变换
6	鼠标追踪球旋转	2	为基本的三维图形程序添加鼠标追踪球旋转功能，理解鼠标追踪球算法原理；掌握 OpenGL 模型变换方法
7	光照与纹理	2	在鼠标追踪球程序的基础上实现一个带光照和纹理的球体绘制程序，理解真实感图形生成的关键技术；掌握 OpenGL 消隐、光照、材质和纹理编程技术
8	三维 B 样条曲面绘制	2	在鼠标追踪球程序的基础上实现一个带光照的三次 B 样条曲面绘制程序，理解 B 样条曲面算法和公式；加深 OpenGL 光照、材质编程方法

实验1 GLUT 的 Hello World 程序

1.1 实验目的

- (1) 熟练使用实验主要开发平台 VC6 或 VS2010;
- (2) 掌握 GLUT 开发库在 VC 平台下的配置和 GLUT 工程的建立方法;
- (3) 掌握 GLUT 二维图形程序开发的基本方法和基本的 GLUT 回调。

1.2 实验原理

GLUT 是一个跨平台的 OpenGL 应用工具包, 英文全称为 OpenGL Utility Toolkit, 它与具体的操作系统无关, 作为 OpenGL 的 AUX 库的替代品, 它可以隐藏不同窗口系统 API 的复杂性。使用 GLUT 可以让开发人员不必去了解具体的视窗操作系统, 及该系统对 OpenGL 提供的专有函数, 使得 OpenGL 的开发得到极大的简化。

GLUT 提供了一组 C 风格的 API 函数, 事件采用回调函数的方式进行处理。一个最基本的 GLUT 程序, 只需要包含一个主程序和一个 display 回调函数, 其中主程序负责初始化 GLUT 库、创建窗口、设置 display 回调函数并进入 GLUT 事件循环; display 回调函数则主要通过调用 OpenGL 函数来实现图形绘制。

一个普通的 GLUT 程序除了 display 回调函数以外, 通常还需要设置 reshape 回调, 用于处理 OpenGL 视窗变换和投影变换。通过设置 mouse 回调来处理鼠标事件, 通过设置 key 回调来处理键盘事件。GLUT 常用的窗口回调函数有:

- 显示回调:

```
void glutDisplayFunc(void (GLUTCALLBACK *func)(void));
```

- 窗口调整大小回调:

```
void glutReshapeFunc(void (GLUTCALLBACK *func)(int w, int h));
```

w: 窗口宽度

h: 窗口高度

- 键盘回调:

```
glutKeyboardFunc(void (GLUTCALLBACK *func)(unsigned char key, int x, int y));
```

key: 按键 ASCII 码值

x, y: 键盘事件发生时的鼠标窗口坐标

- 鼠标回调:

`glutMouseFunc(void (GLUTCALLBACK *func)(int button, int state, int x, int y));`

button: 鼠标按键, 可能取值: GLUT_LEFT、GLUT_MIDDLE、GLUT_RIGHT

state: 按键状态, 可能取值: GLUT_DOWN、GLUT_UP

x, y: 鼠标事件发生时的鼠标窗口坐标

- 鼠标拖拽回调 (鼠标拖拽事件在按下并移动鼠标时发生):

`glutMotionFunc(void (GLUTCALLBACK *func)(int x, int y));`

x, y: 鼠标拖拽时的鼠标窗口坐标

- 鼠标移动回调

`glutPassiveMotionFunc(void (GLUTCALLBACK *func)(int x, int y));`

x, y: 鼠标移动时的鼠标窗口坐标

1.3 主要实验步骤

(1) 搭建开发环境

检查实验电脑上是否安装了 VC6 或者 VS2008/2010/2012 开发工具, 如果没有则先下载安装, 对于本课程实验安装 VC6 即可满足实验要求。

下载 GLUT 3.7.6 或更高版本的 Windows 开发库, 解压缩下载的压缩包, 找出文件: `glut.h`、`glut32.lib`、`glut32.dll` (这些文件可以在配套的“实验 1”目录中找到)。为了方便之后的实验课程, 将它们拷贝到 VC 的开发目录中。对于 VC6 按如下方式配置:

- 将 `glut.h` 拷贝到 %VC6%\VC98\Include\GL 目录中
- 将 `glut32.lib` 拷贝到 %VC6%\VC98\Lib 目录中
- 将 `glut32.dll` 拷贝到 %Windows%\System32 目录中

上面 %VC6 表示 VC6 的安装目录, %Windows 表示 Windows 系统的安装目录。

对于 VS2008/VS2010/VS2012 等开发工具配置方式如下：

- 将 glut.h 拷贝到 %Program Files\Microsoft SDKs\Windows\v7.0A\Include\GL 目录中
- 将 glut32.lib 拷贝到 %Program Files\Microsoft SDKs\Windows\v7.0A\Lib 目录中
- 将 glut32.dll 拷贝到 %Windows\System32 目录中

上面 %Program Files 表示系统的 Program Files 目录，通常在 C 盘中。中间目录 v7.0A 对于不同的 VS 环境可能不同，VS2008 为 v6.0A，VS2010 为 v7.0A，VS2012 为 v8.0A，请根据所使用的开发工具正确选择。

(2) 新建工程

使用开发工具新建一个控制台工程，若使用 VC6 步骤如下：

- 选择菜单：【File->New】，打开新建对话框，选择 Projects 选项页，在列表选中 Win32 Console Application 选项，选择一个合适的工程目录，并输入工程名 HelloWorld，并按下【Ok】按钮，如图 1-1 所示。

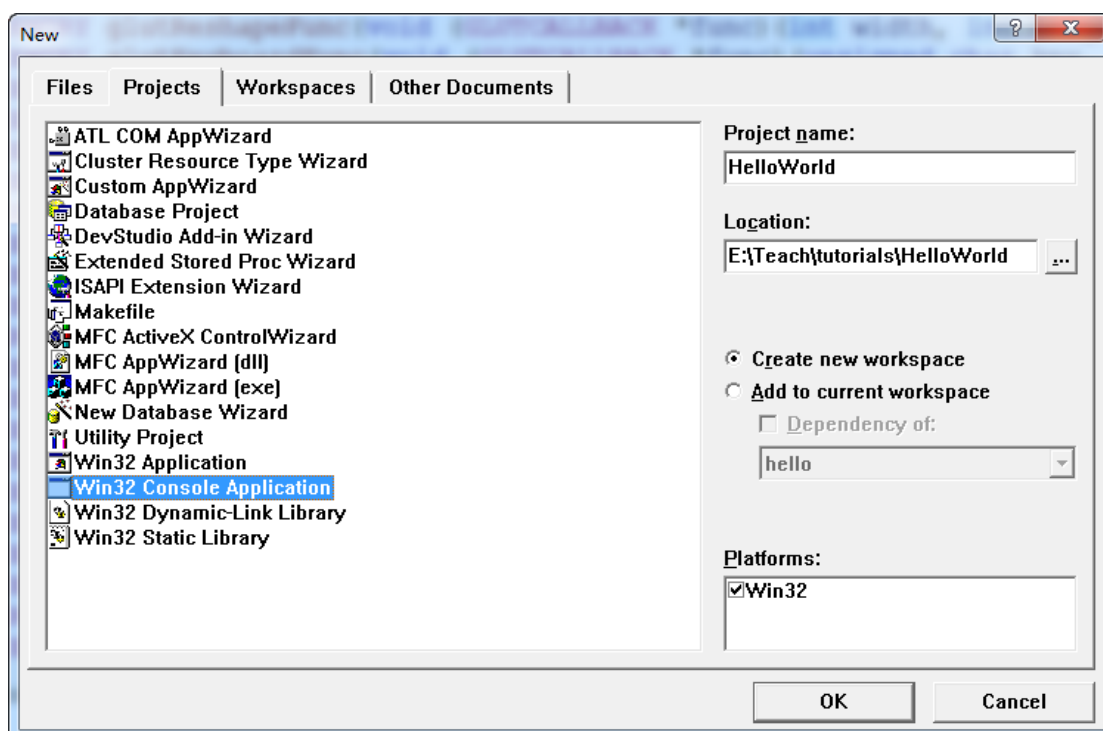


图 1-1 VC6 新建控制台工程 HelloWorld

- 在接下来的向导对话框中选择 An empty project，如图 1-2 所示。

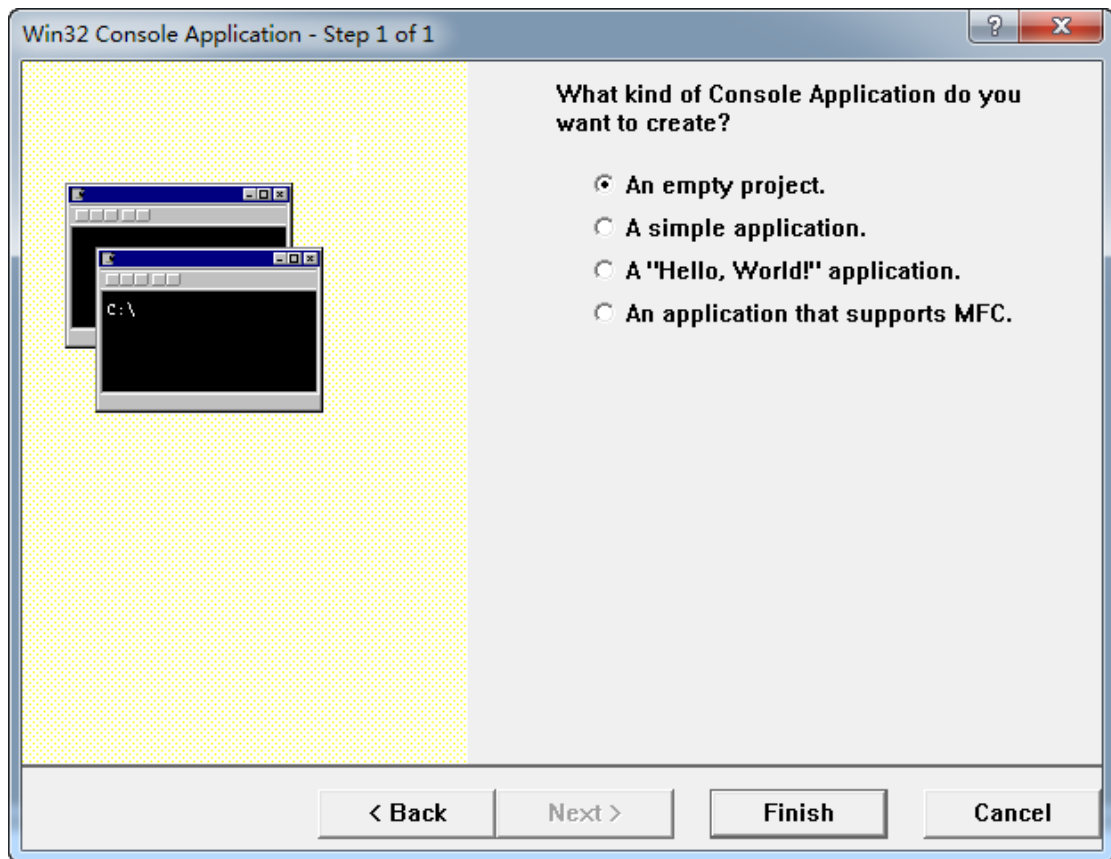


图 1-2 VC6 新建空工程

若使用 VS2010 步骤如下：

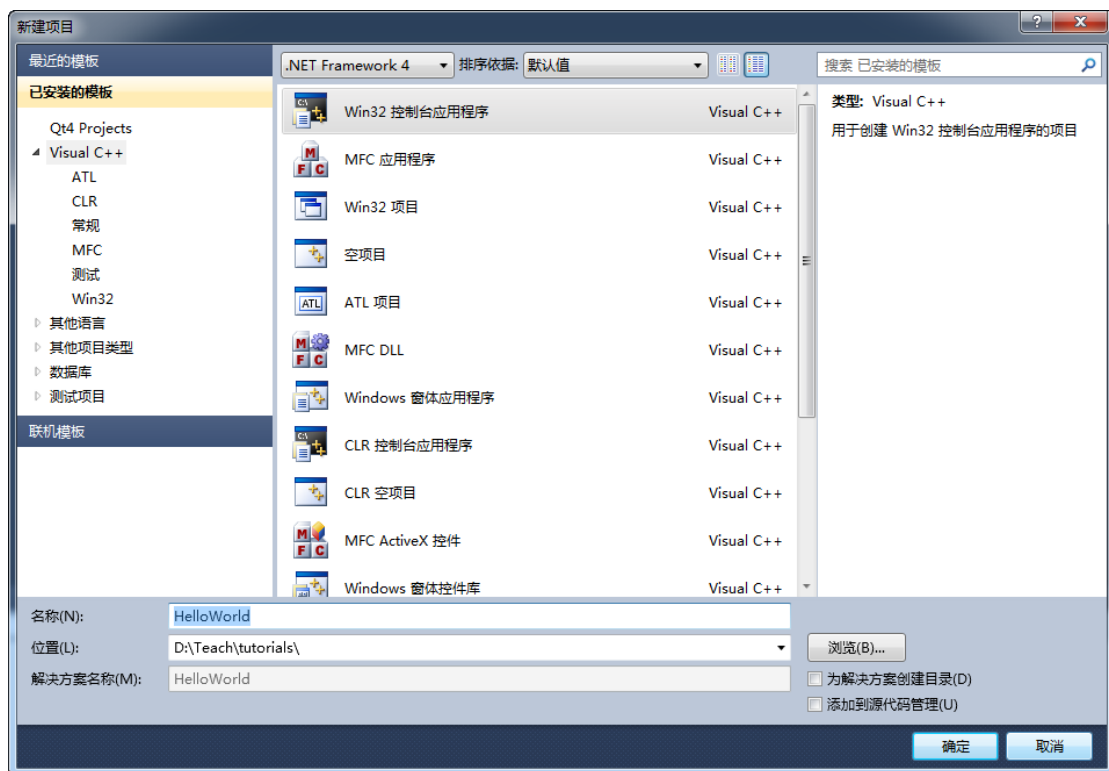


图 1-3 VS2010 新建控制台工程 HelloWorld

- 选择菜单:【文件->新建项目】, 打开新建项目对话框, 选择项目模板为 Visual C++->Win32 控制台应用选项, 选择一个合适的工程目录, 并输入工程名 HelloWorld, 并按下【确定】按钮, 如图 1-3 所示。
- 在接下来的应用程序向导对话框的第一页中直接选择【下一步】按钮, 进入第二页。在第二页中应用程序类型保持为: 控制台应用程序, 附加选项请勾选上: 空项目复选框, 请参考图 1-4。最后按下【完成】按钮, 结束新建工程。



图 1-4 VS2010 新建空工程

(3) 新建 main.cpp 源程序文件

新建工程完毕后, 再新建一个源程序文件 main.cpp, 并将其加入到 HelloWorld 工程中。如使用 VC6, 选择菜单:【File->New】打开新建对话框, 在对话框中输入文件名为: main.cpp, 文件位置保持在 HelloWorld 工程目录中, 并确保对话框右上角处的 Add to project 复选框被选中。**注意:** 若未勾选 Add to project, 新建的文件不会加入到工程中, 工程构建时将不会使用该文件。如果在新建文件时忘记了勾选 Add to project 也可以在以后通过菜单:【Project->Add to

project->Files】将指定文件加入到工程中。新建文件如图 1-5 所示。

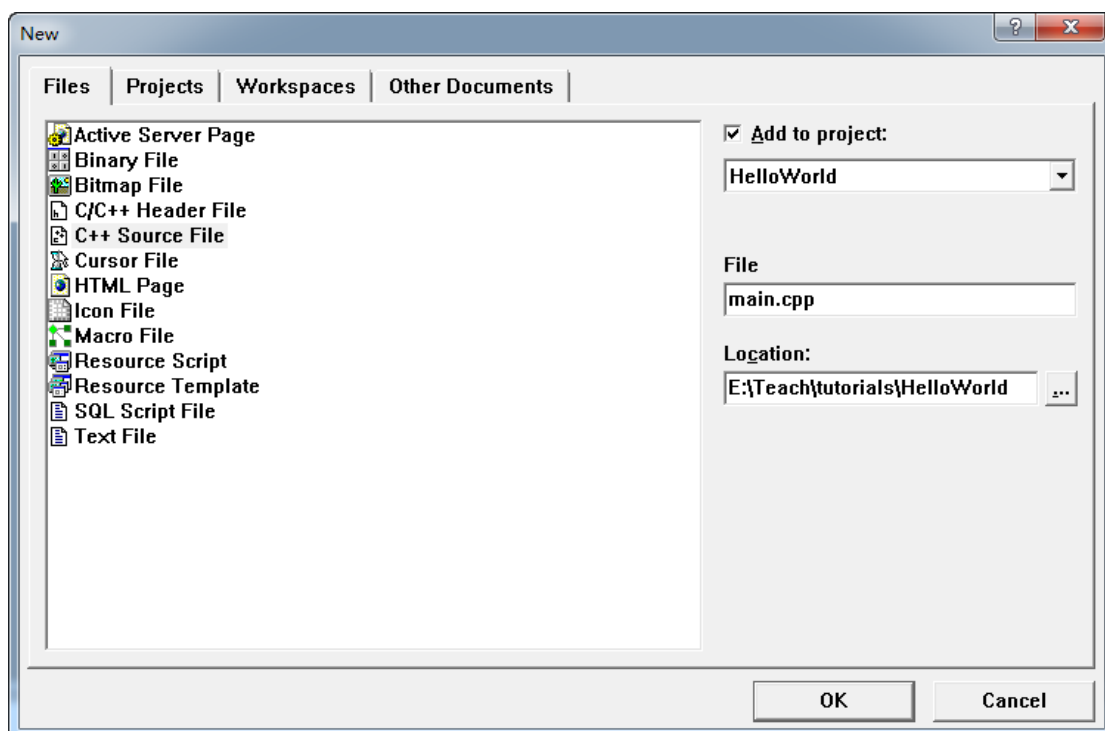


图 1-5 VC6 中新建 main.cpp 文件

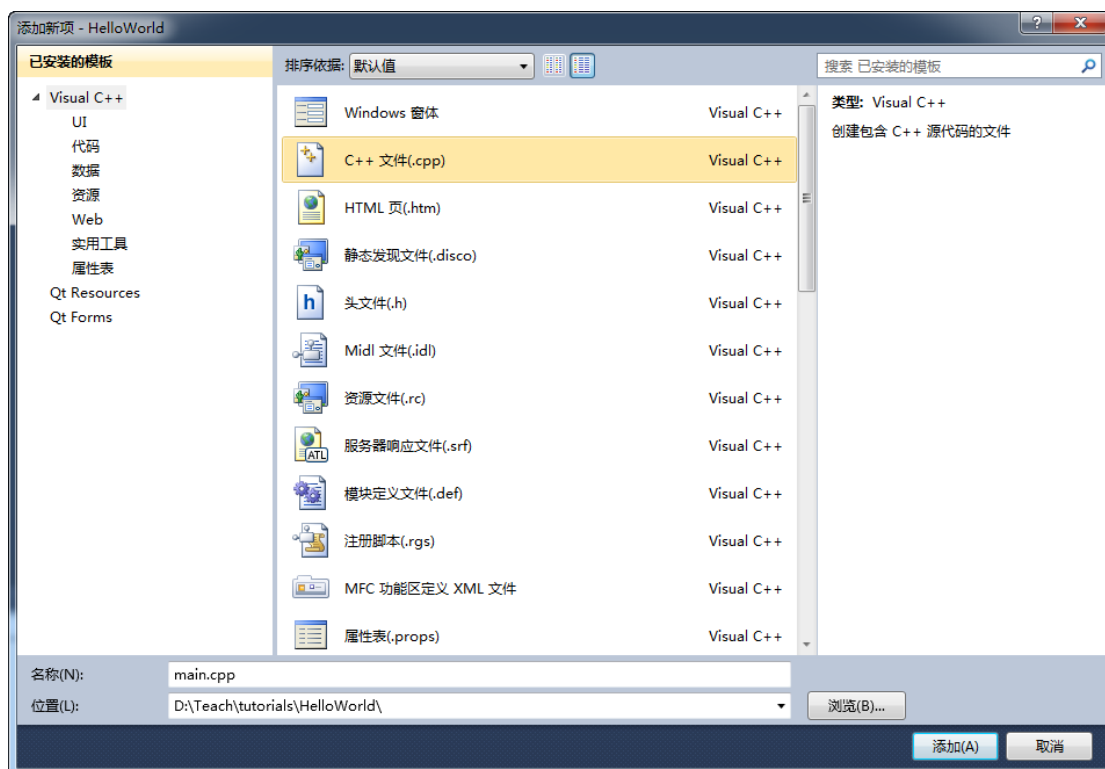


图 1-6 VS2010 中新建 main.cpp 文件

如果使用 VS2010，在解决方案资源管理器中先选中【HelloWorld】项目，然后点击右键，在弹出的快捷菜单中选择【添加->新建项】，弹出【添加新项】

对话框，在对话框中选择模板为：**【Visual C++->C++文件】**，输入文件名为：**main.cpp**，文件位置保持在 **HelloWorld** 工程目录中，然后按下确定按钮。如图 1-6 所示。

(4) 实现一个最基本的 GLUT 程序

选择并打开 **main.cpp** 文件，在文件中编写一个 **onDisplay** 和 **main** 函数，实现一个最基本的 GLUT 程序，显示一个空白窗口。为此在 **main.cpp** 文件中输入以下代码：

```
144 #include <gl/glut.h>
145
146 void onDisplay()
147 {
148     // 设置清屏颜色
149     glClearColor(1, 1, 1, 0);
150     // 用指定颜色清除帧缓存
151     glClear(GL_COLOR_BUFFER_BIT);
152     // 交换双缓存
153     glutSwapBuffers();
154 }
155
156 int main(int argc, char *argv[])
157 {
158     // 初始化 glut
159     glutInit(&argc, argv);
160     // 设置 OpenGL 显示模式(双缓存, RGB 颜色模式)
161     glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
162     // 设置窗口初始尺寸
163     glutInitWindowSize(400, 300);
164     // 设置窗口初始位置
165     glutInitWindowPosition(100, 100);
166     // 设置窗口标题
167     glutCreateWindow("Hello");
168     // 设置显示回调函数
169     glutDisplayFunc(onDisplay);
170     // 进入 glut 事件循环
171     glutMainLoop();
172
173     return 0;
174 }
```

编写完毕后对于 VC6 选择菜单：**【Build->Build HelloWorld.exe (F7)】**构建程序，如果没有错误选择菜单：**【Build->Execute HelloWorld.exe (Ctrl+F5)】**运行程序。对于 VS2010 则选择菜单：**【生成->生成解决方案 (F7)】**构建程序，如果没有错误选择菜单：**【调试->开始执行不调试 (Ctrl+F5)】**运行程序。程序正确运行结果如图 1-7 所示。

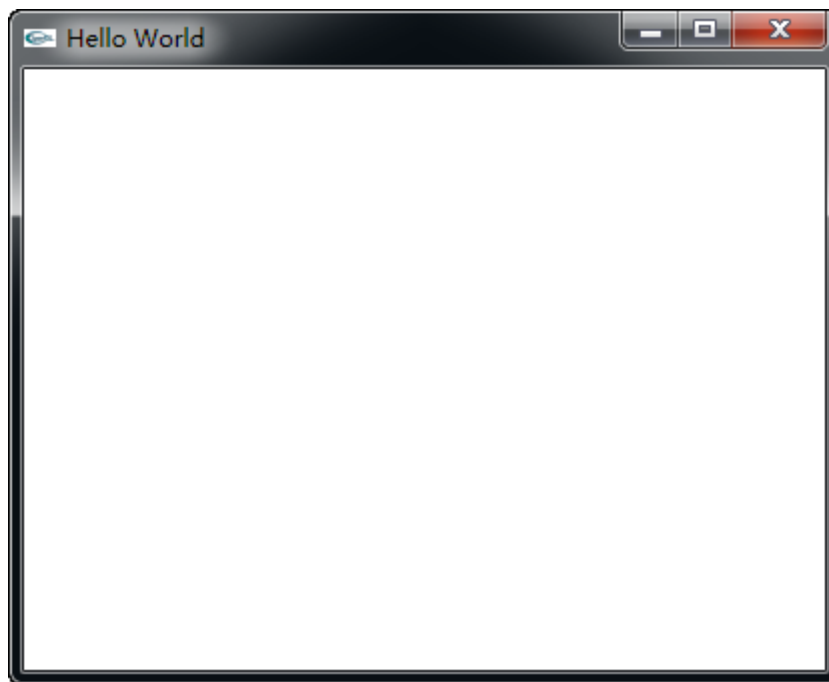


图 1-7 一个最基本的 GLUT 程序运行效果

(5) 增加 onReshape 函数，增加投影功能

在 main 函数之前插入 onReshape 函数，实现二维投影变换功能，通过投影变换使得 OpenGL 坐标系与 Windows 坐标系一致，具体代码如下：

```
13 void onReshape(int w, int h)
14 {
15     // 设置视口大小
16     glViewport(0, 0, w, h);
17     // 切换矩阵模式为投影矩阵
18     glMatrixMode(GL_PROJECTION);
19     // 载入单位矩阵
20     glLoadIdentity();
21     // 进行二维平行投影
22     gluOrtho2D(0, w, h, 0);
23     // 切换矩阵模式为模型矩阵
24     glMatrixMode(GL_MODELVIEW);
25     // 发送重绘
26     glutPostRedisplay();
27 }
```

实现 onReshape 函数后，在主程序 glutMainLoop 调用前设置该回调函数

```
42 // 设置显示回调函数
43 glutDisplayFunc(onDisplay);
44 // 设置窗口尺寸变化回调函数
45 glutReshapeFunc(onReshape);
46 // 进入 glut 事件循环
47 glutMainLoop();
```

(6) 修改 onDisplay 函数，增加文字显示功能

OpenGL 中显示文字比较复杂，文字显示通常有光栅和矢量两种，这里通过

GLUT 提供的 `glutBitmapCharacter` 函数来显示字符。在原有 `onDisplay` 函数 `glClear` 函数调用之后加入如下代码：

```
8  glClear(GL_COLOR_BUFFER_BIT);
9  static char text[] = "Hello World!";
10 // 定位输出位置
11 glRasterPos2d(50, 100);
12 // 设置字符串颜色
13 glColor3f(1, 0, 0);
14 for(int i=0; text[i] != '\0'; i++)
15 {
16     // 输出字符
17     glutBitmapCharacter(GLUT_BITMAP_8_BY_13, text[i]);
18 }
19 // 交换双缓存
20 glutSwapBuffers();
```

完成上述代码后构建并运行程序得到一个白色窗口，窗口左上角（50，100）位置处显示出“Hello World”字符串，如图 1-8 所示。

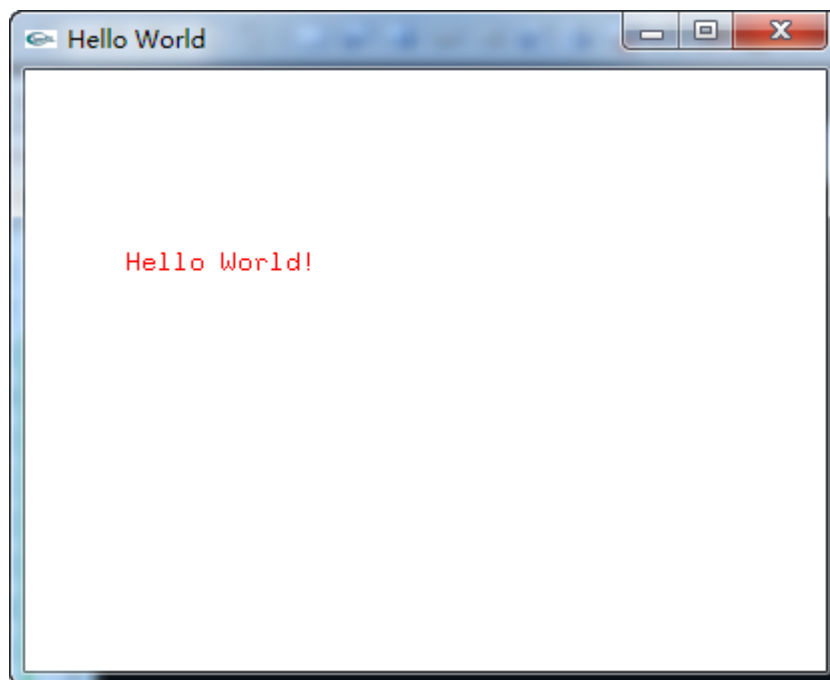


图 1-8 输出“Hello World”效果

（7） 增加鼠标交互功能

GLUT 的鼠标回调函数允许程序在鼠标按键按下或释放时执行指定的事件函数，利用鼠标事件可以使程序响应用户输入，修改“Hello World”的显示位置。为此首先在程序首部增加字符输出位置变量 `gx`, `gy`：

```
3  #include <gl/glut.h>
4
5  int gx = 50;
6  int gy = 100;
```

接下来在 `onDisplay` 函数中使用该坐标显示字符串，修改原有程序为：

```
6  void onDisplay()
7  {
8      ...
9      char text[] = "Hello World!";
10     // 定位输出位置
11     glRasterPos2d(x, y);
12     // 设置字符串颜色
13     glColor3f(1, 0, 0);
14     ...
15 }
```

然后在 `main` 函数前增加鼠标回调函数，当鼠标左键按下时修改 `gx`, `gy` 变量，然后发送重绘消息，调用 `onDisplay` 刷新窗口。

```
44 // 鼠标事件函数
45 void onMouse(int button, int state, int x, int y)
46 {
47     if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
48     {
49         gx = x;
50         gy = y;
51         glutPostRedisplay();
52     }
53 }
```

最后在主程序 `glutMainLoop` 调用前设置鼠标回调函数：

```
54 void main(int argc, char *argv[])
55 {
56     ...
57     // 设置窗口尺寸变化回调函数
58     glutReshapeFunc(onReshape);
59     // 设置鼠标事件回调函数
60     glutMouseFunc(onMouse);
61     // 进入 glut 事件循环
62     glutMainLoop();
63     ...
64 }
```

完成上述修改后构建并运行程序，在窗口中按下鼠标左键“Hello World”字符串将被移动到鼠标所在位置。

实验2 交互直线绘制

2.1 实验目的

- (1) 理解 Bresenham 直线生成算法，实现直线生成程序；
- (2) 掌握基本的 GLUT 交互式程序设计方法；
- (3) 实现完整的交互直线绘制程序。

2.2 实验原理

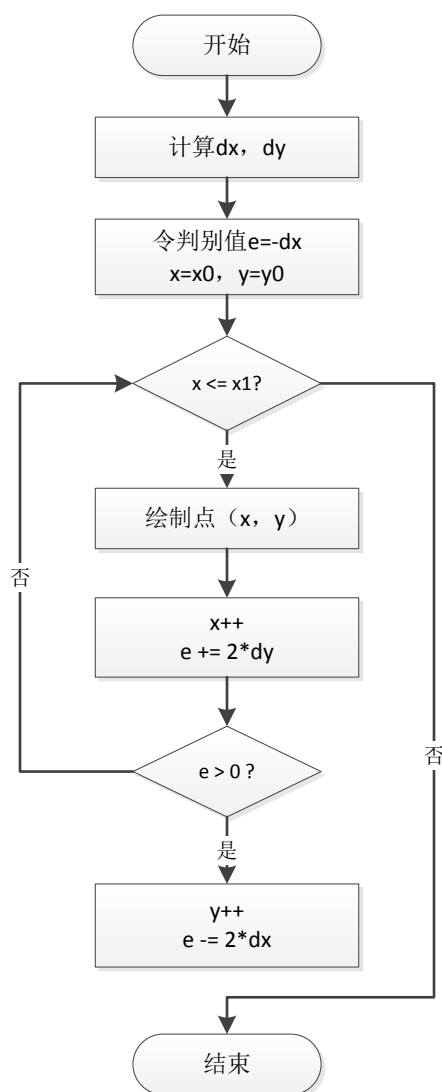


图 2-1 Bresenham 算法流程图

光栅直线的绘制就是在有限个像素组成的矩阵中，确定一组最佳逼近于该直

线的像素组。Bresenham 是一种高效的光栅直线绘制算法,假设直线斜率 $k \in [0,1)$,该方法在 X 方向上每次前进一个像素,同时根据判别值 e 是否大于 0,决定 Y 的走向,并按照递推公式更新判别值 e。Bresenham 算法无需浮点数和四舍五入运算,仅需整数加减运算,因而效率很高,算法流程图如图 2-1 所示。

上述基本的 Bresenham 算法仅能绘制 0—45 度之间的直线段,其它角度的直线在绘制时要进行一定的处理,对于 $|dx| > |dy|$ 的情况要交换 X、Y 方向,对于 $x_0 > x_1$ 的情况则需要交换起始点和终止点的位置。

在直线绘制程序的基础上,本次实验还要结合鼠标事件,实现一个简单交互直线绘制程序。为了简化实验,该程序只考虑操作并绘制一条直线,通过按下鼠标左键确定直线起点,通过拖拽鼠标移动直线的终点,释放鼠标左键完整直线绘制。交互程序通常由模型 M、视图 V、控制器 C 三部分构成,模型是图形的内部数据结构,视图是显示图形的窗口对象,控制器接受用户输入并调度模型和视图完成用户交互功能。本次试验完成的简单程序对 MVC 模式进行了简化,用 GLine (直线类) 充当模型,在 onDisplay 回调函数中实现视图绘制功能,在 onMouse 和 onMotion 鼠标及拖拽回调函数中实现控制功能。



2.3 主要实验步骤

(1) 新建工程

在 VC6 或 VS2010 中新建一个工程名为“Line”的空控制台工程,具体操作参考按照 1.3 节步骤 (2)。

(2) 添加文件

为了简化实验,将“实验 2”目录中的准备好的源代码文件: main.cpp、point2.h、line2.h、tools.h/cpp 拷贝到“Line”工程目录中。对于 VC6 选择菜单:【Project->Add to project->Files】,在弹出的文件选择对话框中选中上述文件并按下【确定】按钮。注意:由于 VC6 和一些较新版本的操作系统不兼容,文件添加功能可能会造成 VC6 系统崩溃退出。如遇到这种情况,请将“实验 2”目录中的 FileTool.dll 作为插件添加到 VC6 中,选择菜单【Tools->Customize】,在弹出的 Customize 对话框中选择 Add-ins and Macro Files 选项页,如图 2-2 所示。点击【Browse...】

按钮，在弹出的对话框中选中 FileTool.dll 文件，然后按下【打开】按钮。关闭 Customize 对话框退回 VC6 主界面，会发现多了一个工具条：。通过工具条上的  按钮，即可向工程中添加已有文件。

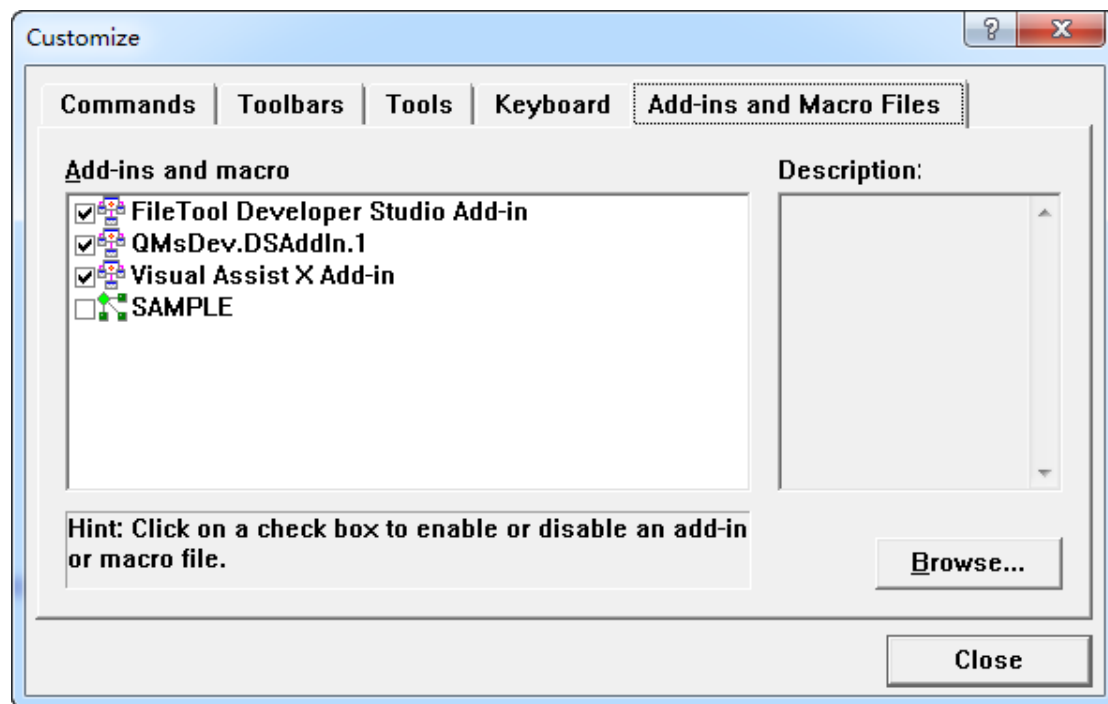


图 2-2 VC6 Customize 对话框

对于 VS2010 选择菜单：【项目->添加现有项】，在弹出的文件选择对话框中选择待添加的：main.cpp、point2.h、line2.h、tools.h/cpp，然后点击【添加】按钮。除了通过菜单添加现有文件，也可先在文件浏览器中选中待添加的文件，然后将它们拖拽到 VS2010 的解决方案资源管理器对应的项目中，完成添加。

添加的 main.cpp 文件是“实验 1”已完成的程序，point2.h 定义并实现了 GPoint2 二维顶点模板类，该类用于存储顶点的 X、Y 坐标。line2.h 定义并实现了 GLine2 二维直线模板类，该类用于存储直线的两个端点。GPoint2 和 GLine2 定义并实现的功能如下表所示。

表 2-1 GPoint2 类函数列表

函数名	函数功能
GPoint2()	默认构造函数，初始化 x、y 坐标均为 0
GPoint2(T x, T y)	通过指定 x, y 坐标初始化 GPoint2 对象
T x()	返回 x 坐标
T y()	返回 y 坐标

void setX(T x)	设置 x 坐标
void setY(T y)	设置 y 坐标
void set(T x, T y)	设置 x, y 坐标
GPoint2 & operator = (const GPoint2 &other)	赋值运算符, 将 other 赋值给当前 GPoint2 对象

表 2-2 GLine2 类函数列表

函数名	函数功能
GLine2()	默认构造函数, 初始化起始和终止顶点坐标均为 (0, 0)
GLine2(const GPoint2<T> &begin, const GPoint2<T> &end)	通过指定 begin 和 end 顶点初始化 GLine2 对象
GLine2(T x0, T y0, T x1, T y1)	通过指定顶点坐标初始化 GLine2 对象
GLine2(const GLine2 &other)	拷贝构造函数
const GPoint2<T> & getBeginPt()	返回起始顶点坐标
const GPoint2<T> & geEndPt()	返回终止顶点坐标
void setBeginPt(const GPoint2<T> &begin)	设置起始顶点对象
void setEndPt(const GPoint2<T> &begin)	设置终止顶点对象
void set((const GPoint2<T> &begin, const GPoint2<T> &end)	设置直线的起始和终止顶点对象
void set(T x0, T y0, T x1, T y1)	设置直线顶点的坐标
GLine2 & operator = (const GLine2 &other)	赋值运算符, 将 other 赋值给当前 GLine2 对象

GPoint2 和 GLine2 是模板类, 为了方便使用程序中还使用下面语句定义了常用的浮点和整形顶点与直线类。

```

65 typedef GPoint2<float> GPoint2f;      // 单精度顶点
66 typedef GPoint2<double> GPoint2d;    // 双精度顶点
67 typedef GPoint2<int> GPoint2i;       // 整形顶点
68 typedef GLine2<float> GLine2f;       // 单精度直线
69 typedef GLine2<double> GLine2d;      // 双精度直线
70 typedef GLine2<int> GLine2i;         // 整形直线

```

(3) 实现直线绘制函数

tools.h/cpp 文件给出了直线绘制函数的定义, 但并未实现。为此先打开 tools.cpp 文件并定位到 gltLine2i 函数, 然后输入下面的代码。

```

71 void gltLine2i(int x0, int y0, int x1, int y1)
72 {
73     bool isSwapXY = false;
74     int x, y, dx, dy, tx, ty, e, dx2, dy2;
75
76     dx = abs(x1 - x0);
77     dy = abs(y1 - y0);
78     if(dx < dy)
79     {
80         swap(x0, y0);

```

```

81     swap(x1, y1);
82     swap(dx, dy);
83     isSwapXY = true;
84 }
85
86 if(x0 <= x1) tx = 1;
87 else tx = -1;
88 if(y0 <= y1) ty = 1;
89 else ty = -1;
90
91 e = -dx;
92 x = x0;      y = y0;
93 dx2 = dx << 1;
94 dy2 = dy << 1;
95
96 glBegin(GL_POINTS);
97 while(x != x1)
98 {
99     if(isSwapXY) glVertex2i(y, x);
100    else glVertex2i(x, y);
101
102    x += tx;
103    e += dy2;
104    if(e > 0)
105    {
106        y += ty;
107        e -= dx2;
108    }
109 }
110
111 if(isSwapXY) glVertex2i(y, x);
112 else glVertex2i(x, y);
113 glEnd();
114 }

```

除了 `gltLine2i` 函数外 `tools.h/cpp` 文件中还定义并实现了如下工具函数：

表 2-3 tools 函数列表

函数名	函数功能
<code>gltRect2i(int x0, int y0, int x1, int y1)</code>	绘制整数坐标的矩形
<code>gltLine2d(int x0, int y0, int x1, int y1)</code>	绘制双精度坐标的直线
<code>gltLine2d(int x0, int y0, int x1, int y1)</code>	绘制双精度坐标的矩形
<code>void gltRasterText(double x, double y, const char *text, void *font)</code>	在指定光栅位置 x, y 处绘制光栅字符串 text

(4) 增加直线对象，实现 `onDisplay` 函数

添加到程序中 `main.cpp` 文件，是一个基本的 GLUT 程序，仅仅实现了清屏显示功能，并加入了空的鼠标及拖拽事件函数。为了实现交互直线绘制，首先在程序首部引用“`gpoint2.h`”、“`gline2.h`”和“`tools.h`”文件，并定义一些全局对象和变量。

```

1  #include "tools.h"
2  #include "gpoint2.h"
3  #include "gline2.h"
4
5  // 直线对象
6  GLine2i gLine;
7  // 是否绘制视图
8  bool gIsPaint = false;
9  // 鼠标左键是否按下
10 bool gIsLBtnDown = false;

```

在原有 onDisplay 函数 glClear 调用之后加入如下代码：

```

11 void onDisplay()
12 {
13     glClearColor(1, 1, 1, 0);
14     glClear(GL_COLOR_BUFFER_BIT);
15     if(gIsPaint)
16     {
17         GPoint2i pt0, pt1;
18         pt0 = gLine.getBeginPt();
19         pt1 = gLine.getEndPt();
20
21         // 绘制直线
22         glColor3f(0, 0, 0);
23         gltLine2i(pt0.x(), pt0.y(), pt1.x(), pt1.y());
24         // 绘制起点红色十字
25         glColor3f(1, 0, 0);
26         gltLine2i(pt0.x()-8, pt0.y(), pt0.x()+8, pt0.y());
27         gltLine2i(pt0.x(), pt0.y()-8, pt0.x(), pt0.y()+8);
28         // 绘制尾点方框
29         glColor3f(0, 0, 1);
30         gltRect2i(pt1.x()-5, pt1.y()-5, pt1.x()+5, pt1.y()+5);
31         // 标注坐标
32         char text[32];
33         sprintf(text, "(%d, %d)", pt0.x(), pt0.y());
34         glColor3f(1, 0, 0);
35         gltRasterText(pt0.x()+10, pt0.y()-5, text);
36         sprintf(text, "(%d, %d)", pt1.x(), pt1.y());
37         glColor3f(0, 0, 1);
38         gltRasterText(pt1.x()+10, pt1.y()-5, text);
39     }
40     glutSwapBuffers(); // 交换缓存
41 }

```

(5) 实现鼠标交互功能

在鼠标事件 onMouse 回调函数中输入下面代码：

```

42 void onMouse(int button, int state, int x, int y)
43 {
44     if(button == GLUT_LEFT_BUTTON)
45     {
46         if(state == GLUT_DOWN)
47         {
48             gLine.set(x, y, x, y);
49             gIsLBtnDown = true;
50             gIsPaint = true;
51         }
52         else if(state == GLUT_UP)

```

```
53     {  
54         gLine.setEndPt(GPoint2i(x, y));  
55         gIsLBtnDown = false;  
56     }  
57     glutPostRedisplay();  
58 }  
59 }
```

在鼠标拖拽事件 `onMotion` 回调函数中输入下面代码：

```
60 void onMouseMotion(int x, int y)  
61 {  
62     if(gIsLBtnDown)  
63     {  
64         gLine.setEndPt(GPoint2i(x, y));  
65         glutPostRedisplay();  
66     }  
67 }
```

完成上述修改后构建并运行程序，在窗口中按下鼠标左键并拖拽即可绘制出一条如图 2-3 所示的直线段。

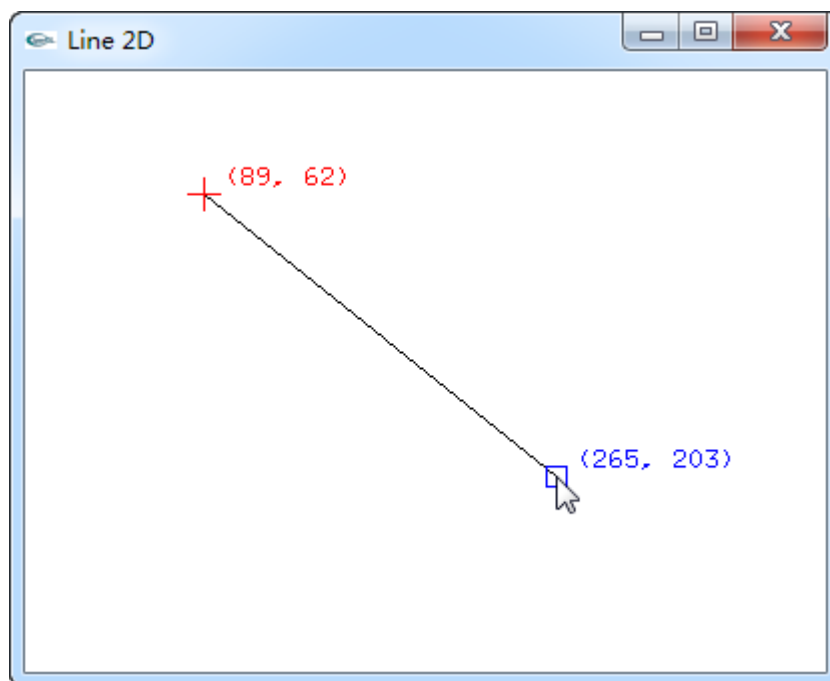


图 2-3 交互绘制直线程序运行效果

实验3 多边形填充

3.1 实验目的

- (1) 实现多边形交互编辑功能；
- (2) 理解扫描线多边形填充算法原理，实现算法程序；
- (3) 理解改进的扫描线多边形填充算法原理，实现算法程序。

3.2 实验原理

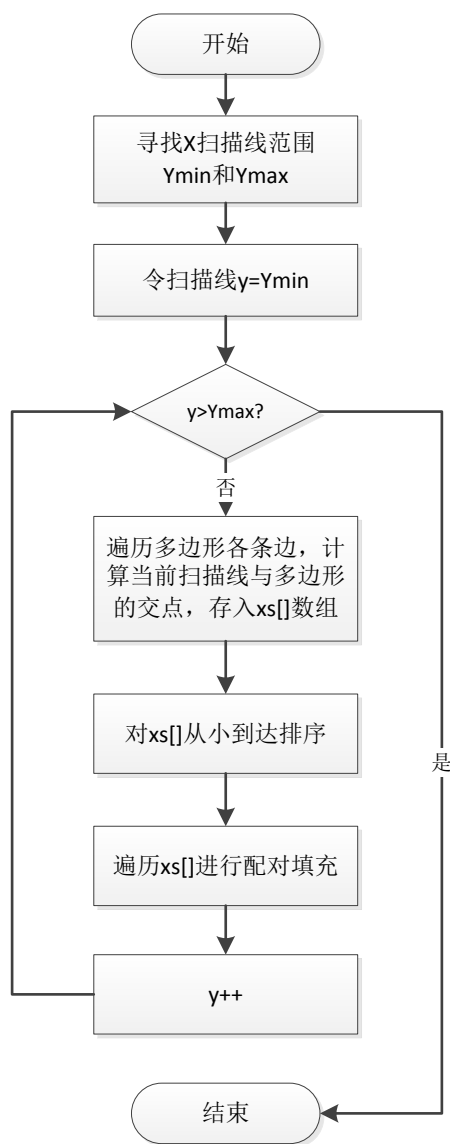


图 3-1 扫描线填充算法流程图

多边形的扫描线填充算法是一种简单而有效的方法，填充的基本思想是，使

用水平（也可使用垂直）扫描线与多边形各条边求交，将交点按照 X 坐标大小排序，然后再将交点配对（0、1 配对，2、3 配对...）进行区间填充。扫描线填充的算法流程如图 3-1 所示。

扫描线算法每条扫描线都要进行求交和排序操作效率不高，为了提高填充效率可以使用改进的扫描线算法。该算法需要使用边表桶和有效边表两个数据结构，这两个数据结构均以多边形的边节点作为基本存储单元，图 3-2（a）给出了边节点数据结构。边节点数据结构中 x 用于记录边与水平扫描线交点的 x 坐标；ymax 是边的 Y 值较大端坐标；1/k 为斜率的倒数即 $1/k = \Delta x / \Delta y$ ；next 指向下一个边结构。

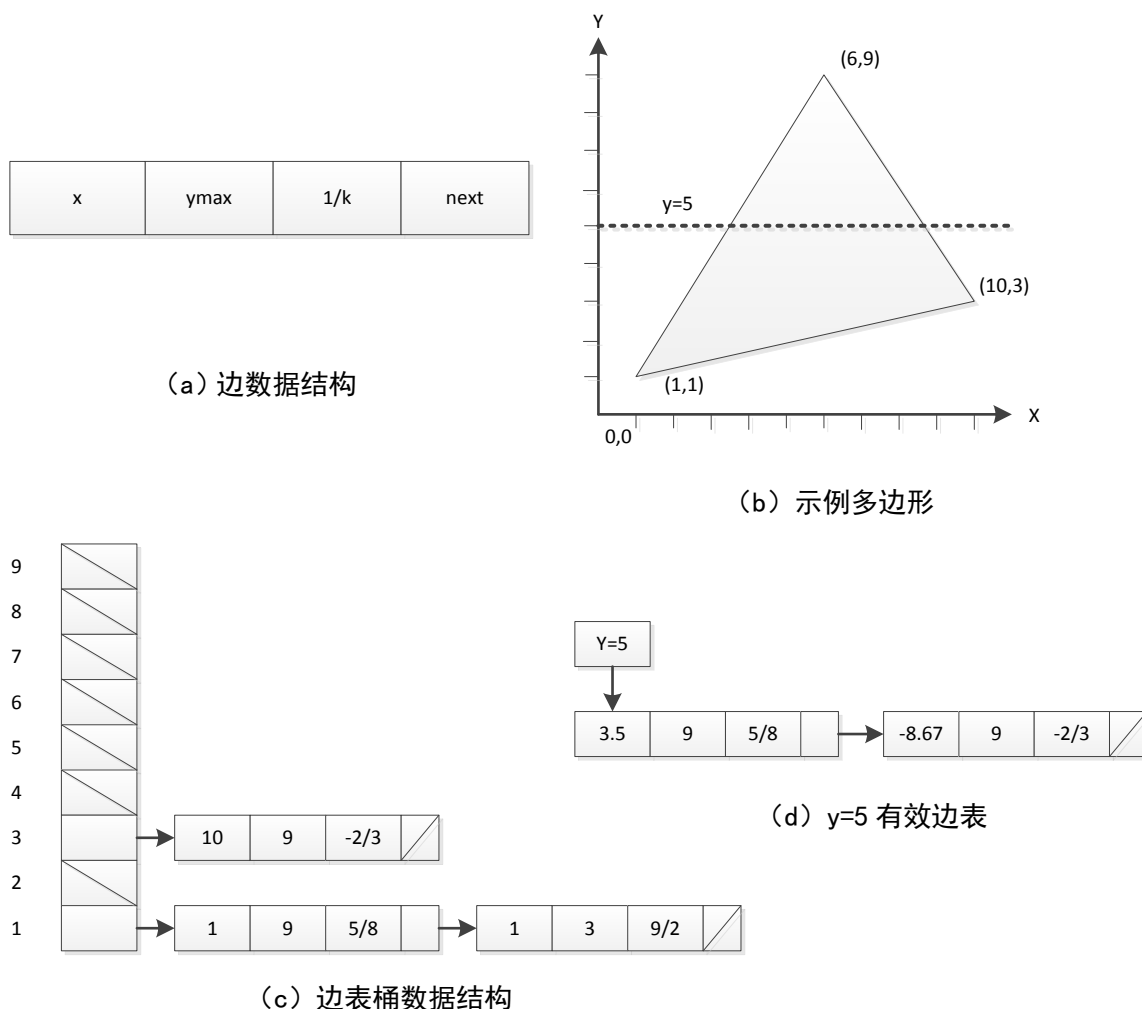


图 3-2 改进扫描线算法数据结构

图 3-2（b）是一个简单的示例多边形，X 扫描线范围在 0~9 之间。算法首先针对多边形建立边表桶数据结构，桶的长度与有效扫描线条数相同，如图 3-2

(c) 所示左边的桶长度是 9，对应图 (b) 中扫描线 1~9。将多边形中每条边按照其 Y 值较小端置入到边表桶中，并填入边结构所需数据项，即可建立图 (c) 所示的数据结构。

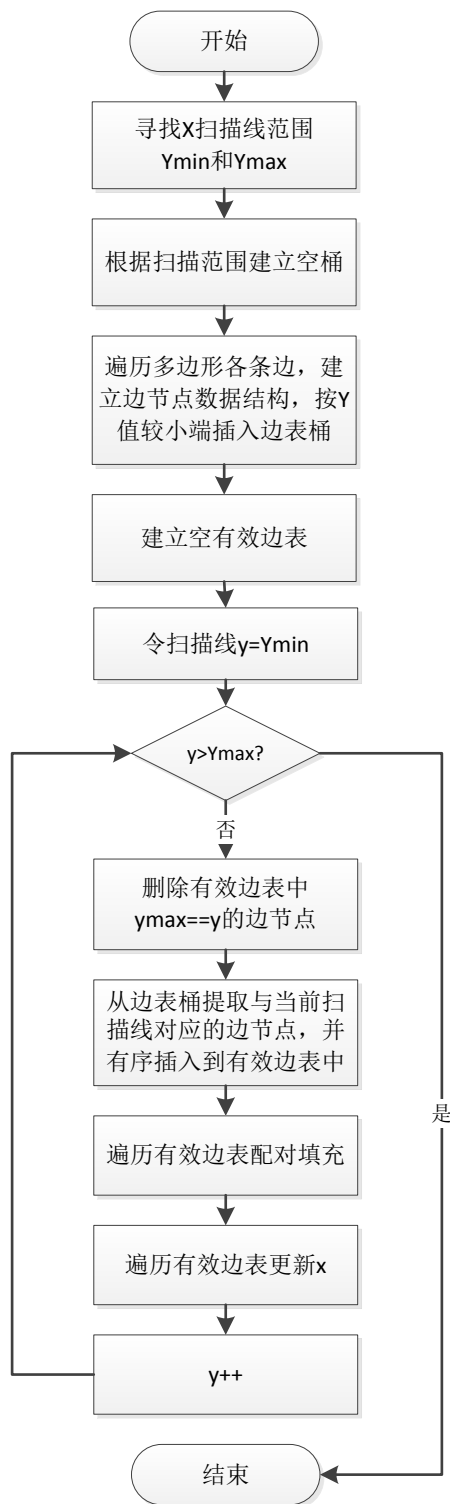


图 3-3 改进扫描线算法流程图

建立边表桶后，设置一个空的有效边表，然后由下至上遍历所有扫描线。对

于每一条扫描线，首先删除 y_{\max} 与当前扫描线值相同的边节点；然后检查边表中对应扫描线位置处是否有新的节点加入，如有则将它们按照 x 字段的大小（如果 x 相等，则再比较 $1/k$ 字段）有序插入到有效边表中；遍历有效边表中所有节点进行配对填充；填充完成后再次遍历有效边表将边节点的 x 字段增加 $1/k$ 。改进的扫描线填充算法如图 3-3 所示。

除了完成上述两种填充算法外，本次实验还要实现多边形交互编辑功能，多边形交互编辑的基本思路与实验 2 直线编辑一致，使用 GPolygon 对象作为模型，存储多边形顶点坐标，在 onDisplay 函数中实现多边形显示；在 onMouse 和 onMotion 函数中实现用户交互控制，完成多边形交互编辑。交互编辑操作要求：在没有控制点的位置上按下鼠标左键增加控制点；在已有控制点位置上按下鼠标左键并拖拽修改控制点坐标；按下鼠标左键的同时按下 Shift 键删除已有控制点。

3.3 主要实验步骤

(1) 打开工程

启动 VC6 或 VS2010 开发环境，对于 VC6 选择菜单：**【File->Open Workspace】**，在弹出的 OpenWorkspace 对话框中选择“实验 3”目录中的 FillPoly.dsw 文件，然后按下**【打开】**按钮，打开实验项目。对于 VS2010 选择菜单：**【文件->打开->项目/解决方案】**，在弹出的打开项目对话框中选择“实验 3”目录中的 FillPoly.sln 文件，然后按下打开项目，打开实验项目。注意：一个项目或解决方案通常由多个源文件及相关文件构成，因此仅仅打开 main.cpp 文件进行编译是无法正常通过的，请务必使用打开项目菜单进行操作。

(2) 增加 GPolygon 对象，实现 onDisplay 函数

现有的 main.cpp 文件仅实现了基本的投影和窗口显示功能，为了实现多边形显示和交互编辑，首先需要引入 GPolygon 类的对象。GPolygon 是多边形模板类，提供了多边形顶点存取功能，该类实现的功能如表 3-1 所示。

表 3-1 GPolygon2 类函数列表

函数名	函数功能
GPolygon2	默认构造函数，初始化空多边形
GPolygon2(const Polygon2<T> other)	拷贝构造函数

GPolygon2 & operator = (const GPolygon2 &other)	赋值运算符，将 other 赋值给当前 GPolygon2 对象
GPoint2<T> at(int index) const	返回第 index 顶点对象
bool set(int index, const GPoint2<T> &pt)	设置第 index 顶点对象
operator const GPoint2<T> * () const	将 GPolygon2 对象转换成 GPoint2<T>常指针
operator GPoint2<T> * ()	将 GPolygon2 对象转换成 GPoint2<T>指针
void addFirst(const GPoint2<T> &pt)	在起始位置处添加控制点
void addLast(const GPoint2<T> &pt)	在末尾处添加控制点
bool insert(int index, const GPoint2<T> &pt)	在任意位置处插入控制点，如果插入位置无效返回 false
bool removeFirst()	删除起始位置处控制点
bool removeLast()	删除末尾处控制点
bool remove(int index)	删除任意位置处控制点，如果指定位置无效返回 false
void clear()	清空所有控制点
bool empty()	返回多边形是否不包含控制点
int count() const	返回控制点数目
int findByCoord(T x, T y, T w = 5, T h = 5)	查找指定范围内距离坐标(x, y)最近的控制点，返回控制点下标，返回-1 表示附近无有效控制点

打开 main.cpp 文件，并在文件的首部添加如下代码：

```

1  #include <gl/glut.h>
2  #include <stdio.h>
3  #include <windows.h>
4  #include "tools.h"
5  #include "gpoint2.h"
6  #include "gpolygon2.h"
7
8  // 样条控制点数组
9  GPolygon2i gPolygon;
10 // 鼠标左键选中的控制点
11 int gSelectedIndex = -1;
12 // Shift 键是否按下
13 bool gIsShiftDown = false;
14 // 是否已完成多边形填充
15 bool gIsFillPolygon = false;
```

找到 onDisplay 函数，在原有 onDisplay 函数 glClear 调用之后加入如下代码：

```

57 void onDisplay()
58 {
59     glClearColor(1, 1, 1, 0);
60     glClear(GL_COLOR_BUFFER_BIT);
61
62     // 填充多边形
63     glColor3f(1, 1, 0);
64     if(gIsFillPolygon) gltFillPolygon(gPolygon, gPolygon.count());
65     // 绘制控制点
66     drawPoints();
67     // 绘制多边形外框线
```

```

68     drawCtrlLines();
69     // 绘制顶点标注
70     drawTags();
71
72     glutSwapBuffers(); // 交换缓存
73 }

```

上述代码中的 `drawPoints()`、`drawCtrlLines()`和 `drawTags()`函数均为多边形显示的辅助函数，在 `onDisplay` 函数前已经定义并实现，有需要的同学可以自行阅读参考。`gltFillPolygon()`函数为多边形填充函数，定义在 `tools.cpp` 文件中，将在之后实现。

(3) 实现鼠标交互编辑功能

在鼠标事件 `onMouse` 回调函数中输入下面代码：

```

98 void onMouse(int button, int state, int x, int y)
99 {
100     if(button == GLUT_LEFT_BUTTON)
101     {
102         if(state == GLUT_DOWN)
103         {
104             if(gIsFillPolygon)
105             {
106                 gPolygon.clear();
107                 gIsFillPolygon = false;
108             }
109
110             int index = gPolygon.findByCoord(x, y);
111             if(index < 0) // 添加控制点
112             {
113                 index = gPolygon.count();
114                 gPolygon.addLast(GPoint2i(x, y));
115                 gSelectedIndex = index;
116                 gIsFillPolygon = false;
117             }
118             else // 修改(删除)已有控制点
119             {
120                 if(glutGetModifiers() & GLUT_ACTIVE_SHIFT) // 删除
121                 {
122                     gPolygon.remove(index);
123                 }
124                 else
125                 {
126                     gSelectedIndex = index;
127                 }
128                 gIsFillPolygon = false;
129             }
130         }
131         else if(state == GLUT_UP)
132         {
133             if(gSelectedIndex >= 0)
134             {
135                 gPolygon[gSelectedIndex].set(x, y);
136                 glutPostRedisplay();
137                 gSelectedIndex = -1;

```

```

138     }
139 }
140     glutPostRedisplay();
141 }
142 else if(button == GLUT_RIGHT_BUTTON)
143 {
144     gIsFillPolygon = true;
145     glutPostRedisplay();
146 }
147 }

```

在鼠标拖拽事件 `onMotion` 回调函数中输入下面代码：

```

88 void onMotion(int x, int y)
89 {
90     if(gSelectedIndex >= 0)
91     {
92         gPolygon[gSelectedIndex].set(x, y);
93         glutPostRedisplay();
94     }
95 }

```

编写完成上述代码后，构建并运行程序得到如图 3-4 所示的窗口，在窗口中按下鼠标左键并拖拽即可添加并移动新的控制点；如果在已有控制点附近按下鼠标左键则可以拖拽修改该控制点；如果按下左键的同时按下 `Shift` 键即可删除控制点。点击鼠标右键则可调用填充函数 `gltFillPolygon()`，不过该函数目前还未实现，因而并没有任何效果。

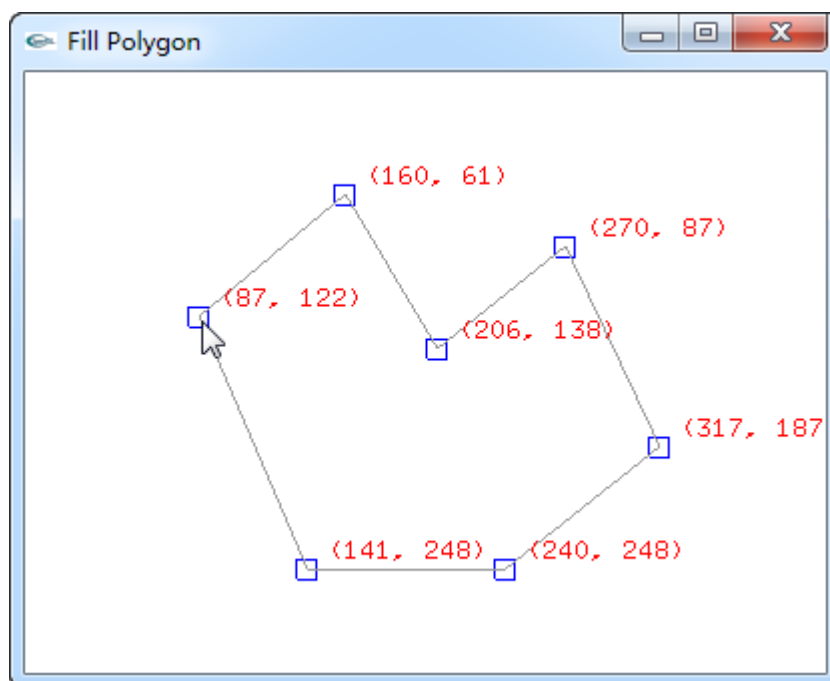


图 3-4 多边形编辑效果

(4) 实现扫描线填充算法

打开文件 `tools.cpp` 文件并定位到 `gltFillPolygon()` 函数，在函数中输入以下代码：

```

1 void gltFillPolygon(const GPoint2i *points, int n)
2 {
3     int i, j, iy, min, max, xm, nc;
4     static int xs[256];
5     GPoint2i pt0, pt1;
6
7     if(n < 3) return ;
8     // 寻找扫描线范围
9     min = max = points[0].y();
10    for(i=1; i<n; i++)
11    {
12        if(min > points[i].y()) min = points[i].y();
13        if(max < points[i].y()) max = points[i].y();
14    }
15
16    for(iy=min; iy<=max; iy++)
17    {
18        // (1)用当前扫描线 y = iy 与多边形各条边求交, 记录交点
19        nc = 0;
20        for(i=0; i<n; i++)
21        {
22            pt0 = points[i];
23            pt1 = points[(i+1)%n];
24            // 跳过水平边
25            if(pt0.y() == pt1.y()) continue ;
26            if(pt0.y() > pt1.y()) swap(pt0, pt1);
27            // 如果扫描线不在边有效范围跳过
28            if(iy < pt0.y() || iy >= pt1.y()) continue ;
29            // 计算交点 x 坐标
30            xm = pt0.x() + (iy - pt0.y()) *
31                (pt1.x() - pt0.x()) / (pt1.y() - pt0.y());
32            // 有序插入交点 x 坐标
33            for(j=nc-1; j>=0; j--)
34            {
35                if(xs[j] <= xm) break ;
36                xs[j+1] = xs[j];
37            }
38            xs[j+1] = xm;
39            nc++;
40            // 如果交点个数太多跳出
41            if(nc == 256) break;
42        }
43        // 配对填充
44        for(i=0; i<nc; i+=2)
45        {
46            gltLine2i(xs[i], iy, xs[i+1], iy);
47        }
48    }
49 }

```

完成上述代码后，构建并运行项目，在窗口中交互编辑一个多边形，然后按下鼠标右键，即可得到一个用黄色填充的多边形，如图 3-5 所示。为了测试不同多边形填充效果，在点击右键填充后，可以继续用鼠标左键重新编辑一个多边形再次进行填充测试。

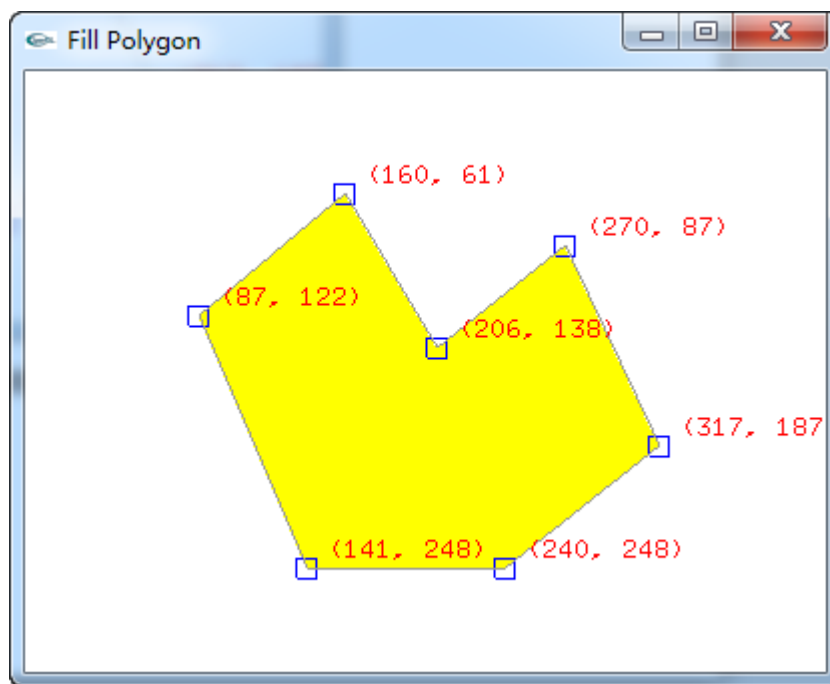


图 3-4 多边形填充效果

(5) 增加边节点数据结构

在 tools.cpp 文件的 gltFillPolygon() 函数前, 输入以下代码, 定义边界点数据结构:

```

1  struct EdgeNode    // 边节点
2  {
3      int yMax;
4      double x, invk;
5      EdgeNode *next;
6
7      EdgeNode()
8      {
9          yMax = 0;
10         x = 0;
11         invk = 0;
12         next = 0;
13     }
14 };

```

(6) 实现改进的扫描线填充算法

删除或注释 gltFillPolygon() 中已实现的扫描线填充代码, 然后输入如下代码:

```

15 void gltFillPolygon(const GPoint2i *points, int n)
16 {
17     if(n < 3) return ;
18
19     int i, index;
20     int min, max;
21     EdgeNode **edgeBucket;    // 边表桶
22     EdgeNode *p, *q, *pEdge, *aet;
23     GPoint2i pt0, pt1, pt;
24
25     // 寻找多边形 Y 方向扫描范围
26     min = max = points[0].y();

```



```

27     for(i=1; i<n; i++)
28     {
29         pt = points[i];
30         if(min > (int)pt.y()) min = (int)pt.y();
31         if(max < (int)pt.y()) max = (int)pt.y();
32     }
33
34     // 建立边表桶
35     edgeBucket = new EdgeNode *[max-min+1];
36     memset(edgeBucket, 0, sizeof(EdgeNode *)*(max-min+1));
37     // 加入边节点到桶中
38     for(i=0; i<n; i++)
39     {
40         pt0 = points[i];
41         pt1 = points[(i+1)%n];
42         // 跳过水平边
43         if(pt0.y() == pt1.y()) continue;
44         pEdge = new EdgeNode;
45         if(pt0.y() > pt1.y()) // 保证 pt0 为 y 值较小端
46         {
47             pt = pt0;
48             pt0 = pt1;
49             pt1 = pt;
50         }
51         pEdge->yMax = pt1.y();
52         pEdge->x = pt0.x();
53         pEdge->invk = (double)(pt1.x()-pt0.x()) / (pt1.y()-pt0.y());
54         // 边表项插入链表头部
55         index = pt0.y() - min;
56         pEdge->next = edgeBucket[index];
57         edgeBucket[index] = pEdge;
58     }
59
60     // 扫描填充
61     aet = new EdgeNode;
62     for(i=min; i<=max; i++)
63     {
64         // (1)删除 y=ymax 的边项
65         p = aet;
66         while(p->next != 0)
67         {
68             pEdge = p->next;
69             if(pEdge->yMax == i)
70             {
71                 p->next = pEdge->next;
72                 delete pEdge;
73             }
74             else p = p->next;
75         }
76         // (2)将边表桶中的边表项加入活动边表
77         index = i - min;
78         p = edgeBucket[index];
79         while(p != 0)
80         {
81             pEdge = p;
82             p = p->next;
83             q = aet;
84             while(q->next != 0)
85             {

```

```
86         if((int)q->next->x > (int)pEdge->x) break ;
87         else if((int)q->next->x == (int)pEdge->x)
88         {
89             if(q->next->invk > pEdge->invk) break ;
90         }
91         q = q->next;
92     }
93     pEdge->next = q->next;
94     q->next = pEdge;
95 }
96 // (3)配对填充
97 p = aet->next;
98 while(p != 0)
99 {
100     gltLine2i(p->x, i, p->next->x, i);
101     p = p->next->next;
102 }
103 // (4)修改 AET 表项, 增加 x 值
104 p = aet->next;
105 while(p != 0)
106 {
107     p->x += p->invk;
108     p = p->next;
109 }
110 }
111 // 释放剩余内存
112 delete aet;
113 delete []edgeBucket;
114 }
```

完成上述修改后，构建并运行程序，在生成的窗口中编辑多边形，点击右键即可进行多边形填充。

实验4 直线和多边形裁剪

4.1 实验目的

- (1) 理解直线的编码算法原理，实现算法程序；
- (2) 理解多边形的逐边裁剪算法原理，实现算法程序。

4.2 实验原理

4.2.1 直线编码裁剪原理

一个复杂的画面中可能包含若干条直线，为了实现直线的快速裁剪，可以先对直线顶点进行编码，为直线的顶点分配一个表示其相对位置的 4 位二进制代码 $C_L C_b C_r C_L$ 。此代码称为区域码。区域码按照端点与窗口边界的相对位置编码，即区域码的 4 位分别代表端点位于窗口的上、中、下、左、右。编码域对应坐标轴区如下图所示：

1001	0001	0101
1000	0000	0100
1010	0010	0110

图 4-1 编码区域

区域码各位的值可以通过对端点坐标 $P(x, y)$ 与窗口边界的比较求得。如果 $x < x_{\min}$ ，则区域码的第一位为 1，否则为 0，如果 $x > x_{\max}$ ，则区域码的第二位为 1，否则为 0，根据图编码公式 (4-1)，依次类推可得出坐标 $P(x, y)$ 的编码值。

$$C_t = \begin{cases} 1 & x < x_{\min} \\ 0 & \text{other} \end{cases} \quad C_b = \begin{cases} 1 & x > x_{\max} \\ 0 & \text{other} \end{cases} \quad C_r = \begin{cases} 1 & y < y_{\min} \\ 0 & \text{other} \end{cases} \quad C_l = \begin{cases} 1 & y > y_{\max} \\ 0 & \text{other} \end{cases}$$

(4-1) 编码公式

对所有直线的端点都建立了区域码之后,就可按区域码判断直线在窗口之内或窗口之外。这可分为如下三种情况:

(1) 若一直线的两个端点的区域码均为 0000, 即 $\text{code1}=0$ 且 $\text{code2}=0$, 则此直线在窗口边界之内, 应全部保留。

(2) 若一直线的两个端点的区域码的同一位同时为 1, 即 $\text{code1} \& \text{code2} \neq 0$, 则此直线全部在窗口边界之外, 应全部裁掉。例如, 若一直线的一个端点的区域码为 1001, 另一个端点的区域码为 0101, 则此两端点的区域码的最后一位均为 1, 说明此两端点均在窗口边界之上, 因此, 直线在窗口边界之外, 应予裁剪。

(3) 以上两种情况之外的直线, 有可能穿过窗口, 也有可能不穿过窗口, 如图 4-2 所示。图中所示一条直线 (P1P2) 穿过窗口, 另一直线 (P3P4) 不穿过窗口。对这类直线可以进行如下处理: 求出直线与窗口的交点, 在交点处把线段分为两段。其中一段完全在窗口外, 可弃之。然后对另一段重复上述处理。

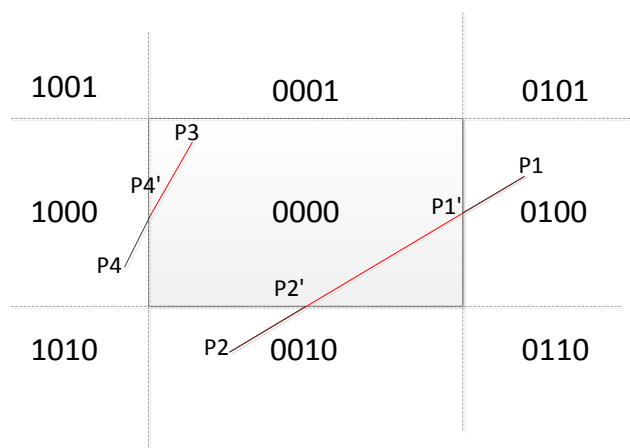


图 4-2 直线与编码域

对上述算法思路进行整理得到直线编码裁剪的算法流程如图 4-3 所示。

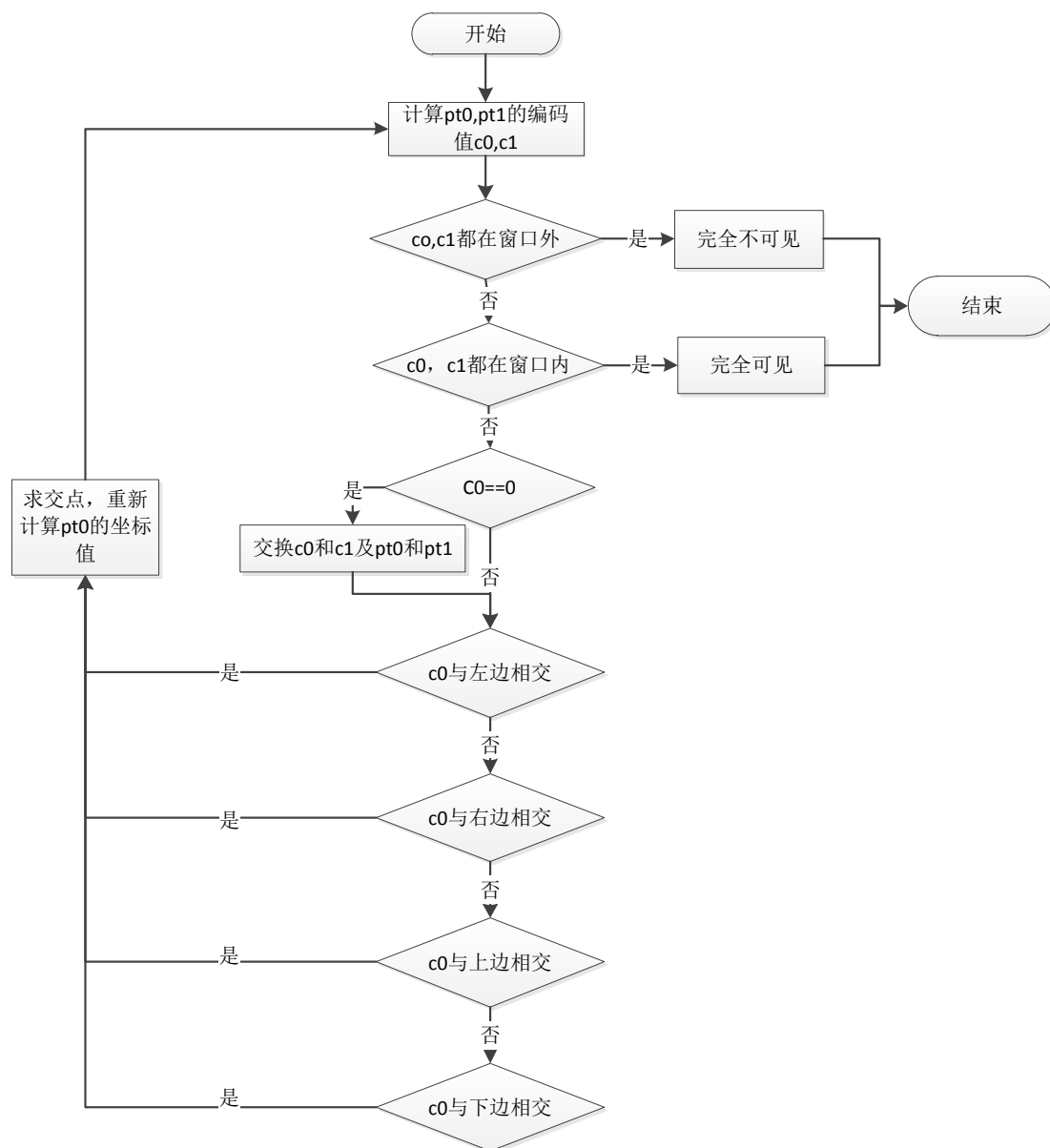


图 4-3 直线编码裁剪算法流程图

4.2.2 多边形逐边裁剪原理

多边形逐边算法基本思想是将多边形界面作为一个整体，每次用窗口的一条边界和多边形进行裁剪。裁剪时，选取一条窗口边界线对多边形求交点，顺序地测试多边形的输入顶点，根据向量与边界的关系，选择保留顶点，删除顶点或插入新的顶点，从而得到一个新的多边形顶点序列。然后以此新的顶点序列作为输入顶点，再用第二条窗口边界线进行裁剪，又得到一个更新的多边形顶点序列。依次下去，直到四条窗口边界线对多边形进行了裁剪，最后输入的顶点序列即为所求的裁剪好了的多边形。

考虑窗口边界线以及延长线构成的裁剪线把多边形各边分成两个部分: 可见一侧; 不可见一侧。如下图 4-4 所示, 依序考虑多边形的各条边。假设当前处理的多边形的边为 SP (箭头表示顺序关系, S 为前一点, P 为当前点), 边 SP 与裁剪线的位置关系只有下面四种情况:

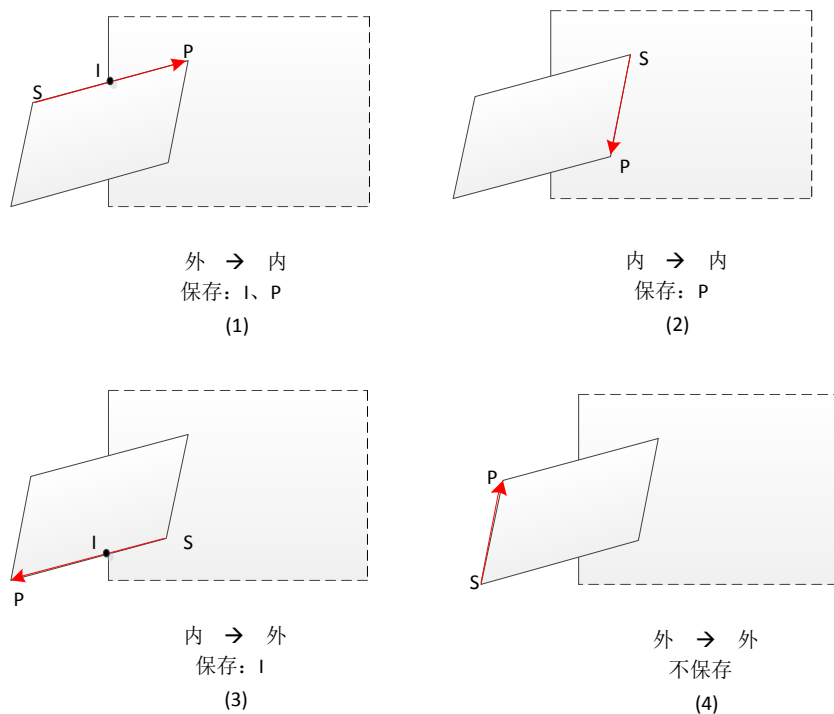


图 4-4 左窗口连续处理多边形顶点对

- (1) S 在外侧, P 在内侧。则交点 I 、当前点 P 保存到新多边形中。
- (2) S 、 P 均在左侧, 则当前点 P 保存到新多边形中。
- (3) S 在左侧, P 在外侧。则交点 I 保存到新多边形中。
- (4) S 、 P 均在外侧。则没有点被保存到新多边形中。

上述算法仅用一条裁剪边对多边形进行裁剪, 得到一个顶点序列, 作为下一条裁剪边处理过程的输入, 对于每一条裁剪边, 算法框图同上, 只是判断点在窗口哪一侧以及求线段 SP 与裁剪边的交点算法应随之改变, 完整的多边形逐边裁剪算法流程如图 4-5 和 4-6 所示。其中图 4-5 是裁剪算法整体流程, 图 4-6 是处理一条

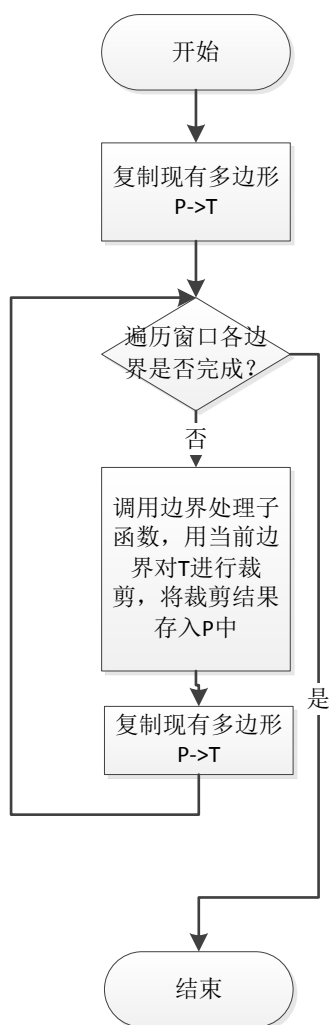


图 4-5 逐边裁剪算法流程

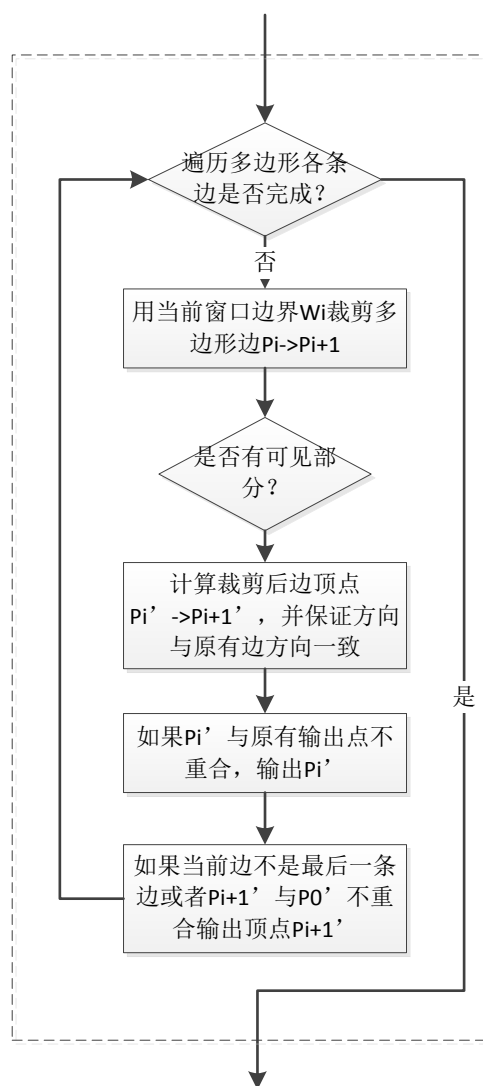


图 4-6 处理线段 SP 子过程

4.3 主要实验步骤

4.3.1 直线编码裁剪主要步骤

(1) 打开工程

启动 VC6 或 VS2010 开发环境, 对于 VC6 选择菜单: **【File->Open Workspace】**, 在弹出的 OpenWorkspace 对话框中选择“实验 4-1”目录中的 clip.dsw 文件, 然后按下 **【打开】** 按钮, 打开实验项目。对于 VS2010 选择菜单: **【文件->打开->项目/解决方案】**, 在弹出的打开项目对话框中选择“实验 4-1”目录中的 clip.sln 文件, 然后按下打开项目, 打开实验项目。

(2) 计算线段编码

打开 tools.cpp 文件，找到 clipCode 函数，添加如下代码：

```

115 char clipCode(const GPoint2d &pt, const GRect2d &rc)
116 {
117     char code = 0;
118     if(pt.y() > rc.y1()) code |= 0x01;
119     else if(pt.y() < rc.y0()) code |= 0x02;
120     if(pt.x() > rc.x1()) code |= 0x04;
121     else if(pt.x() < rc.x0()) code |= 0x08;
122     return code;
123 }

```

上述代码可计算出点 pt (x,y) 的编码值，编码值存储到 char 变量的低 4 位。

(3) 实现编码裁剪功能

找到 gtlLineClip2d 函数，添加如下代码：

```

124 bool gtlLineClip2d(GPoint2d &pt0, GPoint2d &pt1, const GRect2d &rc)
125 {
126     char c0, c1;
127
128     while(true)
129     {
130         c0 = clipCode(pt0, rc);
131         c1 = clipCode(pt1, rc);
132         if((c0 & c1) != 0) return false;
133         if(c0 == 0 && c1 == 0) return true;
134
135         if(c0 == 0)
136         {
137             swap(pt0, pt1);
138             swap(c0, c1);
139         }
140
141         if(c0 & 0x01) // 在 Yt 之上
142         {
143             pt0.setX(pt0.x()-(pt0.y()-rc.y1()*(pt0.x()-pt1.x()))/(pt0.y()-pt1.y()));
144             pt0.setY(rc.y1());
145         }
146         else if(c0 & 0x02) // 在 Yb 之下
147         {
148             pt0.setX(pt0.x()-(pt0.y()-rc.y0()*(pt0.x()-pt1.x()))/(pt0.y()-pt1.y()));
149             pt0.setY(rc.y0());
150         }
151         else if(c0 & 0x04) // 在 Xr 之右
152         {
153             pt0.setY(pt0.y()-(pt0.x()-rc.x1()*(pt0.y()-pt1.y()))/(pt0.x()-pt1.x()));
154             pt0.setX(rc.x1());
155         }
156         else if(c0 & 0x08) // 在 Xl 之左
157         {
158             pt0.setY(pt0.y()-(pt0.x()-rc.x0()*(pt0.y()-pt1.y()))/(pt0.x()-pt1.x()));
159             pt0.setX(rc.x0());
160         }
161     }
162 }

```


完成上述代码后，构建并运行项目，在窗口中按下鼠标左键并拖拽，绘制一条直线，松开鼠标左键后，程序将对输入的直线进行编码裁剪，得到类似图 4-7 所示的裁剪效果。

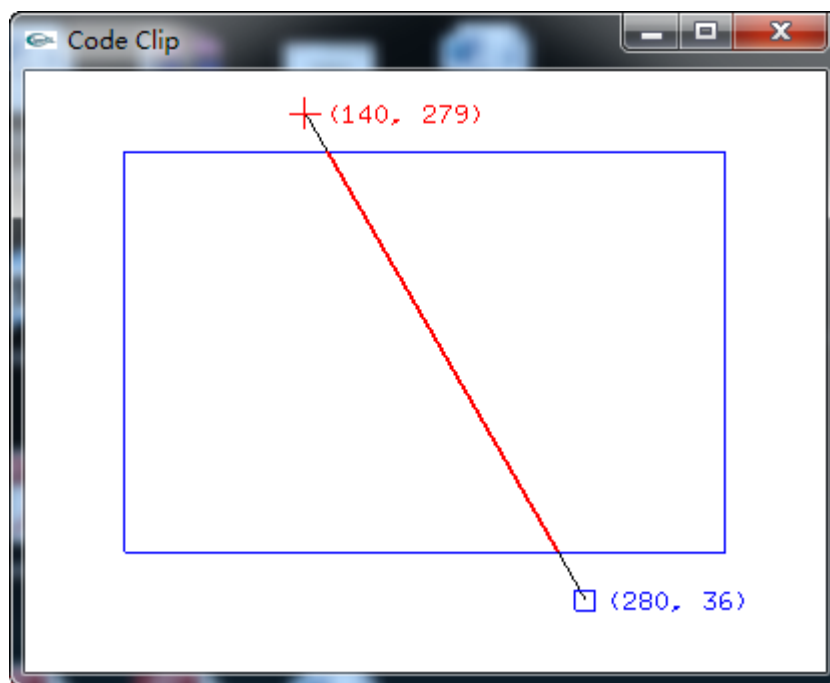


图 4-7 直线编码裁剪效果图

4.3.2 多边形逐边裁剪主要步骤

(1) 打开工程

启动 VC6 或 VS2010 开发环境，对于 VC6 选择菜单：**【File->Open Workspace】**，在弹出的 OpenWorkspace 对话框中选择“实验 4-2”目录中的 clip-polygon.dsw 文件，然后按下**【打开】**按钮，打开实验项目。对于 VS2010 选择菜单：**【文件->打开->项目/解决方案】**，在弹出的打开项目对话框中选择“实验 4-2”目录中的 clip-polygon.sln 文件，然后按下打开项目，打开实验项目。

(2) 实现窗口边界对多边形的裁剪子函数，打开 tools.cpp，找到 winEdgeClip2d 函数，添加如下代码：

```
1 void winEdgeClip2d(int edge, double v, const GPolygon2d &src, GPolygon2d &dst)
2 {
3     bool isSwap;
4     int i, j, n;
5     double d;
6
7     dst.clear();
8     n = src.count();
9     for(i=0; i<n; i++)
```

```

10  {
11      GPoint2d pt0 = src[i];
12      GPoint2d pt1 = src[(i+1)%n];
13
14      isSwap = false;
15
16      if(edge == 0)
17      {
18          if(pt0.x() < v && pt1.x() < v) continue ;
19
20          if(pt0.x() > pt1.x())
21          {
22              isSwap = true;
23              swap(pt0, pt1);
24          }
25          if(pt0.x() < v && pt1.x() > v)
26          {
27              pt0.setY(pt0.y() + (v-pt0.x())*(pt1.y()-pt0.y()/(pt1.x()-pt0.x())));
28              pt0.setX(v);
29          }
30      }
31      else if(edge == 1)
32      {
33          if(pt0.y() < v && pt1.y() < v) continue ;
34
35          if(pt0.y() > pt1.y())
36          {
37              isSwap = true;
38              swap(pt0, pt1);
39          }
40          if(pt0.y() < v && pt1.y() > v)
41          {
42              pt0.setX(pt0.x() + (v-pt0.y()*(pt1.x()-pt0.x()/(pt1.y()-pt0.y())));
43              pt0.setY(v);
44          }
45      }
46      else if(edge == 2)
47      {
48          if(pt0.x() > v && pt1.x() > v) continue ;
49
50          if(pt0.x() > pt1.x())
51          {
52              isSwap = true;
53              swap(pt0, pt1);
54          }
55          if(pt0.x() < v && pt1.x() > v)
56          {
57              pt1.setY(pt0.y() + (v-pt0.x()*(pt1.y()-pt0.y()/(pt1.x()-pt0.x())));
58              pt1.setX(v);
59          }
60      }
61      else if(edge == 3)
62      {
63          if(pt0.y() > v && pt1.y() > v) continue ;
64
65          if(pt0.y() > pt1.y())
66          {
67              isSwap = true;
68              swap(pt0, pt1);
69          }

```

```

70         if(pt0.y() < v && pt1.y() > v)
71         {
72             pt1.setX(pt0.x() + (v-pt0.y()*(pt1.x()-pt0.x()/(pt1.y()-pt0.y())));
73             pt1.setY(v);
74         }
75     }
76
77     if(isSwap) swap(pt0, pt1);
78
79     if(dst.count() <= 0)
80     {
81         dst.addLast(pt0);
82     }
83     else
84     {
85         j = dst.count()-1;
86         d = sqrt((dst[j].x()-pt0.x()*(dst[j].x()-pt0.x() +
87             (dst[j].y()-pt0.y()*(dst[j].y()-pt0.y())));
88         if(d > 1e-8)
89         {
90             dst.addLast(pt0);
91         }
92     }
93     dst.addLast(pt1);
94 }
95
96 j = dst.count()-1;
97 d = sqrt((dst[j].x()-dst[0].x()*(dst[j].x()-dst[0].x() +
98     (dst[j].y()-dst[0].y()*(dst[j].y()-dst[0].y())));
99 if(d < 1e-8) dst.removeLast();
100 }

```

(3) 实现多边形逐边裁剪函数，打开 tools.cpp，找到 gltPolyClip2d 函数，添加如下代码：

```

101 void gltPolyClip2d(const GPolygon2d &src, GPolygon2d &dst, const GRect2d &rc)
102 {
103     int i, n;
104     double v[4];
105     GPolygon2d tmp;
106
107     n = src.count();
108     dst.clear();
109     tmp = src;
110     v[0] = rc.x0();
111     v[1] = rc.y0();
112     v[2] = rc.x1();
113     v[3] = rc.y1();
114
115     for(i=0; i<4; i++)
116     {
117         winEdgeClip2d(i, v[i], tmp, dst);
118         tmp = dst;
119     }
120 }

```

完成上述代码后，构建并运行项目，在窗口中交互编辑一个多边形，然后按下鼠

标右键，程序将对输入的多边形进行裁剪，得到类似图 4-8 所示的裁剪效果。

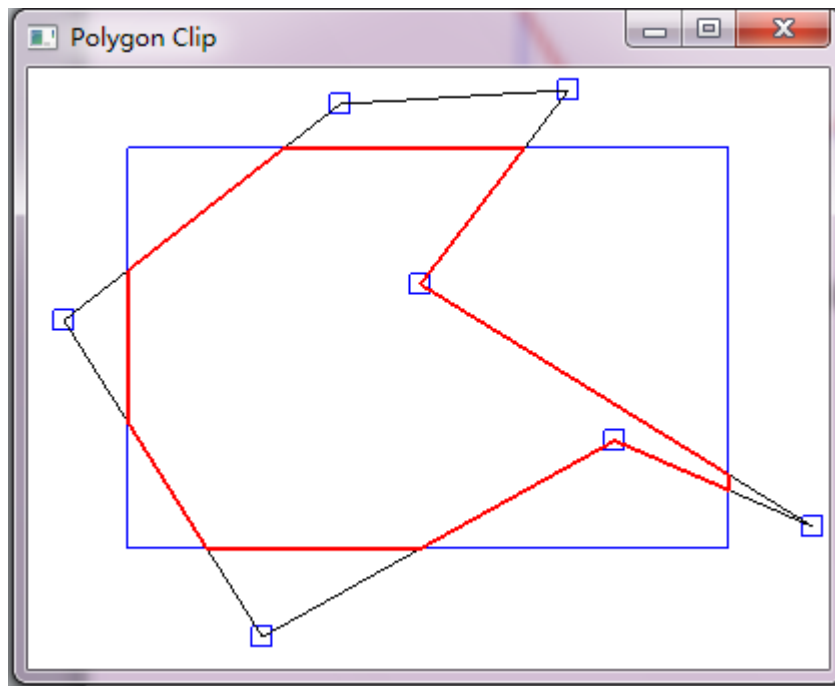


图 4-8 多边形逐边裁剪效果图

实验5 OpenGL 三维图形程序设计

5.1 实验目的

- (1) 实现一个简单的三维 OpenGL 程序并测试几种投影矩阵，理解三维投影变换算法；
- (2) 掌握三维 OpenGL 程序的基本结构；
- (3) 掌握常用的正、斜平行投影矩阵及透视投影矩阵。

5.2 实验原理

屏幕上看到的 3D 图形其实并不是真正 3D 的。我们使用 3D 概念和术语来描述物体，然后这些 3D 数据被“压扁”到一个 2D 的计算机屏幕上。这种将 3D 数据被“压扁”成 2D 数据的处理过程叫做投影（projection），投影在大体上可分为平行影和透视投影。

平行投影是在一束平行光线照射下形成的投影。也可理解为投影中心到投影平面的距离为无穷大时投影。平行投影可分为正平行投影（投影方向垂直于投影面）和斜平行投影（投影方向不垂直于投影面）。

- OpenGL 平行投影函数：

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

left: 左裁剪面坐标；

right: 右裁剪面坐标；

bottom: 下裁剪面坐标；

top: 上裁剪面坐标；

near: 近裁剪面距离；

far: 远裁剪面距离。

函数 `glOrtho()` 定义了一个由六个平面构成的长方体，该长方体也叫作视景体（如图 5-1 所示）。位于视景体内的模型在投影后是可见的，位于视景体外的模型将被裁剪而不可见。

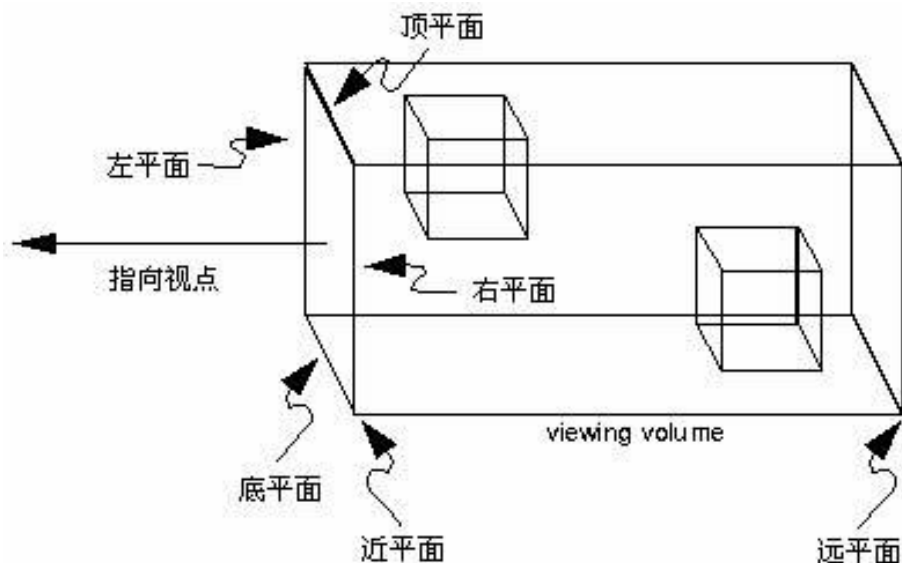


图 5-1 正投影视景体

透视投影属于中心投影。透视投影图简称为透视图或透视，它是从某个投射中心将物体投射到单一投影面上所得到的图形。透视图与人们观看物体时所产生的视觉效果非常接近，所以它能更加生动形象地表现建筑外貌及内部装饰。在已有实景实物的情况下，通过拍照或摄像即能得到透视图；对于尚在设计、规划中的建筑物则作图（手工或计算机）的方法才能画出透视图。透视图以渲染、配景，使之成为形象逼真的效果图。由于是中心投影，因此平行投影中的一些重要性质（如平行性、定比性等）和作图规律，在这里已不适用。

透视投影是用中心投影法将形体投射到投影面上，从而获得的一种较为接近视觉效果的面投影图。它具有消失感、距离感、相同大小的形体呈现出有规律的变化等一系列的透视特性，能逼真地反映形体的空间形象。

透视投影符合人们心理习惯，即离视点近的物体大，离视点远的物体小，远到极点即为消失，成为灭点。它的视景体类似于一个顶部和底部都被切除掉的棱锥，也就是棱台。这个投影通常用于动画、视觉仿真以及其它许多具有真实性反映的方面。

- OpenGL 透视投影函数一：

```
void glFrustum(GLdouble left, GLdouble Right, GLdouble bottom, GLdouble top,
               GLdouble near, GLdouble far);
```

left: 左裁剪面坐标;

right: 右裁剪面坐标;

bottom: 下裁剪面坐标;

top: 上裁剪面坐标;

near: 近裁剪面距离 (透视投影该参数必须 >0);

far: 远裁剪面距离。

该函数定义了一个平截头体 (如图 5-2 所示)。与正投影相似, 位于视景体内的模型才能显示。

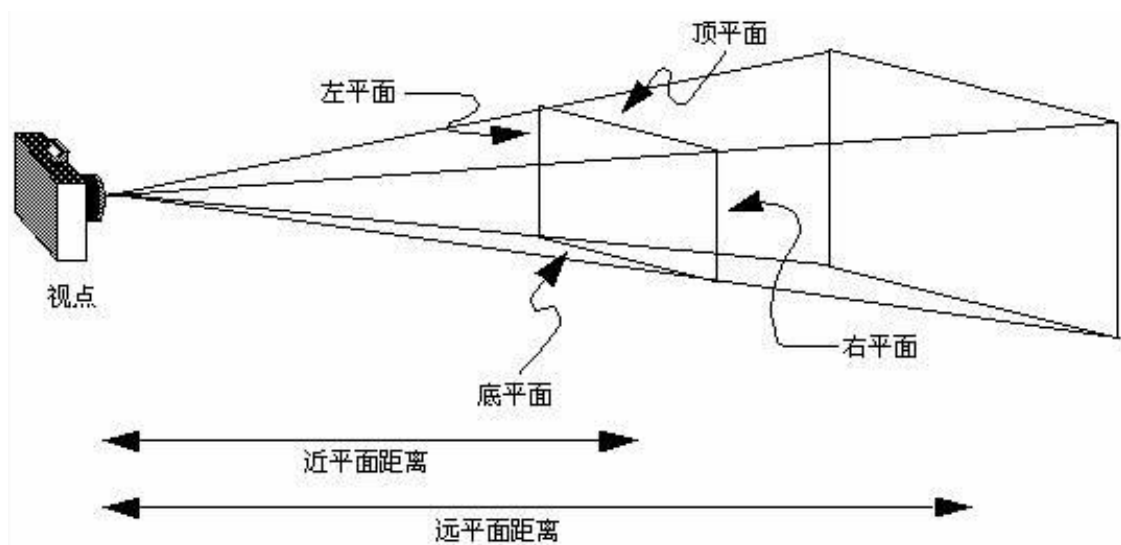


图 5-2 透视投影视景体

- OpenGL 透视投影函数二:

```
void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);
```

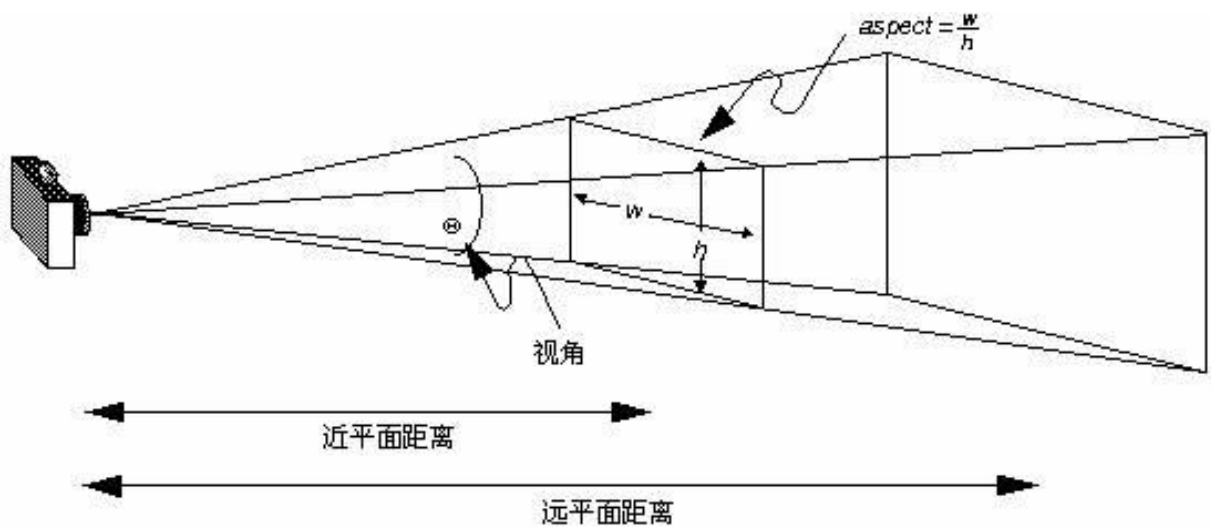


图 5-3 透视投影视景体

fovy: 视角参数, 视角越大观测到景物越多, 被观测物体尺寸越小;

aspect: 视窗宽高比;

near: 近裁剪面距离 (透视投影该参数必须 >0);

far: 远裁剪面距离。

该函数的作用与函数一相似,定义了一个平截头体(如图 5-3 所示)。与正投影相似,位于视景体内的模型才能显示。

在 OpenGL 中,投影是通过变换来实现的,从指定顶点到顶点出现在屏幕上,会发生三种类型的变换:视图变换、模型变换和投影变换。视图变换是应用到场景中的第一种变换,它确定了观察点的位置;模型变换用于操作模型和其中特定的对象,该变换将对象移动到需要的位置,然后再对它们进行旋转和缩放,需要特别注意的是模型变换的不同顺序会得到不同的效果;投影变换将在模型视图变换之后应用到顶点上,该变换实际上定义了视景体并创建了裁剪平面。最后,在对象被映射到屏幕上的窗口中时,还要做一次视口变换,完成到物理窗口坐标的映射。

5.3 实验主要实现步骤

(1) 打开工程

启动 VC6 或 VS2010 开发环境,对于 VC6 选择菜单:【File->Open Workspace】,在弹出的 OpenWorkspace 对话框中选择“实验 5”目录中的 basic3d.dsw 文件,然后按下【打开】按钮,打开实验项目。对于 VS2010 选择菜单:【文件->打开->项目/解决方案】,在弹出的打开项目对话框中选择“实验 5”目录中的 basic3d.sln 文件,然后按下打开项目,打开实验项目。

(2) 调用 glOrtho 函数实现平行投影

打开 main.cpp 文件,进入 onReshape 函数,在 glLoadIdentity() 和 glMatrixMode()函数调用之间,加入如下代码:

```
121 void onReshape(int w, int h)
122 {
123     glViewport(0, 0, w, h);
124     glMatrixMode(GL_PROJECTION);
125     glLoadIdentity();
126
127     h = h > 0 ? h : 1;
128     double aspect = (float)w / h;
129     if(aspect > 1)
```



```

130 {
131     glOrtho(-1*aspect, 1*aspect, -1, 1, 0, 3);
132 }
133 else
134 {
135     glOrtho(-1, 1, -1/aspect, 1/aspect, 0, 3);
136 }
137
138 glMatrixMode(GL_MODELVIEW);
139 glutPostRedisplay();
140 }

```

完成上述代码后，构建并运行项目，可以得到图 5-4 所示的平行投影效果。

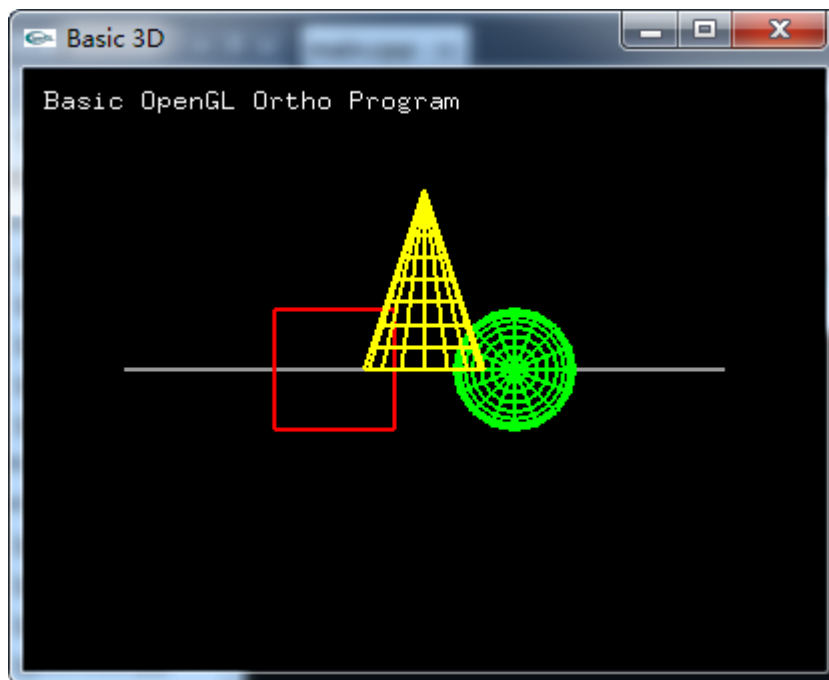


图 5-4 调用 glOrtho 平行投影效果

(3) 调用 glFrustum 函数实现透视投影

打开 main.cpp 文件，进入 onReshape 函数，修改 glLoadIdentity()和 glMatrixMode()函数调用之间的原有代码为：

```

1 void onReshape(int w, int h)
2 {
3     glViewport(0, 0, w, h);
4     glMatrixMode(GL_PROJECTION);
5     glLoadIdentity();
6
7     h = h > 0 ? h : 1;
8     double aspect = (float)w / h;
9     if(aspect > 1)
10    {
11        glFrustum(-1*aspect, 1*aspect, -1, 1, 1, 3);
12    }
13    else
14    {
15        glFrustum(-1, 1, -1/aspect, 1/aspect, 1, 3);

```

```

16     }
17
18     glMatrixMode(GL_MODELVIEW);
19     glutPostRedisplay();
20 }

```

完成上述代码后，构建并运行项目，可以得到图 5-5 所示的透视投影效果。

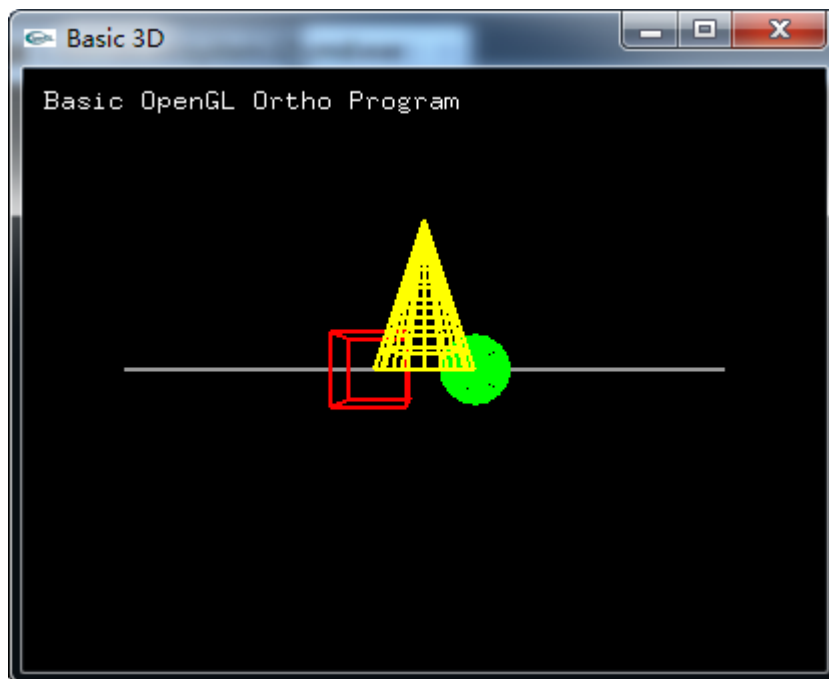


图 5-5 调用 glFrustum 透视投影效果

(4) 调用 gluPerspective 函数实现透视投影

打开 main.cpp 文件，进入 onReshape 函数，修改 glLoadIdentity()和 glMatrixMode()函数调用之间的原有代码为：

```

1 void onReshape(int w, int h)
2 {
3     glViewport(0, 0, w, h);
4     glMatrixMode(GL_PROJECTION);
5     glLoadIdentity();
6
7     h = h > 0 ? h : 1;
8     double aspect = (float)w / h;
9     gluPerspective(90, aspect, 1, 3);
10
11     glMatrixMode(GL_MODELVIEW);
12     glutPostRedisplay();
13 }

```

完成上述代码后，构建并运行项目，可以得到类似图 5-5 所示的透视投影效果。

(5) 三视图投影

OpenGL 程序除了使用系统提供的投影函数生成投影矩阵外，也可以根据需要自行设置投影矩阵，实现不同的投影效果。为了方便验证教材上的多种投影矩

阵，需要在程序中引入 `gvector3d.h/cpp` 三维向量和 `gmatrix3d.h/cpp` 三维齐次矩阵文件。首先定位到 `main.cpp` 文件首部，加入以下代码：

```
1  #include "gvector3d.h"
2  #include "gmatrix3d.h"
```

`GVector3d` 和 `GMatrix3d` 封装了向量和矩阵的基本功能，主要实现功能如表 5-1 和 5-2 所示。

表 5-1 `GVector3D` 类函数列表

函数名	函数功能
<code>GVector3d()</code>	默认构造函数，初始化 x、y、z 分量均为 0
<code>GVector3d(double x, double y, double z)</code>	通过指定 x, y, z 坐标初始化 <code>GVector3d</code> 对象
<code>GVector3d(const double *v)</code>	通过长度为 3 的数组初始化 <code>GVector3d</code> 对象
<code>double x()</code>	返回 x 分量
<code>double y()</code>	返回 y 分量
<code>double z()</code>	返回 z 分量
<code>void setX(double x)</code>	设置 x 分量
<code>void setY(double y)</code>	设置 y 分量
<code>void setZ(double z)</code>	设置 z 分量
<code>void set(double x, double y, double z)</code>	设置 x, y, z 分量
<code>operator double * ()</code>	将 <code>GVector3d</code> 对象转换为 <code>double *</code> 指针
<code>operator +(const GVector3d &v)</code>	向量求和并返回和向量
<code>operator +=(const GVector3d &v)</code>	向量求和，并存回到调用向量中
<code>operator -(const GVector3d &v)</code>	向量求差并返回差向量
<code>operator -=(const GVector3d &v)</code>	向量求差，并存回到调用向量中
<code>operator -()</code>	将调用向量取负
<code>operator * (double d)</code>	向量数乘，并返回结果
<code>operator *=(double d)</code>	向量数乘，并存回到调用向量中
<code>operator / (double d)</code>	向量数除，并返回结果
<code>operator /=(double d)</code>	向量数除，并存回到调用向量中
<code>GVector3d crossMult(const GVector3d &v)</code>	向量叉乘
<code>double dotMult(const GVector3d &v)</code>	向量点乘
<code>double getLength()</code>	计算并返回向量模长
<code>GVector3d getNormal()</code>	计算单位向量并返回
<code>void normalize()</code>	将调用向量单位化
<code>GVector3d transformTo(const GMatrix3d &m)</code>	执行向量的矩阵齐次变换，并返回结果向量

表 5-2 `Matrix3D` 类函数列表

函数名	函数功能
GMatrix3d()	默认构造函数，初始化为单位矩阵
GMatrix3d(const double *m)	通过长度为 16 的数组构造 GMatrix3d 对象
GMatrix3d(const GMatrix3d &m)	拷贝构造函数
operator double *()	将 GMatrix3d 对象转换为 double *指针
operator = (const GMatrix3d &m)	赋值运算符重载，实现矩阵赋值
operator * (const GMatrix3d &m)	矩阵右乘，返回乘积矩阵
operator *= (const GMatrix3d &m)	矩阵右乘，并存回到调用矩阵中
GMatrix3d getTranspose()	矩阵转置，返回转置矩阵
void transpose()	矩阵转置，并存回到调用矩阵中
GMatrix3d getInverse()	计算并返回逆矩阵
void inverse()	计算逆矩阵，并存回到调用矩阵中
static GMatrix3d createZero()	创建并返回单位矩阵
static GMatrix3d createIdentity()	创建并返回单位矩阵
static GMatrix3d createRotateX(double af)	创建并返回绕 X 轴旋转 af 角度的矩阵
static GMatrix3d createRotateY(double af)	创建并返回绕 Y 轴旋转 af 角度的矩阵
static GMatrix3d createRotateZ(double af)	创建并返回绕 Z 轴旋转 af 角度的矩阵
static GMatrix3d createRotate(double af, const GVector3d &v)	创建并返回绕轴 v 旋转 af 角度的矩阵
static GMatrix3d createTranslate(double x, double y, double z)	创建并返回平移矩阵
static GMatrix3d createScale(double sx, double sy, double sz)	创建并返回缩放矩阵
static GMatrix3d createMirror(const GVector3d &v)	创建对称变换矩阵

三视图指正视图、侧视图和俯视图，通过在 onReshape 回调函数中设置特殊的投影矩阵，可以实现上述三种投影图。打开 main.cpp 文件，进入 onReshape 函数，修改 glLoadIdentity()和 glMatrixMode()函数调用之间的原有代码为：

```

1 void onReshape(int w,int h)
2 {
3     glViewport(0, 0, w, h);
4     glMatrixMode(GL_PROJECTION);
5     glLoadIdentity();
6
7     h = h > 0 ? h : 1;
8     double aspect = (float)w / h;
9     GMatrix3d m, ms;
10    m[10] = 0;
11
12    if(aspect > 1)
13    {
14        ms[0] = 1.0/aspect;
15    }
16    else ms[5] = aspect;
17    glMultMatrixd(m*ms);

```

```

18
19  glMatrixMode(GL_MODELVIEW);
20  glutPostRedisplay();
21  }

```

完成上述代码后，构建并运行项目，可以得到如图 5-6(a)所示的正视图效果。为了得到图 5-6(b)所示的侧视图可以修改上述第 10 行给矩阵 m 赋值的代码为：

```

10  m[0] = 0;
11  m[8] = -1;
12  m[10] = 0;

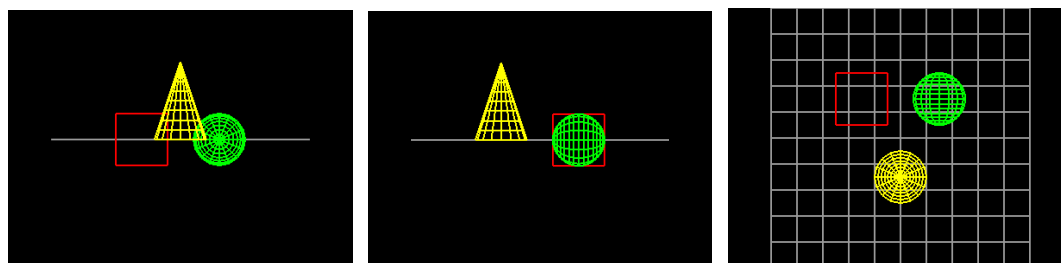
```

为了得到图 5-6(c)所示的俯视图可以修改上述第 10 行给矩阵 m 赋值的代码为：

```

10  m[5] = 0;
11  m[9] = -1;
12  m[10] = 0;

```



(a)正视图

(b)侧视图

(c)俯视图

图 5-6 三视图投影效果

(6) 正轴测投影

正轴测投影是对任意平面作的投影，其投影面与已知坐标轴平面不平行，根据投影面在 X、Y、Z 轴上截距是否相等，正轴测投影分为：等轴测（在 X、Y、Z 三轴上截距全部相等）、正二测（在 X、Z 两轴上截距相等）和正三测（在 X、Y、Z 三轴上截距均不相等）。打开 main.cpp 文件，进入 onReshape 函数，修改上述第 10 行开始为矩阵 m 赋值的代码为：

```

10  m[0] = sqrt(2.0)/2;
11  m[1] = -sqrt(6.0)/6;
12  m[5] = sqrt(6.0)/3;
13  m[8] = -sqrt(2.0)/2;
14  m[9] = -sqrt(6.0)/6

```

完成上述代码后，构建并运行项目，可以得到如图 5-7 所示的等轴测投影，如果希望得到正二测或正三测投影效果可以按照教材中提供的矩阵，修改上述第 10 行开始的代码。

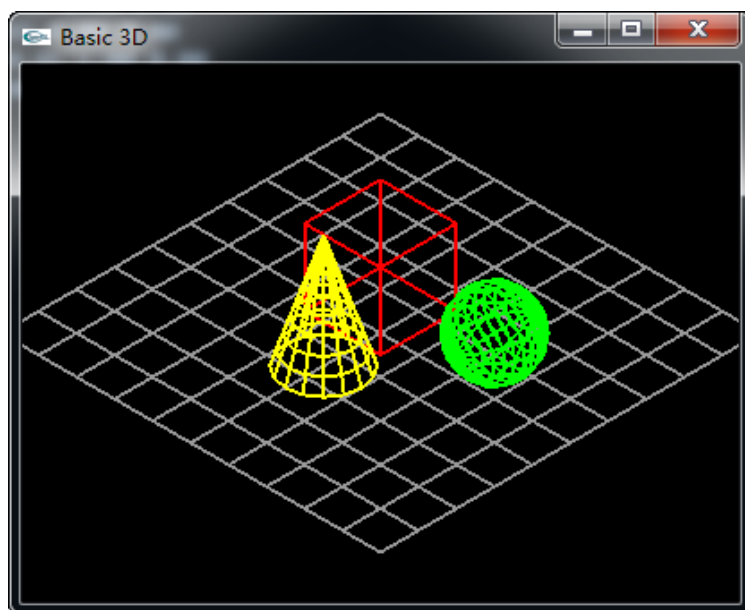


图 5-7 等轴测投影效果

(7) 斜轴测投影

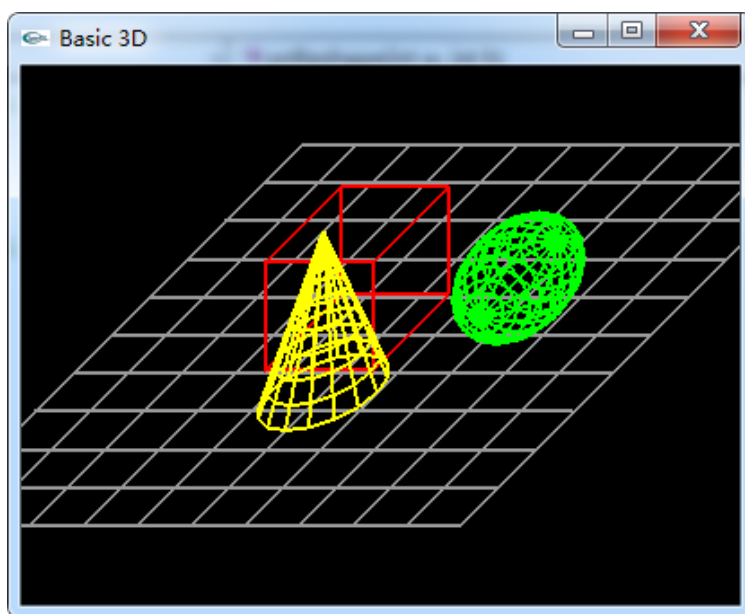


图 5-8 斜等测投影效果

与正轴测投影不同，斜轴测投影的投影线与投影面不垂直。斜轴测图将三视图与正轴测图的特性结合起来，既能像三视图一样进行精确的距离和角度测量，又能像正轴测图那样同时体现三维形体的多个面，具有立体效果。常用的斜轴测图有斜等测图和斜二测图两种。打开 main.cpp 文件，进入 onReshape 函数，修改上述第 10 行开始为矩阵 m 赋值的代码为：

```
10 m[8] = -sqrt(2.0)/2;  
11 m[9] = -sqrt(2.0)/2;
```

完成上述代码后，构建并运行项目，可以得到如图 5-8 所示的斜等测投影，如果希望得到斜二测投影效果可以按照教材中提供的矩阵，修改上述第 10 行开始的代码。

(8) 透视投影

透视投影的投影中心到投影面的距离是有限的，透视投影具有透视缩小效应，会产生远小近大的效果，透视投影不能真是反映物体的精确尺寸和形状，但具深度感，常用于真实感图形生成。在透视投影中不平行投影面的平行线投影后会聚集于一个点，这个点成为灭点。灭点可以有多个，根据灭点的数目可以分为一点透视、两点透视和三点透视。打开 main.cpp 文件，进入 onReshape 函数，修改上述第 10 行开始为矩阵 m 赋值的代码为：

```
10  m[3] = -0.5;  
11  m[7] = 0.5;  
12  m[11] = -0.5;
```

完成上述代码后，构建并运行项目，可以得到如图 5-9 所示的三点透视投影，如果希望得到一点、两点透视投影效果可以按照教材中提供的矩阵，修改上述第 10 行开始的代码。

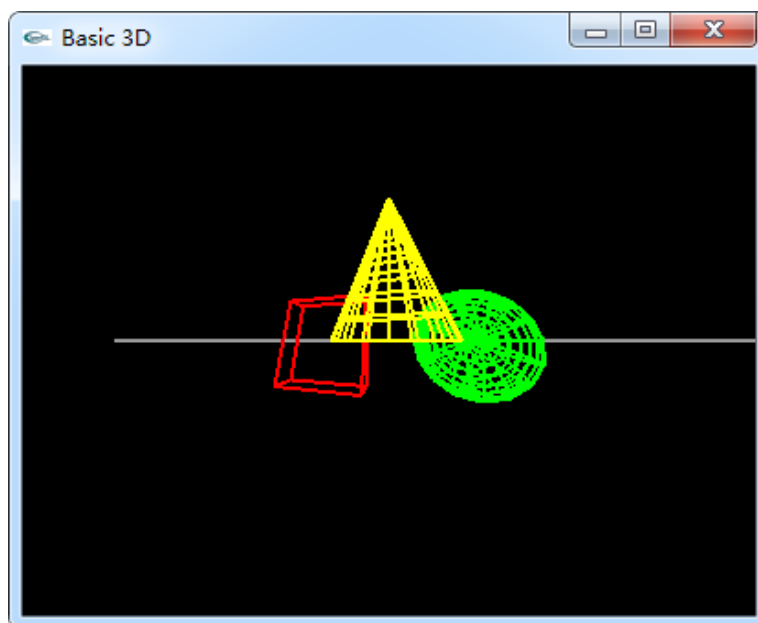


图 5-9 三点透视投影效果

实验6 鼠标追踪球旋转

6.1 实验目的

- (1) 掌握基本的三维几何变换矩阵和组合变换方法；
- (2) 掌握 OpenGL 模型变换函数，实现模型几何变换；
- (3) 理解鼠标追踪球的算法原理，实现算法程序

6.2 实验原理

在三维建模、交互式造型等环境中，用户需要利用鼠标拖动形体在屏幕上做三维任意自由度的旋转，使用户可以观察到模型不同侧面侧面的几何特征。这种利用二维操作设备鼠标来模拟三维几何形体操作的技术就是鼠标追踪球技术，该技术的核心是如何通过变换将鼠标的二维运动转化为屏幕上所选定的形体在图形空间的三维运动。为此需要把二维屏幕坐标投影到三维的半球面上，将用户鼠标在屏幕上的运动，转换为三维球面上的运动。

设屏幕宽度为 w ，高度为 h ，三维场景以屏幕中心 $(w/2, h/2)$ 转动，鼠标移动的起点坐标为 $P_1(x_1, y_1)$ ，终点坐标为 $P_2(x_2, y_2)$ 。

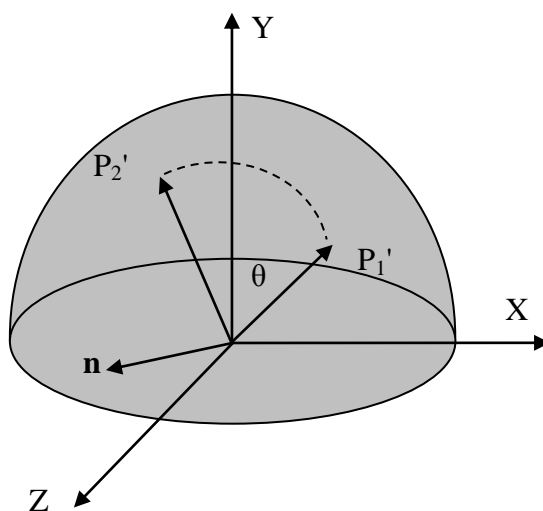


图 6-1 鼠标追踪球示意图

- (1) 将鼠标的屏幕二维坐标 (x, y) 映射到单位球面上，变为三维坐标 P_1' 和 P_2' ，映射公式为：

$$\begin{aligned}
 x' &= (2x - w) / w \\
 y' &= (h - 2y) / h \\
 z' &= \sqrt{1 - x'^2 - y'^2}
 \end{aligned}
 \quad (6-1)$$

(2) 当鼠标从 P_1' 运动到 P_2' 时产生的旋转角度为 θ ，即图 6-1 中向量 $\overrightarrow{OP_1'}$ 与向量 $\overrightarrow{OP_2'}$ 的夹角，这可以先将 $\overrightarrow{OP_1'}$ 和 $\overrightarrow{OP_2'}$ 单位化后通过向量点积求出 $\cos\theta$ ，再用反余弦函数求出 θ 。而旋转轴向量 \mathbf{n} ，为 $\overrightarrow{OP_1'}$ 与向量 $\overrightarrow{OP_2'}$ 的叉积向量。因此可以得到如下旋转参数计算公式：

$$\begin{aligned}
 \theta &= \arccos(\overrightarrow{OP_1'} \cdot \overrightarrow{OP_2'}) \\
 \mathbf{n} &= \overrightarrow{OP_1'} \times \overrightarrow{OP_2'}
 \end{aligned}
 \quad (6-2)$$

(3) 当获得旋转轴和旋转角度后，可以使用三维组合变换，求出绕旋转轴 \mathbf{n} 旋转 θ 的组合变换矩阵。如图 6-2(a) 所示旋转轴向量 \mathbf{n} 所在体对角面与 YOZ 平面所成夹角为 α ， \mathbf{n} 与 ZOX 平面所成夹角为 β 。首先绕 Y 轴旋转 $-\alpha$ 角度，如图 6-2(b) 所示此时旋转轴 \mathbf{n} 位于 YOZ 平面；接下来再绕 X 轴旋转 β 角度，如图 6-2(c) 所示此时旋转轴 \mathbf{n} 重合于坐标轴 Z；当 \mathbf{n} 重合于 Z 轴后，执行绕 Z 轴旋转 θ 角度的变换；最后再绕 X 轴旋转 $-\beta$ 角度，绕 Y 轴旋转 α 角度，回到原始坐标系。组合旋转公式如式 (6-3) 所示。

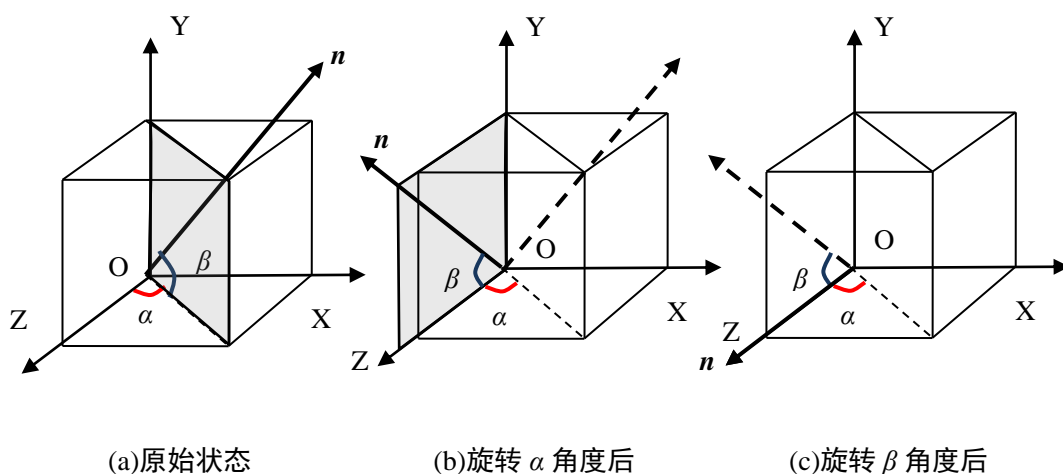


图 6-2 绕任意轴组合旋转示意图

$$M = \begin{bmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\beta & \sin\beta & 0 \\ 0 & -\sin\beta & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (6-3)$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\beta & -\sin\beta & 0 \\ 0 & \sin\beta & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

6.3 实验主要实现步骤

(1) 打开工程

启动 VC6 或 VS2010 开发环境, 对于 VC6 选择菜单: **【File->Open Workspace】**, 在弹出的 OpenWorkspace 对话框中选择“实验 6”目录中的 trackball.dsw 文件, 然后按下 **【打开】** 按钮, 打开实验项目。对于 VS2010 选择菜单: **【文件->打开->项目/解决方案】**, 在弹出的打开项目对话框中选择“实验 6”目录中的 trackball.sln 文件, 然后按下打开项目, 打开实验项目。

(2) 增加全局变量

打开 main.cpp 文件, 翻到文件首部#include...下方, 加入如下全局变量:

```
13 // 鼠标位置
14 int gMouseX = -1;
15 int gMouseY = -1;
16 // 是否进行鼠标追踪球
17 bool gIsStartTrackBall = false;
18 // 旋转矩阵
19 GMatrix3d gRotMatrix;
```

(3) 增加球面向量映射函数

在全局变量定义的下方加入并实现 mousePtToSphereVec() 函数, 该函数使用公式 (6-1) 将鼠标坐标映射为曲面向量:

```
20 GVector3d mousePtToSphereVec(int x, int y, int w, int h)
21 {
22     GVector3d vec;
23     vec.setX((2.0*x - w) / w);
24     vec.setY((h - 2.0*y) / h);
25     double r = (vec.x()*vec.x()+vec.y()*vec.y());
26     if(r > 1) r = 1;
27     vec.setZ(sqrt(1 - r));
28     vec.normalize();
29     return vec;
30 }
```

(4) 修改绘图函数实现场景旋转

在 main.cpp 文件中找到 onDisplay()函数,并且在平移变换后增加旋转调用:

```
31 void onDisplay()
32 {
33     ...
34     glLoadIdentity();
35     glTranslatef(0, 0, -3.5f);
36     glMultMatrixd(gRotMatrix);
37     ...
38 }
```

(5) 修改鼠标事件函数

在 main.cpp 文件中找到 onMouse()函数,并且增加以下代码:

```
39 void onMouse(int button, int state, int x, int y)
40 {
41     if(button == GLUT_LEFT_BUTTON)
42     {
43         if(state == GLUT_DOWN)
44         {
45             gMouseX = x;
46             gMouseY = y;
47             gIsStartTrackBall = true;
48         }
49         else if(state == GLUT_UP)
50         {
51             gIsStartTrackBall = false;
52         }
53         glutPostRedisplay();
54     }
55 }
```

上述代码判断鼠标左键是否按下,如果左键按下则记录鼠标当前的位置,并且设置 gIsStartTrackBall 变量为 true,开始鼠标追踪球旋转;如果左键释放,则将 gIsStartTrackBall 变量置为 false,停止鼠标追踪球旋转。

(6) 修改鼠标移动事件函数

在 main.cpp 文件中找到 onMouseMove()函数,并且增加以下代码:

```
56 void onMouseMove(int x, int y)
57 {
58     if(gIsStartTrackBall)
59     {
60         if(x != gMouseX || y != gMouseY)
61         {
62             int w, h;
63             double rotAngle;
64             GVector3d lastVec, currentVec, axis;
65             GMatrix3d m;
66             w = glutGet(GLUT_WINDOW_WIDTH);
67             h = glutGet(GLUT_WINDOW_HEIGHT);
68             lastVec = mousePtToSphereVec(gMouseX, gMouseY, w, h);
69             currentVec = mousePtToSphereVec(x, y, w, h);
70             gMouseX = x;          gMouseY = y;
```

```

71         rotAngle = acos(lastVec.dotMult(currentVec)) *
72             RADIAN_TO_ANGLE;
73         axis = lastVec.crossMult(currentVec);
74         if (axis.getLength() <= 1e-6) return ;
75         axis.normalize();
76         m = GMatrix3d::createRotate(rotAngle, axis);
77         gRotMatrix *= m;
78         glutPostRedisplay();
79     }
80 }
81 }

```

上述代码调用 mousePtToSphereVec() 函数计算前后两次鼠标位置对应的单位球面向量，根据公式（6-2）计算旋转轴和旋转角，通过 GMatrix3d 类的 createRotate 函数获取绕任意轴向的旋转矩阵，将该旋转矩阵右乘到全局旋转后，调用重回函数实现三维场景的旋转。

（7）实现绕任意轴向旋转函数

打开 gmatrix3d.cpp 文件，找到 createRotate() 函数，并且加入以下代码：

```

82 GMatrix3d GMatrix3d::createRotate(double theta, const GVector3d &v)
83 {
84     GMatrix3d m;
85     double len = v.getLength();
86
87     if(IS_ZERO(theta) || IS_ZERO(len)) return m;
88
89     double lxy = v.getLengthXY();
90     if(IS_ZERO(lxy))
91     {
92         if(v.z() < 0) theta = -theta;
93         return createRotateZ(theta);
94     }
95     double lyz = v.getLengthYZ();
96     if(IS_ZERO(lyz))
97     {
98         if(v.x() < 0) theta = -theta;
99         return createRotateX(theta);
100     }
101     double lxz = v.getLengthZX();
102     if(IS_ZERO(lxz))
103     {
104         if(v.z() < 0) theta = -theta;
105         return createRotateY(theta);
106     }
107     //(1)绕 Y 轴旋转到 YOZ 平面
108     m = createRotateY(-v.x()/lxz, v.z()/lxz);
109     //(2)绕 X 轴旋转到 ZOX 平面
110     m *= createRotateX(v.y()/len, lxz/len);
111     //(3)绕 Z 轴旋转 theta 角度
112     m *= createRotateZ(theta);
113     //(4)绕 X 轴反旋转
114     m *= createRotateX(-v.y()/len, lxz/len);
115     //(5)绕 Y 轴反旋转
116     m *= createRotateY(v.x()/lxz, v.z()/lxz);

```

```

117
118   return m;
119 }

```

上述代码对旋转轴的几种特殊情况进行了判断，并且根据公式（6-3）实现了组合旋转变换，并返回乘积后的组合变换矩阵，实现了绕任意向量旋转指定角度的功能。完成上述代码后，构建并运行项目，在窗口中按下鼠标左键并拖拽即可旋转三维场景如图 6-3 所示。

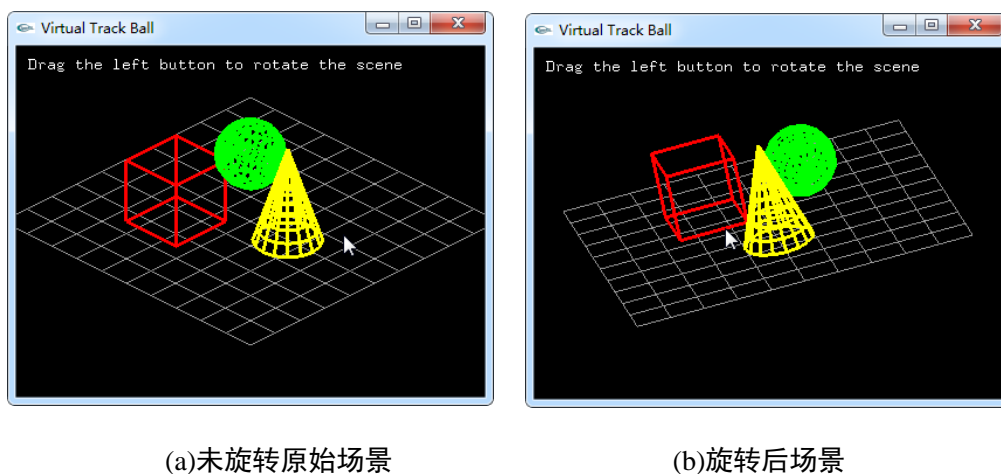


图 6-3 鼠标追踪球旋转效果图

(8) 利用 OpenGL 函数再次修改鼠标移动事件函数

前面的步骤（6）通过调用 GMatrix3d 类的 createRotate()函数获取绕任意轴旋转的变换矩阵，其实 OpenGL 的 glRotated()函数也可以产生这样的旋转矩阵，而且效率更高。利用 OpenGL 获取旋转矩阵代码如下：

```

120 void onMouseMove(int x, int y)
121 {
122     if(gIsStartTrackBall)
123     {
124         if(x != gMouseX || y != gMouseY)
125         {
126             int w, h;
127             double rotAngle;
128             GVector3d lastVec, currentVec, axis;
129             GMatrix3d m;
130             w = glutGet(GLUT_WINDOW_WIDTH);
131             h = glutGet(GLUT_WINDOW_HEIGHT);
132             lastVec = mousePtToSphereVec(gMouseX, gMouseY, w, h);
133             currentVec = mousePtToSphereVec(x, y, w, h);
134             gMouseX = x;          gMouseY = y;
135             rotAngle = acos(lastVec.dotMult(currentVec)) *
136                         RADIAN_TO_ANGLE;
137             axis = lastVec.crossMult(currentVec);
138             if (axis.getLength() <= 1e-6) return ;
139             axis.normalize();
140             glMatrixMode(GL_MODELVIEW);

```

```
141      glPushMatrix();
142      glLoadIdentity();
143      glRotated(rotAngle, axis.x(), axis.y(), axis.z());
144      glGetDoublev(GL_MODELVIEW_MATRIX, m);
145      glPopMatrix();
146      gRotMatrix *= m;
147      glutPostRedisplay();
148   }
149 }
150 }
```

上述代码首先将矩阵模式设为 **ModelView**，并将当前模型变换矩阵压栈，然后将其置为单位矩阵，通过调用 **glRotated()**函数产生旋转矩阵并左乘单位矩阵（其结果仍然是旋转矩阵），最后调用 **glGetDoublev()**函数获取该矩阵。完成上述代码后，构建并运行项目，按下鼠标进行拖拽，可以发现旋转效果与之前的效果一致。

实验7 光照与纹理

7.1 实验目的

- (1) 理解 OpenGL 真实感图形生成的关键技术;
- (2) 掌握 OpenGL 消隐、光照、材质和纹理编程技术;
- (3) 理解三维球体生成技术, 编写带纹理的球体绘制程序。

7.2 实验原理

在 OpenGL 中, 为了使绘制的物体看上去更加真实, 就会对它进行消隐、光照处理或纹理贴图。计算机内部表示的三维场景要经过投影变换才能最终显示在二维屏幕上, 然而显示在屏幕上的二维图形丢失了深度信息, 如果不预先加以处理, 产生的图形往往会产生二义性或不正确的深度关系。因此产生真实感图形的首要问题就是在给定视点和观察方向后, 决定场景中哪些物体的表面或部分表面是可见的, 哪些是被遮挡不可见的, 这一处理过程就称为消隐。OpenGL 提供了深度缓存消隐, 通过调用相关函数可以对场景中的图元进行消隐处理, 显示正确的深度遮挡关系。在 OpenGL 中调用深度缓存消隐的方法如下:

- (1) 在初始化 OpenGL 时启用深度测试功能, 调用函数为:

- `glEnable(GL_DEPTH_TEST);`

- (2) 在图元绘制之前先清除深度缓存

- `glClear(GL_DEPTH_BUFFER_BIT);`

- (3) 可以根据需要设置 OpenGL 深度测试的比较方法

- `glDepthFunc(GLenum func);`

函数的枚举参数 `func` 可以是以下几种之一:

- `GL_NEVER`: 不替换缓存;
- `GL_ALWAYS`: 一直替换缓存;
- `GL_LESS`: 深度值小于缓存值时替换;
- `GL_LEQUAL`: 深度值小于等于缓存值时替换;
- `GL_EQUAL`: 深度值等于缓存值时替换;

- `GL_EQUAL`: 深度值大于等于缓存值时替换;
- `GL_GREATER`: 深度值大于缓存值时替换;
- `GL_NOTEQUAL`: 深度值不等于缓存值时替换。

系统使用 `GL_LESS` 作为默认值。

真实感图形绘制能够模拟光照,正确表现物体表面颜色的明暗变化。`OpenGL` 是一种基于局部光照模型的实时光栅图形系统,物体表面的颜色受到光源和材质两方面因素的影响,简单光照模型下主要考虑:环境光、漫反射光和镜面反射光,3 种类型的光照和材质系数。

(1) 环境光

环境光是光在物体和周围环境之间多次反射的结果,即是来自周围的环境对光的反射。环境光的特点是:它不是直接自光源,而是照射在物体上的光来自周围各个方向,且又均匀地向各个方向反射。在 `OpenGL` 中设置光源和材质环境光系数的函数如下:

- `glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);`
- `glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);`

上述函数中 `GL_LIGHT0` 为光源标号, `OpenGL` 支持最多 9 个光源可以使用 `GL_LIGHT0—GL_LIGHT8` 中的某个光源标号;参数 `GL_AMBIENT` 指示设置环境光系数; `ambient` 为长度为 3 的 `float` 型的环境光系数数组指针,分别用于设置 R、G、B 三分量; `GL_FRONT_AND_BACK` 指定设置材质的外表面和内表面,除此之外可以指定: `GL_FRONT` 外表面, `GL_BACK` 内表面。

(2) 漫反射光

漫反射光是在光的照射下,物体表面均匀向各个方向反射的光,它决定了物体的基本颜色。漫反射模型可以用 Lambert 余弦定理描述,对于一个漫反射体,表面的反射光强度与光源入射角的余弦成正比,因而漫反射光仅与光源和物体位置有关,与观察者位置无关。在 `OpenGL` 中设置光源和材质环境光系数的函数如下:

- `glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);`
- `glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, diffuse);`

上述函数的调用参数与环境光调用函数类似。参数 `GL_DIFFUSE` 指示设置漫反射光系数；`diffuse` 为长度为 3 的 `float` 型的漫反射材质系数数组指针，分别用于设置 R、G、B 三分量。

(3) 镜面反射光

镜面反射是光源照射在比较光滑的材质表面（如镜子、光亮的金属等）所形成的集中反射。它的特点是反射方向的镜面光很强，但是偏离反射方向镜面反射光强迅速衰减，镜面反射光与光源、物体和观察者三者的位置都有密切的关系。在 OpenGL 中设置光源和材质镜面反射光系数的函数如下：

- `glLightfv(GL_LIGHT0, GL_SPECULAR, specular);`
- `glLightf(GL_LIGHT0, GL_SHININESS, shininess);`
- `glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, specular);`
- `glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, shininess);`

上述函数的调用参数与环境光调用函数类似。参数 `GL_SPECULAR` 指示设置镜面反射光系数；`specular` 为长度为 3 的 `float` 型的镜面反射材质系数数组指针，分别用于设置 R、G、B 三分量；参数 `GL_SHININESS` 指示设置镜面反射衰减系数；`shininess` 为镜面衰减系数，`shininess` 越大偏离反射方向后镜面反射光强衰减越迅速，表示物体表面越光洁。

使用光照与材质生成的真实感图形往往由于表面过于平滑和单调，看起来真实性不强。这是因为显示世界中的物体，表面往往有各种纹理，即表面细节。例如木材表面有木纹，建筑物墙面有装饰图案，机器外壳表面有文字说明等，它们是通过颜色色彩或明暗变化来体现出表面的细节，这种纹理称为颜色纹理。OpenGL 支持一维、二维和三维的颜色纹理映射，其中最常用的是二维纹理，纹理贴图的基本调用方式如下：

(1) 生成一个可用的纹理号

- `glGenTextures(1, &bindName);`

该函数调用产生一个有效纹理号，存入 `bindName` 变量中。其中 `bindName` 变量为 `GLuint` 类型。

(2) 载入纹理图像

- `glBindTexture(GL_TEXTURE_2D, bindName);`
- `glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, pixels);`

上述代码首先选定 `bindName` 句柄指向的纹理，然后将一幅 `width*height` 大小的以 `RGBA` 存储在 `pixels` 为首地址中的内存图像上传至显卡形成二维纹理。

(3) 设置纹理参数

- `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);`
- `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);`
- `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);`
- `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);`

上述函数调用设置纹理的缩放方法为线性插值，纹理坐标折绕采用重复方式，即纹理坐标大于 1 后取余。

(4) 开启纹理贴图功能

- `glEnable(GL_TEXTURE_2D);`

该函数调用开启二维纹理贴图功能，默认情况下该功能关闭。

(5) 绘制图元设置纹理坐标

- `glTexCoord2d(s, t);`

上传纹理图像，设置合适的纹理参数并开启纹理贴图功能后，可以调用 `glBindTexture()` 函数选定纹理，在 OpenGL 图元绘制时调用纹理坐标设置函数，为每个几何顶点设置对应的纹理坐标，即进行纹理贴图。

7.3 实验主要实现步骤

(1) 打开工程

启动 VC6 或 VS2010 开发环境，对于 VC6 选择菜单：**【File->Open Workspace】**，在弹出的 `OpenWorkspace` 对话框中选择“实验 7”目录中的 `earth.dsw` 文件，然后按下**【打开】**按钮，打开实验项目。对于 VS2010 选择菜单：**【文件->打开->项目/解决方案】**，在弹出的打开项目对话框中选择“实验 7”目录中的 `earth.sln` 文件，然后按下打开项目，打开实验项目。

(2) 开启深度测试功能

打开 main.cpp 文件, 进入 main() 函数, 修改 glutInitDisplayMode() 函数调用。

```
151 int main(int argc, char *argv[])
152 {
153     glutInit(&argc, argv);
154     glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
155     ...
156 }
```

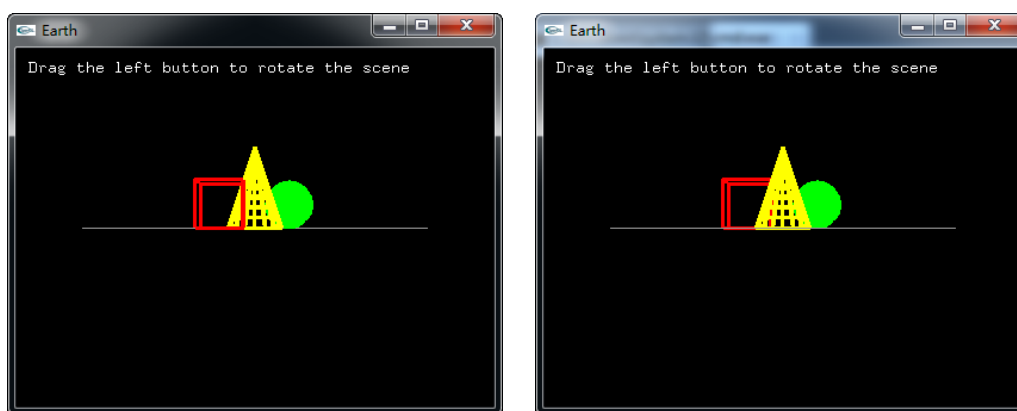
进入 glInit() 函数, 开启深度测试功能。

```
140 void glInit()
141 {
142     glEnable(GL_DEPTH_TEST);
143 }
```

进入 onDisplay() 函数, 在场景绘制前清空深度缓存。

```
60 void onDisplay()
61 {
62     glClearColor(0, 0, 0, 0);
63     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
64     ...
65 }
```

完成上述修改后, 编译并运行程序。如图 7-1 所示, 由于程序增加了深度消隐功能, 原本在图(a)中不正确的深度遮挡关系在图(b)中已经被修正了。



(a)未开启深度测试

(b)开启深度测试

图 7-1 开启深度测试后效果对比图

(3) 增加光照和材质

为了方便在实验程序中实现光照和材质效果, 先在文件首部引入光照和材质封装类 GLight、GMaterial。打开 main.cpp 文件, 翻到文件首部加入:

```
5 #include "glight.h"
6 #include "gmaterial.h"
```

在程序首部全局变量定义处增加一个光源变量。

```
10 GLight gLight;
```

进入 `glInit()` 函数，初始化光源，设定光源参数并开启光源。

```
140 void glInit()
141 {
142     glEnable(GL_DEPTH_TEST);
143
144     gLight.init(0, gDirLight);
145     gLight.setPosition(40, 4, 20);
146     gLight.turnOn();
147 }
```

进入 `onDisplay()` 函数，开启光照效果，并为图元设置材质参数。

```
60 void onDisplay()
61 {
62     ...
63     glDisable(GL_LIGHTING);
64     glLineWidth(1);
65     drawPlane();
66
67     GMaterial m;
68     glEnable(GL_LIGHTING);
69
70     glPushMatrix();
71     glTranslatef(0.3, 0.2, -0.3);
72     m.setDiffuseColor(0, 1, 0);
73     m.apply();
74     glutSolidSphere(0.2f, 16, 16);
75     glPopMatrix();
76
77     glPushMatrix();
78     glTranslatef(0, 0, 0.3);
79     glRotatef(-90, 1, 0, 0);
80     m.setDiffuseColor(1, 1, 0);
81     m.apply();
82     glutSolidCone(0.2f, 0.6f, 16, 8);
83     glPopMatrix();
84
85     glPushMatrix();
86     glTranslatef(-0.3, 0.2, -0.3);
87     m.setDiffuseColor(1, 0, 0);
88     m.apply();
89     glutSolidCube(0.4f);
90     glPopMatrix();
91     ...
92 }
```

完成上述修改后，编译并运行程序。如图 7-2 所示，程序不再绘制线框图，而是绘制出由面片构成的圆锥、球体和立方体场景，场景有一定光照效果，用鼠标旋转场景可以发现光照效果随场景旋转而改变，场景渲染具有一定真实感效果。除了设置漫反射光，可以尝试调用 `GMaterial` 类的其它函数设置材质的环境光、镜面反射光、及辐射光系数，观察不同材质的光照效果。

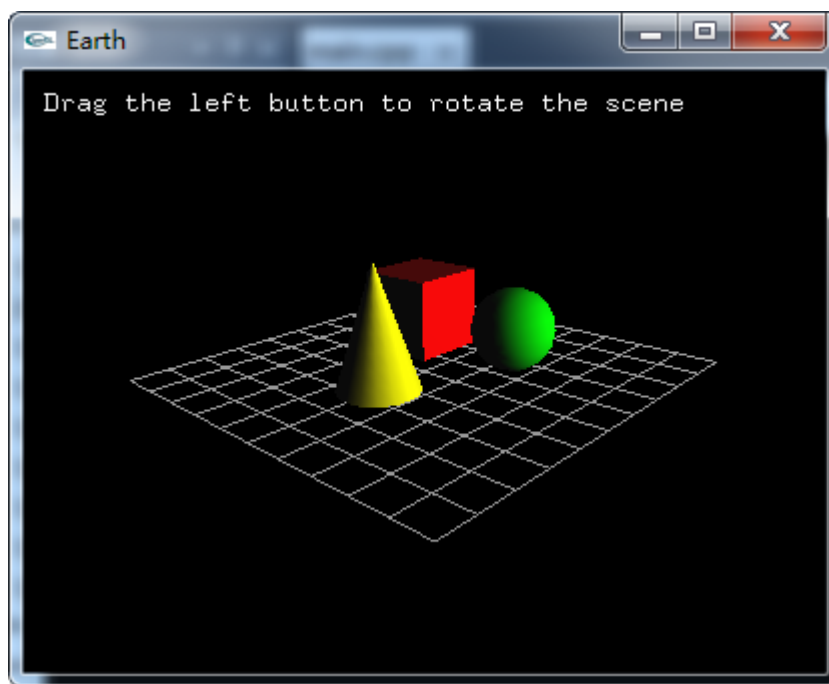


图 7-2 加入光照和材质后的场景绘制效果

(4) 增加纹理贴图

为了方便在实验程序中实现 2D 纹理贴图效果，先在文件首部引入 2D 纹理封装类 GTexture2D。打开 main.cpp 文件，翻到文件首部加入：

```
7 #include "gtexture2d.h"
```

在程序首部全局变量定义处增加一个纹理变量。

```
11 GTexture2D gEarthTex;
```

进入 glInit()函数，初始化纹理对象，载入纹理并设置相应参数。

```
12 void glInit()
13 {
14     glEnable(GL_DEPTH_TEST);
15     glShadeModel(GL_SMOOTH);
16
17     gLight.init(0, gDirLight);
18     gLight.setPosition(40, 4, 20);
19     gLight.turnOn();
20
21     gEarthTex.init();
22     gEarthTex.loadTexImage("earthmap.jpg");
23     gEarthTex.setEnvMode(GL_MODULATE);
24 }
```

进入 onDisplay()函数，删除原有绘制场景，开启纹理贴图效果，使用 OpenGL 图元命令绘制一个矩形，将纹理图形贴到矩形上。

```
60 void onDisplay()
61 {
62     glClearColor(0, 0, 0, 0);
63     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
64
65  glMatrixMode(GL_MODELVIEW);
66  glLoadIdentity();
67  glTranslatef(0, 0, -3.5f);
68  glMultMatrixd(gRotMatrix);
69
70  GMaterial m;
71  m.setDiffuseColor(1, 1, 1);
72  m.setAmbientColor(0.5f, 0.5f, 0.5f);
73  m.apply();
74  glEnable(GL_LIGHTING);
75  glEnable(GL_TEXTURE_2D);
76  gEarthTex.use();
77
78  glBegin(GL_QUADS);
79      glNormal3d(0, 0, 1);
80
81      glTexCoord2f(0, 1);
82      glVertex3d(-1, 1, 0);
83      glTexCoord2f(0, 0);
84      glVertex3d(-1, -1, 0);
85      glTexCoord2f(2, 0);
86      glVertex3d(1, -1, 0);
87      glTexCoord2f(2, 1);
88      glVertex3d(1, 1, 0);
89  glEnd();
90  glDisable(GL_TEXTURE_2D);
91  ...
92 }
```

完成上述修改后，编译并运行程序。如图 7-3 所示，程序显示出一个贴着世界地图的矩形，用鼠标拖拽旋转场景，可以发现该矩形还受到光照影响。



图 7-3 简单纹理贴图效果

(5) 绘制带纹理的球体

OpenGL 的基本图元命令可以绘制点、线、三角形、四边形，如果要绘制复杂的形体则需要将形体的表面离散化成基本图元，如将球体表面离散化成一系列的三角形，当三角形数目较多时就近似逼近了球体表面。下面的代码首先在 Y 方向将球体切分成 nstack 个圆台，然后在 ZOX 平面将圆台切分成 nslice 个四边形小片，最后将每个小片切分成 2 个三角形，并利用 OpenGL 三角带绘制图元命令 GL_TRIANGLE_STRIP 实现球体表面的绘制和纹理贴图。为了实现这一功能，请先打开 ggltools.cpp 文件，进入 gltDrawSphere()函数并加入以下代码：

```

93 void gltDrawSphere(double radius, int nslice, int nstack)
94 {
95     GLint i, j;
96     double drho = PI / nstack;
97     double dtheta = 2.0 * PI / nslice;
98     double ds = 1.0 / nslice;
99     double dt = 1.0 / nstack;
100    double t = 1.0;
101    double s = 0.0;
102    for(i=0; i<nstack; i++)
103    {
104        double rho = i * drho;
105        double srho = sin(rho);
106        double crho = cos(rho);
107        double srho1 = sin(rho + drho);
108        double crho1 = cos(rho + drho);
109        glBegin(GL_TRIANGLE_STRIP);
110        s = 0.0f;
111        for(j=0; j<=nslice; j++)
112        {
113            double theta = j%nslice * dtheta;
114            double stheta = sin(theta);
115            double ctheta = cos(theta);
116            double x, y, z;
117            x = stheta * srho;
118            y = crho;
119            z = ctheta * srho;
120            glTexCoord2d(s, t);
121            glNormal3d(x, y, z);
122            glVertex3d(x * radius, y * radius, z * radius);
123            x = stheta * srho1;
124            y = crho1;
125            z = ctheta * srho1;
126            glTexCoord2d(s, t - dt);
127            glNormal3d(x, y, z);
128            glVertex3d(x * radius, y * radius, z * radius);
129            s += ds;
130        }
131        glEnd();
132        t -= dt;
133    }
134 }

```

打开 main.cpp 文件，进入 onDisplay()，调用已实现的 gltDrawSphere()函数

```
60 void onDisplay()
61 {
62     glClearColor(0, 0, 0, 0);
63     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
64
65     glMatrixMode(GL_MODELVIEW);
66     glLoadIdentity();
67     glTranslatef(0, 0, -3.5f);
68     glMultMatrixd(gRotMatrix);
69
70     GMaterial m;
71     m.setDiffuseColor(1, 1, 1);
72     m.setAmbientColor(0.5f, 0.5f, 0.5f);
73     m.apply();
74     glEnable(GL_LIGHTING);
75     glEnable(GL_TEXTURE_2D);
76     gEarthTex.use();
77     gltDrawSphere(1, 32, 16);
78     glDisable(GL_TEXTURE_2D);
79     ...
80 }
```

完成上述修改后，编译并运行程序。如图 7-4 所示，程序显示出一个贴着世界地图的球体，用鼠标拖拽旋转场景，可以发现该球体还受到光照影响。



图 7-4 受光照影响的地球纹理贴图效果

在 OpenGL 中纹理贴图有很多种模式，可以简单地替换物体表面颜色，也可以综合光照、材质效果，甚至可以进行多重纹理贴图、凹凸纹理贴图和环境纹理贴图。如果要用纹理图像直接替换物体表面，而不考虑光照效果，可以进入

main.cpp 文件中的 glInit()函数，设置纹理的环境模式：

```
140 void glInit()
141 {
142     glEnable(GL_DEPTH_TEST);
143     glShadeModel(GL_SMOOTH);
144
145     gLight.init(0, gDirLight);
146     gLight.setPosition(40, 4, 20);
147     gLight.turnOn();
148
149     gEarthTex.init();
150     gEarthTex.loadTexImage("earthmap.jpg");
151     gEarthTex.setEnvMode(GL_REPLACE);
152 }
```

完成上述修改后，编译并运行程序。如图 7-5 所示，程序显示出的地球不再受到光照影响，与图 7-4 相比显得很明亮。



图 7-5 不受光照影响的地球纹理贴图效果

实验8 三维 B 样条曲面绘制

8.1 实验目的

- (1) 理解三次 B 样条曲面公式。
- (2) 实现 B 样条曲面函数，求取 B 样条曲面点
- (3) 通过 OpenGL 函数调用显示简单的 B 样条曲面模型

8.2 实验原理

B 样条曲面表达式：

$$P(u, w) = \sum_{i=0}^m \sum_{j=0}^n P_{ij} F_{i,m}(u) F_{j,n}(w) \quad (5-1)$$

其中：

$$F_{i,m}(u) = \frac{1}{m!} \sum_{k=0}^{m-i} (-1)^k C_{m+1}^i (u + m - i - k)^m$$

$$\text{其中： } C_{m+1}^i = \frac{(m+1)!}{i!(m+1-i)!}$$

三次 B 样条基函数公式：

$$\begin{aligned} F_{0,3}(u) &= \frac{1}{3!} \sum_{j=0}^3 (-1)^j C_4^j (u + 3 - j)^3 = \frac{1}{6} (-u^3 + 3u^2 - 3u + 1) \\ F_{1,3}(u) &= \frac{1}{6} (3u^3 - 6u^2 + 4) \\ F_{2,3}(u) &= \frac{1}{6} (-3u^3 + 3u^2 + 3u + 1) \\ F_{3,3}(u) &= \frac{1}{6} u^3 \end{aligned} \quad (5-2)$$

式中， P_{ij} 称为控制顶点，所有的 $(m+1)(n+1)$ 个控制顶点组成的空间网格称为控制网格，也称特征网格，对于双三次B样条曲面 $m=n=3$ 。 $F_{i,m}(u)$ 和 $F_{i,n}(w)$ 式定义在 u, w 参数轴上的结点矢量 $U = (u_0, u_1, \dots, u_{2m})$ 和 $W = (w_0, w_1, \dots, w_{2n})$ 的 B 样条基函数。当结点矢量 U, W 沿着 u, w 轴均匀等距分布时，称 $P(u, w)$ 为均

匀 B 样条曲面，否则称为非均匀 B 样条曲面。式 (5-2) 给出了三次均匀 B 样条的基函数。B 样条曲面具有局部支柱性，凸包性，连续性和几何不变性等性质。

8.3 主要实验步骤

(1) 打开工程

启动 VC6 或 VS2010 开发环境，对于 VC6 选择菜单：**【File->Open Workspace】**，在弹出的 OpenWorkspace 对话框中选择“实验 8”目录中的 bspline3d.dsw 文件，然后按下**【打开】**按钮，打开实验项目。对于 VS2010 选择菜单：**【文件->打开->项目/解决方案】**，在弹出的打开项目对话框中选择“实验 8”目录中的 bspline3d.sln 文件，然后按下打开项目，打开实验项目。

(2) 加入全局变量

打开 main.cpp 文件，翻到文件头部全局变量定义处，增加控制点网格相关全局变量，这些变量用于保存控制点网格，代码如下：

```
153 // 鼠标位置
154 int gMouseX = -1;
155 int gMouseY = -1;
156
157 // 是否进行鼠标追踪球
158 bool gIsStartTrackBall = false;
159
160 // 旋转矩阵
161 GMatrix3d gRotMatrix;
162
163 // 控制点网格
164 GPoint3d ** gCtrlGrid = NULL;
165
166 // 网格尺寸
167 int gNumX = 0;
168 int gNumZ = 0;
```

(3) 编写文件读取函数

在全局变量定义之后，增加文件读取函数 load()，载入控制点网格。该函数利用 C 标准运行库函数打开文本文件“surf_data.txt”读取数据。函数编写中需要注意 fscanf 函数中通配字符串的写法，需要用空格如“%f %f %f”，另外读取变量应定义为 float 而非 double 类型，代码如下：

```
192 // 载入控制点文件
193 bool load(const char *fileName)
194 {
195     FILE *fp = fopen(fileName, "r");
```

```

196
197     if(fp == NULL) return false;
198
199     int i, j;
200     float x, y, z;
201     double x0, x1, y0, y1, z0, z1;
202
203     fscanf(fp, "%d", &gNumZ);
204     fscanf(fp, "%d", &gNumX);
205
206     gCtrlGrid = new GPoint3d *[gNumZ];
207     for(i=0; i<gNumZ; i++)
208     {
209         gCtrlGrid[i] = new GPoint3d[gNumX];
210     }
211
212     x0 = y0 = z0 = HUGE_VAL;
213     x1 = y1 = z1 = -HUGE_VAL;
214     for(i=0; i<gNumZ; i++)
215     {
216         for(j=0; j<gNumX; j++)
217         {
218             fscanf(fp, "%f %f %f", &x, &y, &z);
219             gCtrlGrid[i][j].set(x, y, z);
220
221             if(x < x0) x0 = x;
222             if(x > x1) x1 = x;
223             if(y < y0) y0 = y;
224             if(y > y1) y1 = y;
225             if(z < z0) z0 = z;
226             if(z > z1) z1 = z;
227         }
228     }
229
230     for(i=0; i<gNumZ; i++)
231     {
232         for(j=0; j<gNumX; j++)
233         {
234             x = gCtrlGrid[i][j].x();
235             y = gCtrlGrid[i][j].y();
236             z = gCtrlGrid[i][j].z();
237             x = -1 + 2*(x-x0)/(x1-x0);
238             y = -0.5 + (y-y0)/(y1-y0);
239             z = -1 + 2*(z-z0)/(z1-z0);
240             gCtrlGrid[i][j].set(x, y, z);
241         }
242     }
243     fclose(fp);
244     return true;
245 }

```

(4) 编写控制点网格绘制函数

在文件读取函数 `load()` 之后, `onDisplay()` 函数之前, 增加控制点网格绘制函数 `drawCtrlGrid()`, 将 B 样条曲面控制点网格绘制出来, 检验数据读取的正确性。在写好 `drawCtrlGrid` 函数之后, 翻到 `onDisplay()` 函数增加该函数调用, 同时翻到

main()函数增加对 load()函数的调用，并运行程序检查效果，代码如下：

```

246 void drawCtrlGrid()
247 {
248     int i, j;
249
250     glColor3f(0.8f, 0.8f, 0.8f);
251     for(i=0; i<gNumZ; i++)
252     {
253         for(j=0; j<gNumX-1; j++)
254         {
255             glBegin(GL_LINES);
256             glVertex3dv(gCtrlGrid[i][j]);
257             glVertex3dv(gCtrlGrid[i][j+1]);
258             glEnd();
259         }
260     }
261
262     for(j=0; j<gNumX; j++)
263     {
264         for(i=0; i<gNumZ-1; i++)
265         {
266             glBegin(GL_LINES);
267             glVertex3dv(gCtrlGrid[i][j]);
268             glVertex3dv(gCtrlGrid[i+1][j]);
269             glEnd();
270         }
271     }
272 }

```

drawCtrlGrid()书写完成后，翻到 onDisplay()函数增加对该函数的调用，代码如下：

```

273 void onDisplay()
274 {
275     glClearColor(0, 0, 0, 0);
276     glClear(GL_COLOR_BUFFER_BIT);
277
278     glMatrixMode(GL_MODELVIEW);
279     glLoadIdentity();
280     glTranslatef(0, 0, -3.5f);
281     glMultMatrixd(gRotMatrix);
282     drawCtrlGrid();
283     ...
284 }

```

翻到 main()函数增加对 load()函数的调用，代码如下：

```

249 int main(int argc, char *argv[])
250 {
251     glutInit(&argc, argv);
252     glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
253     glutInitWindowSize(400, 300);
254     glutInitWindowPosition(100, 100);
255     glutCreateWindow("B-Spline 3D");
256     glInit();
257     load("surf_data.txt");
258     ...
259 }

```

完成上述修改后编译并运行程序，可以得到如图 8-1 所示的控制点网格图：

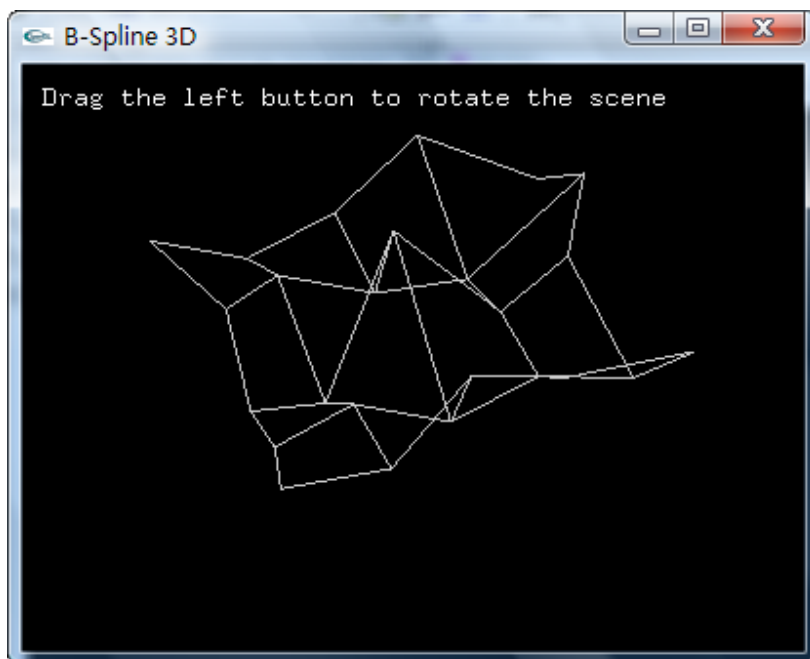


图 8-1 控制点网格绘制效果

(5) 编写 B 样条曲面计算函数

打开 ggltools.cpp 文件，在 gltDrawBSpline3d 函数之前编写 B 样条曲面计算函数，用于求取 B 样条曲面点，该函数根据式 (8-1) 和 (8-2) 计算给定参数(u, v)处的 B 样条曲面点，最外重的三次循环用于分别计算 x, y, z 位置矢量。函数编写时注意 B 样条基函数的系数是否书写正确，代码如下：

```

33 GPoint3d cubicBSplineBase3d(int ix0, int iz0, GPoint3d **ctrGrid,
34                             double u, double v)
35 {
36     int i, iu, iv;
37     double d, bu[4], bv[4];
38     GPoint3d pt;
39     bu[0] = ( -u * u * u + 3 * u * u - 3 * u + 1) / 6;
40     bu[1] = ( 3 * u * u * u - 6 * u * u + 4) / 6;
41     bu[2] = ( -3 * u * u * u + 3 * u * u + 3 * u + 1) / 6;
42     bu[3] = (u * u * u) / 6;
43     bv[0] = ( -v * v * v + 3 * v * v - 3 * v + 1) / 6;
44     bv[1] = ( 3 * v * v * v - 6 * v * v + 4) / 6;
45     bv[2] = ( -3 * v * v * v + 3 * v * v + 3 * v + 1) / 6;
46     bv[3] = (v * v * v) / 6;
47     for(i=0; i<3; i++)
48     {
49         d = 0;
50         for(iu=0; iu<4; iu++)
51         {
52             for(iv=0; iv<4; iv++)
53             {
54                 d += ctrGrid[iv+iz0][iu+ix0][i] * bu[iu] * bv[iv];
55             }
56         }

```

```

57     pt[i] = d;
58     }
59     return pt;
60 }

```

(6) 编写 B 样条曲面绘制函数

进入 ggltools.cpp 文件中的 gltDrawBSpline3d()函数, 完成 B 样条曲面的绘制。程序将每个 B 样条曲面片, 分成 10×10 个子曲面片, 调用 OpenGL 的 GL_QUADS, 四边形图元进行绘制, 代码如下

```

65 void gltDrawBSpline3d(GPoint3d **ctrGrid, int nx, int nz)
66 {
67     if(nx < 4 || nz < 4) return ;
68     int ix, iz, nu, nv, iu, iv;
69     double u, v, du, dv;
70     GPoint3d pt0, pt1;
71     nu = nv = 11;
72     du = dv = 1.0 / (nu-1);
73     glBegin(GL_LINES);
74     for(iz=0; iz<nz-3; iz++)
75     {
76         for(ix=0; ix<nx-3; ix++)
77         {
78             for(iv=0, v=0; iv<nv; iv++, v+=dv)
79             {
80                 for(iu=0, u=0; iu<nu; iu++, u+=du)
81                 {
82                     pt0 = cubicBSplineBase3d(ix, iz, ctrGrid, u, v);
83                     pt1 = cubicBSplineBase3d(ix, iz, ctrGrid, u, v+dv);
84                     pt2 = cubicBSplineBase3d(ix, iz, ctrGrid, u+du, v+dv);
85                     pt3 = cubicBSplineBase3d(ix, iz, ctrGrid, u+du, v);
86                     glVertex3dv(pt0);
87                     glVertex3dv(pt1);
88                     glVertex3dv(pt2);
89                     glVertex3dv(pt3);
90                 }
91             }
92         }
93     }
94     glEnd();
95 }

```

完成 gltDrawBSpline3d()函数后, 回到 main.cpp 的 onDisplay()函数增加对 gltDrawBSpline3d()函数的调用, 代码如下:

```

96 void onDisplay()
97 {
98     glClearColor(0, 0, 0, 0);
99     glClear(GL_COLOR_BUFFER_BIT);
100    glMatrixMode(GL_MODELVIEW);
101    glLoadIdentity();
102    glTranslatef(0, 0, -3.5f);
103    glMultMatrixd(gRotMatrix);
104    drawCtrlGrid();
105    gltDrawBSpline3d(gCtrlGrid, gNumX, gNumZ);

```

完成上述修改后，编译并运行程序，可以得到如图 8-2 所示的 B 样条网格曲面。

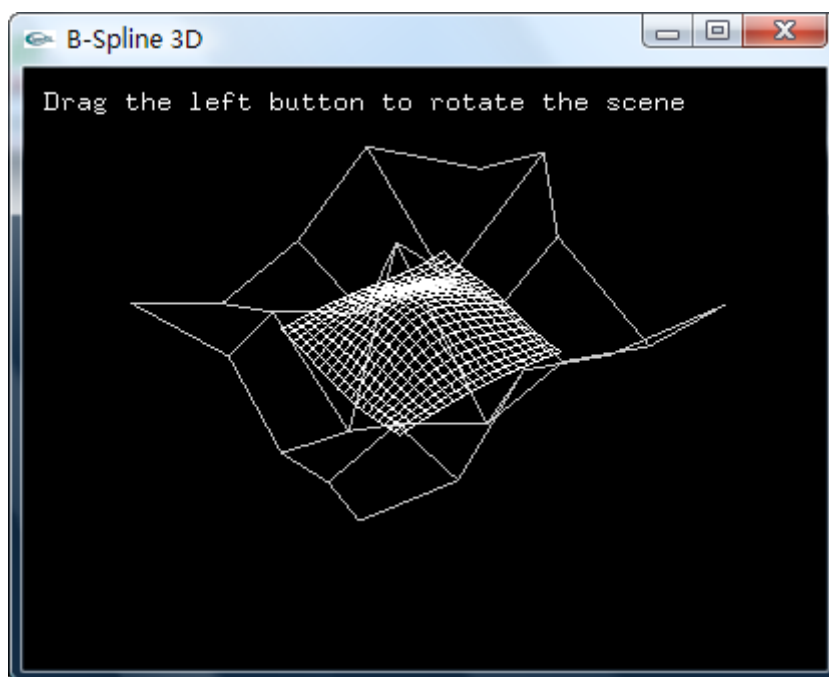


图 8-2 三次 B 样条曲面绘制效果

(7) 实现带光照效果的 B 样条曲面

打开 ggltools.cpp 文件，在 gltDrawBSpline3d()函数之前编写 B 样条曲面 U 方向和 V 方向切矢量计算函数，用于计算给定参数(u, v)处的 B 样条曲面在 U、V 方向上的切矢量，对 U、V 方向切矢量叉乘可以得到 B 样条曲面在(u, v)处的法矢量，可以用于 OpenGL 光照计算。

```

60 GPoint3d cubicBSplineUTangent(int ix0, int iz0, GPoint3d **ctrGrid,
61                               double u, double v)
62 {
63     int i, iu, iv;
64     double d, bu[4], bv[4];
65     GPoint3d pt;
66     bu[0] = ( -3 * u * u + 6 * u - 3 ) / 6;
67     bu[1] = ( 9 * u * u - 12 * u ) / 6;
68     bu[2] = ( -9 * u * u + 6 * u + 3 ) / 6;
69     bu[3] = ( 3 * u * u ) / 6;
70     bv[0] = ( -v * v * v + 3 * v * v - 3 * v + 1 ) / 6;
71     bv[1] = ( 3 * v * v * v - 6 * v * v + 4 ) / 6;
72     bv[2] = ( -3 * v * v * v + 3 * v * v + 3 * v + 1 ) / 6;
73     bv[3] = ( v * v * v ) / 6;
74     for(i=0; i<3; i++)
75     {
76         d = 0;
77         for(iu=0; iu<4; iu++)
78         {
79             for(iv=0; iv<4; iv++)
80             {
81                 d += ctrGrid[iv+iz0][iu+ix0][i] * bu[iu] * bv[iv];

```



```

82     }
83     }
84     pt[i] = d;
85 }
86 return pt;
87 }
88
89 GPoint3d cubicBSplineVTangent(int ix0, int iz0, GPoint3d **ctrGrid,
90                               double u, double v)
91 {
92     int i, iu, iv;
93     double d, bu[4], bv[4];
94     GPoint3d pt;
95     bu[0] = ( -u * u * u + 3 * u * u - 3 * u + 1) / 6;
96     bu[1] = ( 3 * u * u * u - 6 * u * u + 4) / 6;
97     bu[2] = ( -3 * u * u * u + 3 * u * u + 3 * u + 1) / 6;
98     bu[3] = (u * u * u) / 6;
99     bv[0] = (-3 * v * v + 6 * v - 3) / 6;
100    bv[1] = (9 * v * v - 12 * v) / 6;
101    bv[2] = (-9 * v * v + 6 * v + 3) / 6;
102    bv[3] = (3 * v * v) / 6;
103    for(i=0; i<3; i++)
104    {
105        d = 0;
106        for(iu=0; iu<4; iu++)
107        {
108            for(iv=0; iv<4; iv++)
109            {
110                d += ctrGrid[iv+iz0][iu+ix0][i] * bu[iu] * bv[iv];
111            }
112        }
113        pt[i] = d;
114    }
115    return pt;
116 }

```

修改 `gltDrawBSpline3d()` 函数，增加 B 样条曲面顶点法向量计算与设置，使得 OpenGL 可以进行正确的光照处理。

```

117 void gltBSpline3d(GPoint3d **ctrGrid, int nx, int nz)
118 {
119     if(nx < 4 || nz < 4) return ;
120
121     int ix, iz, nu, nv, iu, iv;
122     double u, v, du, dv;
123     GPoint3d pt0, pt1, pt2, pt3;
124     GVector3d v0, v1, n;
125     nu = nv = 11;
126     du = dv = 1.0 / (nu-1);
127     glBegin(GL_QUADS);
128     for(iz=0; iz<nz-3; iz++)
129     {
130         for(ix=0; ix<nx-3; ix++)
131         {
132             for(iv=0, v=0; iv<nv-1; iv++, v+=dv)
133             {
134                 for(iu=0, u=0; iu<nu-1; iu++, u+=du)
135                 {
136                     pt0 = cubicBSplineBase3d(ix, iz, ctrGrid, u, v);

```

```

137         pt1 = cubicBSplineBase3d(ix, iz, ctrGrid, u, v+dv);
138         pt2 = cubicBSplineBase3d(ix, iz, ctrGrid,
139                                 u+du, v+dv);
140         pt3 = cubicBSplineBase3d(ix, iz, ctrGrid, u+du, v);
141
142         v0 = cubicBSplineUTangent(ix, iz, ctrGrid, u, v);
143         v1 = cubicBSplineVTangent(ix, iz, ctrGrid, u, v);
144         n = v1.crossMult(v0);
145         n.normalize();
146         glNormal3dv(n);
147         glVertex3dv(pt0);
148
149         v0 = cubicBSplineUTangent(ix, iz, ctrGrid, u, v+dv);
150         v1 = cubicBSplineVTangent(ix, iz, ctrGrid, u, v+dv);
151         n = v1.crossMult(v0);
152         n.normalize();
153         glNormal3dv(n);
154         glVertex3dv(pt1);
155
156         v0 = cubicBSplineUTangent(ix, iz, ctrGrid,
157                                 u+du, v+dv);
158         v1 = cubicBSplineVTangent(ix, iz, ctrGrid,
159                                 u+du, v+dv);
160         n = v1.crossMult(v0);
161         n.normalize();
162         glNormal3dv(n);
163         glVertex3dv(pt2);
164
165         v0 = cubicBSplineUTangent(ix, iz, ctrGrid, u+du, v);
166         v1 = cubicBSplineVTangent(ix, iz, ctrGrid, u+du, v);
167         n = v1.crossMult(v0);
168         n.normalize();
169         glNormal3dv(n);
170         glVertex3dv(pt3);
171     }
172 }
173 }
174 }
175 glEnd();
176 }

```

回到 main.cpp 文件，在文件首部全局变量定义处增加一个全局光源变量。

```
15 GLight gLight0;
```

进入 glInit()函数初始化光源、设置光源参数、开启光源。

```

200 void glInit()
201 {
202     glEnable(GL_DEPTH_TEST);
203     glShadeModel(GL_SMOOTH);
204
205     gLight0.init(0, gDirLight);
206     gLight0.setPosition(0, 4, -4);
207     gLight0.turnOn();
208 }

```

进入 onDisplay()函数增加材质变量，设置材质参数，打开光照效果，为了便于在各个角度观察曲面，程序开启了双面光照效果，当曲面背对视点时仍然可以

观察到明亮的曲面。

```
96 void onDisplay()
97 {
98     glClearColor(0, 0, 0, 0);
99     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
100
101     glMatrixMode(GL_MODELVIEW);
102     glLoadIdentity();
103     glTranslatef(0, 0, -3.5f);
104     glMultMatrixd(gRotMatrix);
105
106     drawCtrlGrid();
107
108     GMaterial m;
109     glEnable(GL_LIGHTING);
110     m.setDiffuseColor(0.8, 0.8, 0);
111     m.setSpecularColor(0.4, 0.4, 0.4);
112     m.setShininess(7);
113     m.apply();
114     glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
115     gltBSpline3d(gCtrlGrid, gNumX, gNumZ);
116     glDisable(GL_LIGHTING);
117     ...
118 }
```

完成上述修改后，编译并运行程序可以得到如图 8-3 所示的 B 样条曲面效果。

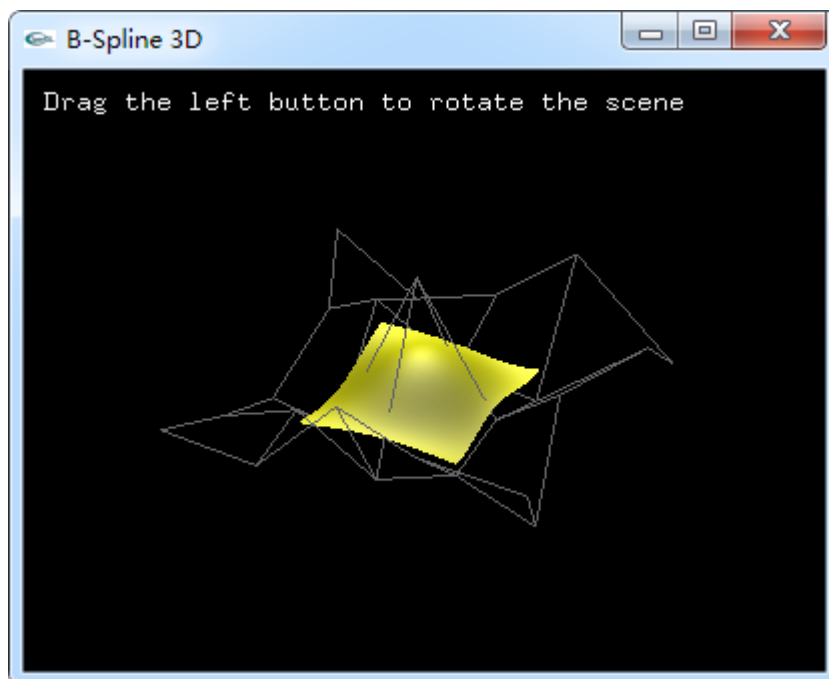


图 8-3 具有光照效果的三次 B 样条曲面