OpenGl超级宝典

**2.5.2 代码**

int \_tmain(int argc, char\* argv[])

{

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGBA);

glutCreateWindow("Simple");

glutDisplayFunc(RenderScene);

setupRC();

glutMainLoop();

return 0;

}

void setupRC()

{

glClearColor(0.0f, 0.0f, 1.0f, 1.0f);

}

void RenderScene()

{

glClear(GL\_COLOR\_BUFFER\_BIT);

glFlush();

}

**代码说明：**

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGBA)

告诉glut函数库，在创建窗口时应该使用哪种显示模式，这个函数告诉OpenGL使用一个单缓冲窗口，并且使用RGBA颜色模式。单缓冲模式意味着所有的绘图命令都是在显示窗口上执行的，双缓冲模式实际上是在一个屏幕之外的缓冲区中执行的，然后快速执行交换

glutCreateWindow("Simple")

创建OpenGL窗口，标题为Simple

glutDisplayFunc(RenderScene);

将RenderScene函数显示为回调函数，这意味着当窗口需要绘制时，GLUT将绘制调用这个函数。例如，当窗口第一次显示或者当窗口改变大小，或者当窗口从被覆盖的状态中恢复时，就会发生这个调用，这也是我们进行OpenGL渲染的地方

glutMainLoop();

这个函数启动了OpenGL框架的运行，glutMainLoop()函数一经调用便不再返回，直到程序终止。因此，它在应用程序中只能调用一次，这个函数处理所有操作系统特定的消息、击键等事件，直到程序终止。

glClearColor(0.0f, 0.0f, 1.0f, 1.0f)

这个函数设置了一种颜色，用于清除窗口

glClear(GL\_COLOR\_BUFFER\_BIT);

从窗口中清除最后一次所绘制的图形

glFlush();

刷新队列，简单地告诉OpenGL应该处理提供给它的绘图指令，而不是继续等待其他指令

**2.5.3代码**

const GLsizei WIDTH = 480;

const GLsizei HEIGHT = 800;

int \_tmain(int argc, char\* argv[])

{

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGBA);

glutInitWindowSize(WIDTH, HEIGHT);

glutCreateWindow("Drawing Rect");

glutDisplayFunc(SceneRender);

glutReshapeFunc(ChangeSize);

SetRC();

glutMainLoop();

system("pause");

return 0;

}

void SceneRender()

{

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(1.0f, 0.0f, 0.0f);

glRectf(-25.0f, 25.0f, 25.0f, -25.0f);

glutSwapBuffers();

}

void ChangeSize(GLsizei width, GLsizei height)

{

GLfloat aspectRatio = 0.0f;

if(height)

aspectRatio = ((GLfloat)width / WIDTH) / ((GLfloat)height / HEIGHT);//这里是表示改变形状后x方向的每一份像素个数与y方向每一份像素个数比

else

height = 1;

//设置视图为窗口大小

glViewport(0, 0, width, height);

//更改坐标体系

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

//建立裁剪区域

if(width < (GLfloat)WIDTH / (GLfloat)HEIGHT \* height)

glOrtho(-(GLfloat)WIDTH / 2, (GLfloat)WIDTH / 2, -(GLfloat)HEIGHT / 2 / aspectRatio, (GLfloat)HEIGHT / 2 / aspectRatio, -100.0f, 100.0f);

else

glOrtho(-(GLfloat)WIDTH / 2 \* aspectRatio, (GLfloat)WIDTH / 2 \* aspectRatio, -(GLfloat)HEIGHT / 2, (GLfloat)HEIGHT / 2, -100.0f, 200.0f);

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

}

**代码说明：**

glMatrixMode：指定哪一个矩阵是当前矩阵，可选值GL\_MODELVIEW、GL\_PROJECTION、GL\_TEXTURE。

GL\_MODELVIEW：对模型视景矩阵堆栈应用随后的矩阵操作。这个是对模型视景的操作，接下来的语句描绘一个以模型为基础的场景，这样来设置参数，接下来用到的就是像gluLookAt这样的函数

GL\_PROJECTION：对投影矩阵应用随后的矩阵操作。及时对投影相关进行操作，也就是把物体投影到一个平面上，就像我们照相一样，把三维物体投影到二维的平面上，接下来的语句可以是跟透视相关的函数，比如glFrustum或gluPerspective

GL\_TEXTURE：对纹理矩阵堆栈应用随后的矩阵操作

与glLoadIdentity一同使用，glLoadIdenttry功能是重置当前指定的矩阵为单位矩阵

具体说明：

我们在三维世界如果要观察一个物体可以：

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，我们可以放大或者缩小它（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，等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。例如，gluPerspective的意思是设定投影变换，但是要先通过glMatrixMode设定成投影矩阵才可以变换，如果你glMatrixMode设定成模型矩阵变换，gluPerspective就不会设定在投影矩阵，而是模型矩阵，导致没有图像。

我们在实际操作过程中，通常将裁剪区域与视口保持一致，则可以让正方形在窗口大小改变的情况下不变形

**OpenGL实现多视口**

<http://blog.csdn.net/tulun/article/details/5537750>

**openGL绘图**

GL\_POINTS、GL\_LINES、GL\_LINES\_STRIP、GL\_LINE\_LOOP、GL\_TRIANGLES、GL\_TRIANGLE\_SKIP、GL\_TRIANGLE\_FAN、GL\_POLYGON、GL\_QUADS、GL\_QUAD\_STRIP

**画一个圆：**

//绘制一个圆

glBegin(GL\_POLYGON);

for(GLsizei i = 0; i < N; i++)

{

glVertex2f(RADIUS \* cos(i \* 2 \* PI / N), RADIUS \* sin(i \* 2 \* PI / N));

}

glEnd();

**多边形的两面及绘制**

从三维的角度来看，一个多边形具有两个面，每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

**反转：**

glFrontFace(GL\_CCW)//设置逆时针方向为正面CounterClockWise

glFrontFace(GL\_CW)//设置顺时候方向为正面ClockWise

注意：比如一个瓶子。同属于瓶子的外侧，但在某些地方算是正面，某些地方算是反面

一个例子：

//测试多边形两面及绘制模式

glPolygonMode(GL\_FRONT, GL\_FILL); //正面为填充模式

glPolygonMode(GL\_BACK, GL\_LINE); //背面为轮廓模式

glFrontFace(GL\_CCW); //逆时针为正面

glBegin(GL\_QUADS); **// 注意上面的设置都应该放在glBegin之前，否则不起作用**

glVertex2f(-25.0f, -25.0f);

glVertex2f(0.0f, -25.0f);

glVertex2f(0.0f, 0.0f);

glVertex2f(-25.0f, 0.0f);

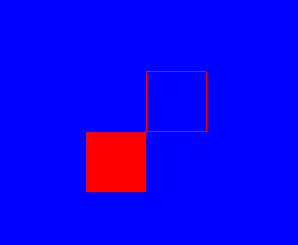
glVertex2f(0.0f, 0.0f);

glVertex2f(0.0f, 25.0f);

glVertex2f(25.0f, 25.0f);

glVertex2f(25.0f, 0.0f);

glEnd();



**剔除多边形表面：**

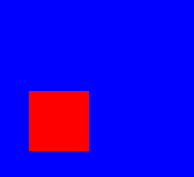
首先要使用glEnable(GL\_CULL\_FACE)启动剔除功能，然后使用glCullFace来进行剔除，glCullFace的参数可以是GL\_FRONT、GL\_BACK或者GL\_FRONT\_AND\_BACK

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，所有多边形都将被剔除，所以所看见的就只有点和直线

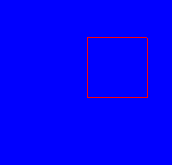
对上面的代码添加：

glEnable(GL\_CULL\_FACE);

glCullFace(GL\_BACK);



对上面的代码添加：



**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)启动镂空模式，然后采用glPolygonStipple这是镂空样式

void glPolygonStipple(const Glubyte\* mask);

其中的参数mask指向一个长度为128字节的空间表示了一个32\*32的矩形应该如何进行镂空。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

//镂空多边形

glEnable(GL\_POLYGON\_STIPPLE);

glPolygonStipple(Mask);

glRectf(-50.0f, 0.0f, 0.0f, -50.0f);

glDisable(GL\_POLYGON\_STIPPLE);

glRectf(0.0f, 50.0f, 50.0f, 0.0f);

static GLubyte Mask[128] =

{

0x00, 0x00, 0x00, 0x00, // 这是最下面的一行

0x00, 0x00, 0x00, 0x00,

0x03, 0x80, 0x01, 0xC0,

0x06, 0xC0, 0x03, 0x60,

0x04, 0x60, 0x06, 0x20,

0x04, 0x30, 0x0C, 0x20,

0x04, 0x18, 0x18, 0x20,

0x04, 0x0C, 0x30, 0x20,

0x04, 0x06, 0x60, 0x20,

0x44, 0x03, 0xC0, 0x22,

0x44, 0x01, 0x80, 0x22,

0x44, 0x01, 0x80, 0x22,

0x44, 0x01, 0x80, 0x22,

0x44, 0x01, 0x80, 0x22,

0x44, 0x01, 0x80, 0x22,

0x44, 0x01, 0x80, 0x22,

0x66, 0x01, 0x80, 0x66,

0x33, 0x01, 0x80, 0xCC,

0x19, 0x81, 0x81, 0x98,

0x0C, 0xC1, 0x83, 0x30,

0x07, 0xE1, 0x87, 0xE0,

0x03, 0x3F, 0xFC, 0xC0,

0x03, 0x31, 0x8C, 0xC0,

0x03, 0x3F, 0xFC, 0xC0,

0x06, 0x64, 0x26, 0x60,

0x0C, 0xCC, 0x33, 0x30,

0x18, 0xCC, 0x33, 0x18,

0x10, 0xC4, 0x23, 0x08,

0x10, 0x63, 0xC6, 0x08,

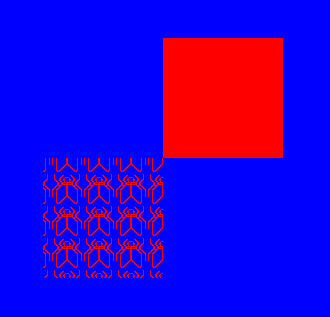
0x10, 0x30, 0x0C, 0x08,

0x10, 0x18, 0x18, 0x08,

0x10, 0x00, 0x00, 0x08 // 这是最上面的一行

};

运行结果：



**OpenGL的颜色模式：**

openGL支持两种颜色模式：一种是RGBA，一种是颜色索引模式

无论哪种颜色模式，计算机都必须为每一个子像素保存一些数据，不同的是，RGBA模式中，数据直接代表了颜色，而在颜色索引模式中，数据代表的是一个索引，要得到真正的颜色，还必须去查索引表

**顶点着色模式：glShadeMode(GL\_SMOOTH)与glShadeMode(GL\_FLAT)**

**3.4.2代码**

void RenderScene()

{

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(1.0f, 0.0f, 0.0f);

glPushMatrix();

glRotatef(30.0f, 1.0f, 0.0f, 0.0f);

glRotatef(30.0f, 0.0f, 1.0f, 0.0f);

glBegin(GL\_POINTS);

GLfloat z = -50.0f;

for(GLfloat angle = 0.0f; angle <= (2 \* GL\_PI) \* 3; angle += 0.1f)

{

GLfloat x = 50.0f \* sin(angle);

GLfloat y = 50.0f \* cos(angle);

glVertex3f(x, y, z);

z += 0.5f;

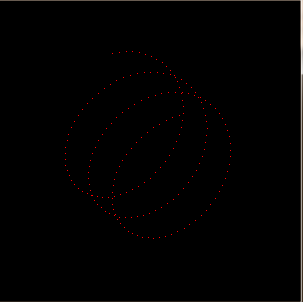
}

glEnd();

glPopMatrix();

glutSwapBuffers();

}



**为什么要使用glPushMatrix和glPopMatrix函数？**

将本次需要执行的缩放、平移等操作放在glPushMatrix和glPopMatrix之间。两者的配对可以消除上一次的变换对本次变换的影响。使本次变换是以世界坐标系的原点为参考点进行。

1. OpenGL中的modelview矩阵变换是以个马尔科夫过程：上一次的变换结果对本次变换有影响，上次modelview变换后物体在世界坐标系下的位置是本次modelview变换的起点，默认时本次变换和上次变换不独立
2. OpenGL物体建模实际上是分两步走的。第一步，在世界坐标系的原点位置绘制出该物体；第二步，通过modelview变换矩阵对世界坐标系原点处的物体进行仿射变换，将该物体移动到直接坐标系的目标位置处
3. 将modelview变换放在glPushMatrix和glPopMatrix之间可以使本次变换和上次变换独立
4. 凡是使用glPushMatrix和glPopMatrix的程序一般可以判定是采用世界坐标系建模，即世界坐标系固定，modelview矩阵移动物体

**设置点的大小**

当我们绘制一个点时，在默认情况下大小是1个像素，我们可以使用glPointiSize函数修改点的大小

Void glPointSize（Glfloat size）

glPointSize函数接受一个参数，用于指定被绘制点的近似直径（以像素计），但是，点的大小设置还存在限制，我们应该确保自己所指定的点的大小是可行的，我们可以采用下面的方式获取点大小的范围以及它们之间的最小间隔值

void RenderScene()

{

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(1.0f, 0.0f, 0.0f);

GLfloat sizes[2];

GLfloat step;;

GLfloat curSize;

glGetFloatv(GL\_POINT\_SIZE\_RANGE, sizes);

glGetFloatv(GL\_POINT\_SIZE\_GRANULARITY, &step);

curSize = sizes[0];

glPushMatrix();

glRotatef(30.0f, 1.0f, 0.0f, 0.0f);

glRotatef(30.0f, 0.0f, 1.0f, 0.0f);

GLfloat x, y, z = -50.0f;

for(GLfloat angle = 0.0f; angle <= (2 \* GL\_PI) \* 3; angle += 0.1f)

{

x = 50.0f \* sin(angle);

y = 50.0f \* cos(angle);

glPointSize(curSize);

glBegin(GL\_POINTS);

glVertex3f(x, y, z);

glEnd();

z += 0.5f;

curSize += step;

}

glPopMatrix();

glutSwapBuffers();

}



必须要注意到，glPointSize必须在glBegin/glEnd语句之外调用，在两者之内调用并没有作用（已经证实），在glBegin/glEnd之间并不是所有的OpenGL函数都是合法的，尽管glPointSize会影响它后面所有将被绘制的点，但是在调用glBegin之前，这些点并不被绘制。

在上面的程序中，sizes数组将包含两个元素，分别表示glPointSize的最小有效值和最大有效值，另外，变量step表示点的大小范围之内最小允许的步进值。OpenGL规范要求只支持1个单位的大小，即1.0。例如，MicroSoft所提供的OpenGL软件实现允许点大小的范围从0.5到10.0，最小步进值为0.125，指定一个位于范围之外的值并不会产生错误，系统会使用最大或最小受支持的值，也就是最靠近指定值的有效值。

点和其他图形不一样，它并不会收到透视除法的影响，也就是说，当它们离视点更远时，它们看上去并不会变得更小，离视点更近时，看上去也不会变得更大。另外，点总是正方形的像素，即使使用glPointSize增加点的大小，情况也不会发生变化，我们所得到的只是更大的方块而已，为了获的圆点，必须在抗锯齿模式点绘制点。

**为什么可以用一个小于1的值作为点大小的步进值呢？**

如果1.0表示一个像素，那么怎样才能绘制一个大小为2.5的点呢？

glPointSize函数所指定的点的大小并不是以像素为单位的准确点的大小，而是一个圆的近似直径，这个圆恰好包含了绘制这个点所使用的所有像素

**Opengl的glRotatef旋转方向判断**

具体法则就是右手法则，即张开右手掌，大拇指指向向量方向，然后四指往里握成拳头，四手指旋转的方向即旋转方向，当给定角度为负数时候，旋转方向就与手指旋转方向相反了

**3.6.1代码**

void RenderScene()

{

cout << "RenderScene" << endl;

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(1.0f, 0.0f, 0.0f);

glBegin(GL\_LINES);

for(GLfloat angle = 0.0f; angle < GL\_PI; angle += GL\_PI / 20)

{

GLfloat x = 50.0f \* cos(angle);

GLfloat y = 50.0f \* sin(angle);

glVertex2f(x, y);

x = 50.0f \* cos(angle + GL\_PI);

y = 50.0f \* sin(angle + GL\_PI);

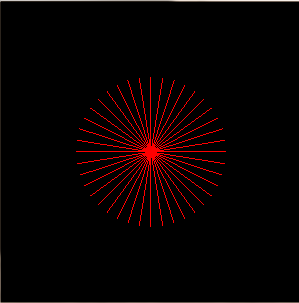
glVertex2f(x, y);

}

glEnd();

glutSwapBuffers();

}



每两个指定的顶点画一条直线，如果GL\_POINTS指定奇数个顶点，那么最后一个顶点将被忽略

**画线带：**

glBegin(GL\_LINE\_STRIP);

glVertex3f(0.0f, 0.0f, 0.0f);

glVertex3f(50.0f, 50.0f, 0.0f);

glVertex3f(50.0f, 100.0f, 0.0f);

glEnd();

**画线环：**

glBegin(GL\_LINE\_LOOP);

glVertex3f(0.0f, 0.0f, 0.0f);

glVertex3f(50.0f, 50.0f, 0.0f);

glVertex3f(50.0f, 100.0f, 0.0f);

glEnd();

**用直线构成近似曲线：**

void RenderScene()

{

cout << "RenderScene" << endl;

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(1.0f, 0.0f, 0.0f);

glPushMatrix();

glRotatef(30.0f, 1.0f, 0.0f, 0.0f);

glRotatef(30.0f, 0.0f, 1.0f, 0.0f);

GLfloat z = -50.0f;

glBegin(GL\_LINE\_STRIP);

for(GLfloat angle = 0.00f; angle <= GL\_PI \* 2 \* 3; angle += 0.1f)

{

GLfloat x = 50.0f \* cos(angle);

GLfloat y = 50.0f \* sin(angle);

glVertex3f(x, y, z);

z += 0.5f;

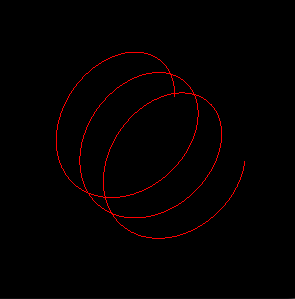
}

glEnd();

glPopMatrix();

glutSwapBuffers();

}



**设置直线的宽度：**

采用glLineWidth函数，该函数接受一个参数，用于指定被绘制直线的近似宽度（以像素为单位），和点的大小一样，直线的宽度也是要受到限制的，所以要确保自己所指定的直线宽度是有效的

void RenderScene()

{

cout << "RenderScene" << endl;

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(1.0f, 0.0f, 0.0f);

GLfloat sizes[2];

GLfloat step;

glGetFloatv(GL\_LINE\_WIDTH\_RANGE, sizes);

glGetFloatv(GL\_LINE\_WIDTH\_GRANULARITY, &step);

GLfloat curLineWidth = sizes[0];

for(GLfloat y = -90.0f; y <= 90.0f; y += 20.0f)

{

glLineWidth(curLineWidth);

glBegin(GL\_LINES);

glVertex3f(-80.0f, y, 0.0f);

glVertex3f(80.0f, y, 0.0f);

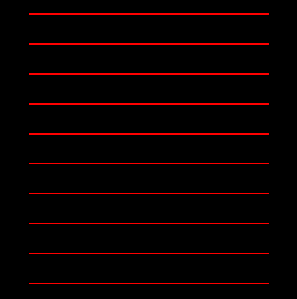
glEnd();

curLineWidth += 0.1f;

}

glutSwapBuffers();

}



**注意：**如果将glLineWidth(curLineWidth)放入glBegin与glEnd对中，则线条的宽度始终不变，得不到想要的效果。

**直线点画**

除了修改直线的宽度之外，还可以用点线和虚线模式来创建直线，这称为**点画，**为了使用直线画点，首先必须启动这面这行代码启动点画功能。

glEnable(GL\_LINE\_STIPPLE);

注意，调用glEnable所开启的任何功能都可以采用glDisable函数调用进行取消，然后可以用glLineStripple函数创建这种模式，以便进行画。

void glLineStripple(Glint factor, Glushort pattern);

pattern参数是一个16位的值，它指定了一个模式，供绘直线所用。它的每一位都表示线段的一部分或者处于打开状态，或者处于关闭状态。在默认情况下，每一位对应一个像素，但是factor可以作为乘法因子增加模式的宽度，例如，把factor设置为5将导致模式中的每个位代表一行中的连续5个像素，它们同时处于打开或关闭状态。另外，模式的第0位（最低有效位）首先被用于指定直线。

**为什么点画的位模式在用于画线时为什么以相反的方向顺序使用？**

在内部，openGL把这个模式向左移动一位可以快速获得下一个掩码值，主要是通过左移一位判断奇偶更加快捷

**3.6.4代码**

void RenderScene()

{

cout << "RenderScene" << endl;

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(1.0f, 0.0f, 0.0f);

GLfloat factor = 1;

**GLushort pattern = 0x5555;**

glEnable(GL\_LINE\_STIPPLE);

for(float y = -90.0f; y <= 90.0f; y += 20.0f)

{

**glLineStipple(factor, pattern);**

glBegin(GL\_LINES);

glVertex3f(-80.0f, y, 0);

glVertex3f(80.0f, y, 0);

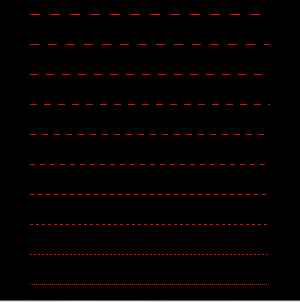
glEnd();

factor++;

}

glutSwapBuffers();

}



**3.7.1代码**

**绘制三角形**

glClear(GL\_COLOR\_BUFFER\_BIT);

glBegin(GL\_TRIANGLES);

glVertex2f(0.0f, 0.0f);

glVertex2f(25.0f, 25.0f);

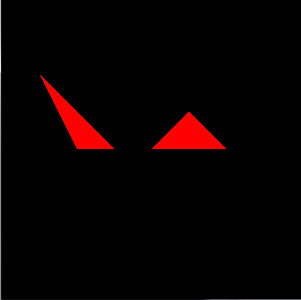
glVertex2f(50.0f, 0.0f);

glVertex2f(-50.0f, 0.0f);

glVertex2f(-75.0f, 50.0f);

glVertex2f(-25.0f, 0.0f);

glEnd();



这两个三角形将会按照当前选择的颜色进行填充，如果没有指定绘图颜色，那么就无法获得该结果。

**环绕：**顶点的指定顺序及方向的组合称为环绕

在默认情况下，openGL认为逆时针方向环绕的一面是多边形的正面

**为什么要给物体分为背面和正面呢？**

因为我们常常希望为一个多边形的正面和背面分别设置不用的物理特性，我们可以完全隐藏一个物体的背面，或者给它设置不同的颜色和反射属性，在一个场景中，把所有多边形保持环绕方向相同并使用正面多边形来绘制所有实心物体的外表面是非常重要的

如果想改变openGL的这个默认行为，可以调用下面这个函数：

glFrontFace(GL\_CW);

GL\_CW参数告诉openGL顺时针环绕的多边形将被认为是正面的，为了把多边形的正面重新恢复为逆时针环绕，可以再这个函数中使用GL\_CCW参数

**三角形带**

对于许多表面和形状，可能需要绘制几个相连的三角形，使用GL\_TRIANGLE\_STRIP图元，可以绘制一串相连的三角形，从而节省大量的时间。

**为什么使用三角形带而不是分别指定每个三角形？**

首先，用前三个顶点指定了第一个三角形之后，对于接下来的每个三角形，只需要再指定一个顶点，需要绘制大量的三角形时，采用这种方式可以节省大量的程序代码和数据存储空间

其次，运算性能的提高和带宽的节省，更少的顶点意味着数据从内存传输到图形卡的速度更快，并且顶点变换的次数也可以更少一些

**三角形扇**

除了三角形带之外，还可以使用GL\_TRIANGLE\_FAN创建一组围绕一个中心点的相连三角形。

用前3个顶点指定第1个三角形，后续的每个顶点和原点以及前驱的那个顶点形成了接下来的那个三角形

void SetRC()

{

glClearColor(0, 0, 0, 0);

glShadeModel(GL\_FLAT);

}

void RenderScene()

{

cout << "Render Scene" << endl;

glClear(GL\_COLOR\_BUFFER\_BIT);

GLsizei ipvort = 0;

glBegin(GL\_TRIANGLE\_FAN);

glColor3f(1.0f, 0.0f, 0.0f);

glVertex3f(0.0f, 0.0f, 0.0f);

for(GLfloat angle = 0.0f; angle <= 2 \* GL\_PI + 0.1f; angle += (GL\_PI / 8.0f))

{

GLfloat x = 50.0f \* cos(angle);

GLfloat y = 50.0f \* sin(angle);

if(ipvort % 2 == 0)

glColor3f(0.0f, 1.0f, 0.0f);

else

glColor3f(1.0f, 0.0f, 0.0f);

glVertex3f(x, y, 0.0f);

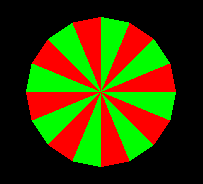
ipvort++;

}

glEnd();

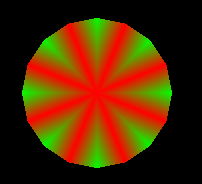
glutSwapBuffers();

}



如果改为：

glShadeModel(GL\_SMOOTH);



说明：

在openGL中，颜色实际上是以顶点为单位指定的，而不是以多边形为单位，着色模式决定了多边形是单调着色的（最后一个顶点的颜色作为整个多边形的填充颜色）还是渐变着色的（根据每个顶点的颜色进行平滑着色）

**注意：对于旋转而言，必须要把它放在glBegin/glEnd函数以外，否则不会起反应**

glRotatef(45.0f, 0.0f, 0.0f, 1.0f);//注意这里必须放在glBegin/glEnd后面glBegin(GL\_TRIANGLES);

glVertex3f(0.0f, 0.0f, 0.0f);

glVertex3f(50.0f, 0.0f, 0.0f);

glVertex3f(0.0f, 50.0f, 0.0f);

glEnd();

**深度缓冲区**

if(bDepth)

glEnable(GL\_DEPTH\_TEST);

else

glDisable(GL\_DEPTH\_TEST);

深度测试是一种有效的用于隐藏表面消除的技巧，openGL提供了一些函数，允许在幕后完成这个任务，它的概念非常简单：当一个像素被绘制时，它将被设置一个值（称为z值），以表示它和观察者之间的距离，以后当这个屏幕需要绘制另一个像素时，新像素的z值就会与原先已经存储的那个像素进行比较。在内部，这个任务是通过深度缓冲区实现的，它存储了屏幕上每个像素的深度值。

**裁剪区域**

有一种提高渲染性能的方法就是只对屏幕上进行了修改的部分进行更新，OpenGL把渲染区域限制在窗口内部一块更小的矩形区域中，OpenGL允许在窗口中制定一个裁剪矩形，让渲染只在这个区域中进行

void glScissor(Glint x, GLint y, GLsizei width, GLsizei height)

X和y指定了裁剪框的左下角，width与height指定了裁剪框的宽度与高度（以像素为单位）

注意这里的x与y和glViewport函数中的参数x和y一样，都是指实际窗口的左下角

void RenderScene()

{

glClearColor(1.0f, 0.0f, 0.0f, 0.0f);

glClear(GL\_COLOR\_BUFFER\_BIT);

glClearColor(0.0f, 1.0f, 0.0f, 0.0f);

glEnable(GL\_SCISSOR\_TEST);

glScissor(100, 100, 280, 600);

glClear(GL\_COLOR\_BUFFER\_BIT);

glClearColor(0.0f, 0.0f, 1.0f, 0.0f);

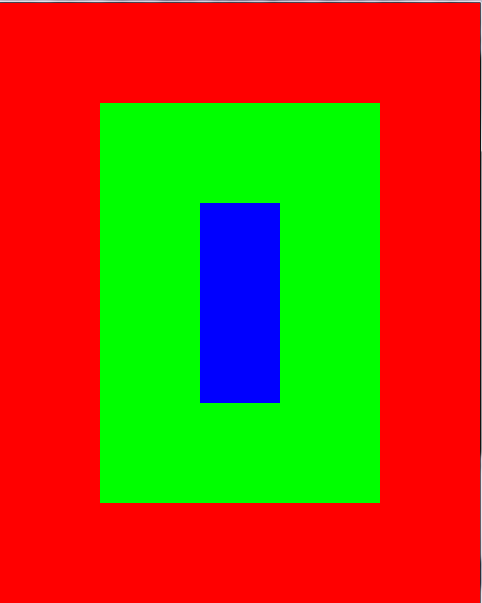
glScissor(200, 200, 80, 400);

glClear(GL\_COLOR\_BUFFER\_BIT);

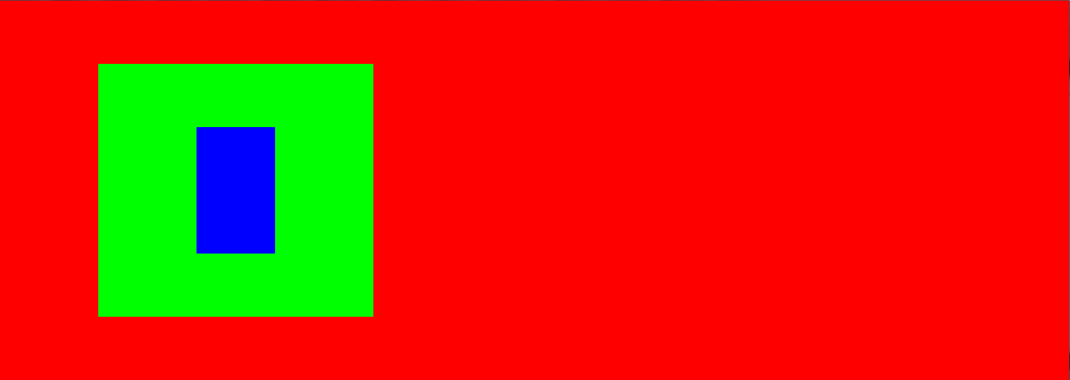
glDisable(GL\_SCISSOR\_TEST);

glutSwapBuffers();

}



实际上当我们改变实际窗口大小时候，该区域仍然不变



**模型视图变换**

视图变换：

视图变换是场景所应用的第一个变换。它用于确定场景的拍摄点，在默认情况下，在透视投影中，观察者是从原点向Z轴的负方向看过去（垂直穿入监视器），这个观察点相对于视觉坐标系统进行移动，以提供一个特定的拍摄点，当观察点位于原点时，场景中所绘制的z值为正的物体就位于观察者的后面

视图变换允许吧观察点放在自己所希望的任何位置，并允许在任何方向上观察场景，确定视图变换就像在场景中放置照相机并让它指向某个方向

作为总体原则，在进行任何其他变换之前必须先指定视图变换，原因是视图变换的效果相当于根据视觉坐标系统移动当前所使用的坐标系统，然后，根据最新修改的坐标系统进行其他所有的后续变换。

模型变换：

模型变换用于对模型以及模型内部的特定物体进行操纵，可以移动、旋转以及放缩

**投影变换**

投影变换是在模型视图变换之后应用于物体顶点上的，这种投影实际上定义了可视区域，并建立了裁剪平面，裁剪平面是3D空间的平面方程式，OpenGL采用它来确定几何图形能否被观察者看到，更为具体地说，投影变换指定了一个完成的场景（在所有模型变换都完成以后）投影到屏幕上的最终图像

**视口变换**

在上面所有的操作都完成后，最终所获得的是场景的二维投影，将被映射到屏幕上的某个窗口，这种到物理窗口坐标的映射是最后一个完成的变换，称为视口变换

**变换管线**

从原始的顶点数据通往屏幕坐标的路是相当漫长的，如下图所示



首先，把坐标转变为一个1\*4的矩阵，前三个值分别是x、y和z坐标，第四个元素是放缩因子，通常为1.0

接着，把顶点和模型视图矩阵相乘，产生经过变换的视觉坐标，然后将这个视觉坐标与投影矩阵相乘，产生裁剪坐标，这样就有效地消除了可视区域之外的所有数据，这个裁剪区域随后除以w坐标，产生规范化的设备坐标。W值可能会被投影矩阵或模型视图矩阵所修改，具体取决于顶点所发生的变换

最后，我们把这个坐标通过视口变换映射到一个2D平面中

**矩阵堆栈**

三种矩阵堆栈，模型视图堆栈、投影堆栈以及纹理堆栈，这些堆栈都有最大深度，我们可以使用下面两个函数之一来获取这个值：

glGet(GL\_MAX\_MODELVIEW\_STACK\_DEPTH)

和

glGet(GL\_MAX\_PROJECTION\_STACK\_DEPTH)

如果超过了这个深度就会产生GL\_STACK\_OVERFLOW错误，堆栈的深度因不同的OpenGL实现而不同，在Microsoft软件实现中，模型视图矩阵的最大值为32，投影矩阵堆栈的最大值为2

通过使用矩阵堆栈的一个例子：

void RenderScene()

{

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

static GLfloat angle = 0.0f;

glEnable(GL\_DEPTH\_TEST);

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

glTranslatef(0.0f, 0.0f, -50.0f);

//创建红色原子核

glColor4f(1.0f, 0.0f, 0.0f, 0.0f);

glutSolidSphere(10.0f, 15, 15);

//创建绿色电子

glPushMatrix();

glColor4f(0.0f, 1.0f, 0.0f, 0.0f);

glRotatef(45.0f, 0.0f, 0.0f, 1.0f);

glRotatef(angle, 0.0f, 1.0f, 0.0f);

glTranslatef(25.0f, 0.0f, 0.0f);

glutSolidSphere(6.0f, 15, 15);

glPopMatrix();

//创建蓝色电子

glPushMatrix();

glColor4f(0.0f, 0.0f, 1.0f, 0.0f);

glRotatef(angle, 0.0f, 1.0f, 0.0f);

glTranslatef(-20.0f, 0.0f, 0.0f);

glutSolidSphere(5.0f, 15, 15);

glPopMatrix();

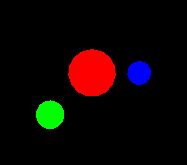
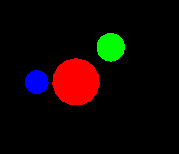
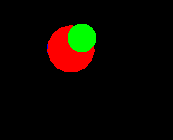
angle += 5.0f;

if(angle >= 360.0f)

angle = 0.0f;

glutSwapBuffers();

}

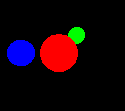
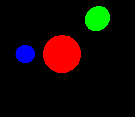
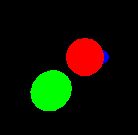
  

**透视投影**

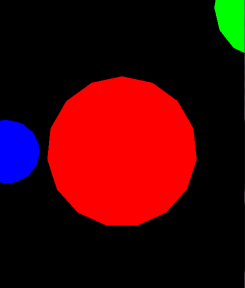
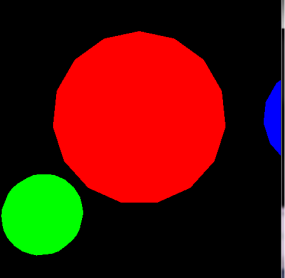
透视投影执行透视除法，对距离观察者较远的物体进行缩短和收缩。在投影到屏幕之后，可视区域后端和前端的度量单位并不相同，因此，如果两个逻辑大小相同的物体分别位于可视区域的前面和后面，前者看上去要比后者大

在上面的程序中如果我们改为：

gluPerspective(**145.0f**, (GLfloat)width / (GLfloat)height, 25.0f, 425.0f);



如果改为：gluPerspective(**45.0f**, (GLfloat)width / (GLfloat)height, 25.0f, 425.0f);



**高级矩阵操作**

openGL在表示一个4\*4的矩阵时并没有使用浮点型的二维数组，而是使用一个包含16个浮点值的一维数组来表示，为什么不用二维数组表示？主要是第一种方式更为高效



许多OpenGL实现具有所谓的硬件T&L（变换和光照），这意味着变换矩阵与成千上万个顶点的乘法是在特殊的图形硬件上进行的，操作的速度非常非常快

下面的一个例子是自己加载矩阵：

M3DMatrix44f transformationMatrix;//旋转矩阵的存储空间

static GLfloat yRot = 0.0f;//动画的旋转角度

yRot += 0.5f;

//创建一个旋转矩阵

m3dRotationMatrix44(transformationMatrix, m3dDegToRad(yRot), 0.0f, 1.0f, 0.0f);

**transformationMatrix[12] = 0.0f;**

**transformationMatrix[13] = 0.0f;**

**transformationMatrix[14] = -50.0f;**

//加载矩阵

glLoadMatrixf(transformationMatrix);

glBegin(GL\_LINES);

glVertex3f(0.0f, 0.0f, 0.0f);

glVertex3f(25.0f, 0.0f, 0.0f);

glEnd();

最终运行的记过就是一条直线绕着Y轴旋转且距离Y轴50个单位（已经验证）

上面的功能也可以采用下面的方法执行：

M3DMatrix44f transformationMatrix, rotateNatrix;//旋转矩阵的存储空间

static GLfloat yRot = 0.0f;//动画的旋转角度

yRot += 0.5f;

//创建一个旋转矩阵

m3dRotationMatrix44(rotateNatrix, m3dDegToRad(yRot), 0.0f, 1.0f, 0.0f);

**m3dTranslationMatrix44(transformationMatrix, 0.0f, 0.0f, -50.0f);**

**m3dMatrixMultiply44(transformationMatrix, transformationMatrix, rotateNatrix);**

//加载矩阵

glLoadMatrixf(transformationMatrix);

glBegin(GL\_LINES);

glVertex3f(0.0f, 0.0f, 0.0f);

glVertex3f(25.0f, 0.0f, 0.0f);

glEnd();

在下面的方法中是先得到两个4\*4的矩阵然后相乘得到目标矩阵，在上面的方法中是直接改变4\*4的矩阵，因此上面的方法不需要相乘

OpenGL还提供了自己的矩阵乘法函数：glMultMatrix，这个函数接受一个矩阵，并把它与当前加载的矩阵相乘，然后把结果存储在矩阵堆栈的顶部，因此上面的代码还可以改为：

M3DMatrix44f transformationMatrix, rotateNatrix;//旋转矩阵的存储空间

static GLfloat yRot = 0.0f;//动画的旋转角度

yRot += 0.5f;

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

//创建一个旋转矩阵

m3dRotationMatrix44(rotateNatrix, m3dDegToRad(yRot), 0.0f, 1.0f, 0.0f);

m3dTranslationMatrix44(transformationMatrix, 0.0f, 0.0f, -50.0f);

glMultMatrixf(transformationMatrix);

glMultMatrixf(rotateNatrix);

glBegin(GL\_LINES);

glVertex3f(0.0f, 0.0f, 0.0f);

glVertex3f(25.0f, 0.0f, 0.0f);

glEnd();

程序员一般有两种指定变换的方式：

（1）指定矩阵：glLoadMatrix与glMultMatrix

（2）指定操作：glRotatef、glOrtho等

模型视图矩阵堆栈：32个4\*4矩阵对阵

投影矩阵堆栈：2个4\*4矩阵

初始栈顶矩阵都是单位矩阵

在OPENGL的红宝书里面，首先提出了**全局固定坐标系统**和**局部移动坐标系统**，这里这两个坐标系的代码实际上是差不多的。区别在于，全局用的是多个矩阵相乘即glMultMatrixf函数，而局部用的是模型变换函数，包括glTranslate,glRotate,glScale等。因此全局中的多个矩阵相乘，是让矩阵和顶点坐标做左乘操作，其矩阵相乘的前后顺序和代码的顺序相反；局部中的模型变换函数，是对模型坐标系的操作，例如glRotate旋转的不是模型上某个顶点的坐标，而是模型局部坐标系，这点是必须要搞清楚的。但是你去找高手的文章，这种低级入门概念是找不到的

**照相机管理**

采用gluLookAt函数

gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, Gldouble upx, Gldouble upy, Gldouble upz)

这个函数接受观察点的位置、观察者前面直接位置上的一个点以及向上向量的方向

该函数定义了一个视图矩阵，并与当前矩阵相乘

//设置视口

glViewport(0, 0, width, height);

//重置坐标体系

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

//设置裁剪区域

**gluPerspective(145.0f, (GLfloat)width / (GLfloat)height, 25.0f, 425.0f);**

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

绘图函数为：

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);

glLoadIdentity();

**gluLookAt(0.0f, 0.0f, 0.0f, 0.0f, 0.0f, -1.0f, 1.0f, 1.0f, 0.0f)**;

glBegin(GL\_QUADS);

for(int i = 0; i < 15; i++)

{

glVertex3f(-50.0f, 50.0f, -25.0f \* (i + 1));

glVertex3f(-50.0f, -50.0f, -25.0f \* (i + 1));

glVertex3f(50.0f, -50.0f, -25.0f \* (i + 1));

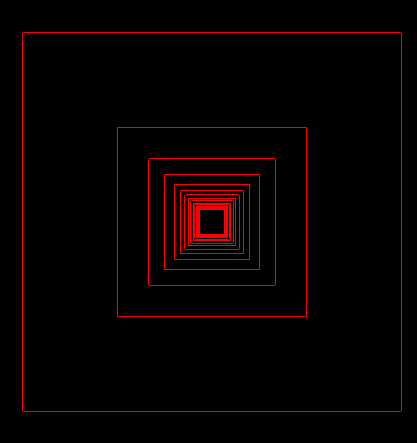
glVertex3f(50.0f, 50.0f, -25.0f \* (i + 1));

}

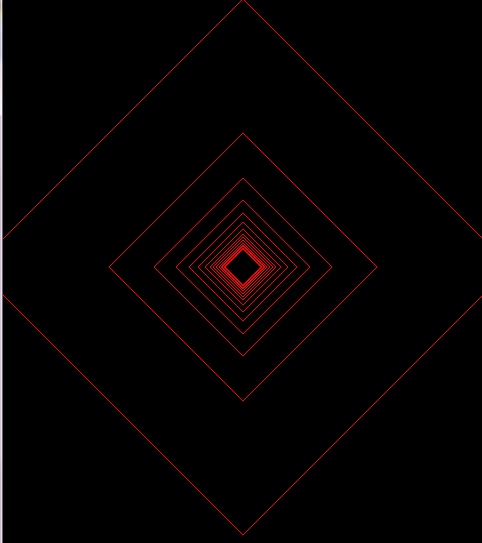
glEnd();

在上面的例子中有以下几种情况：

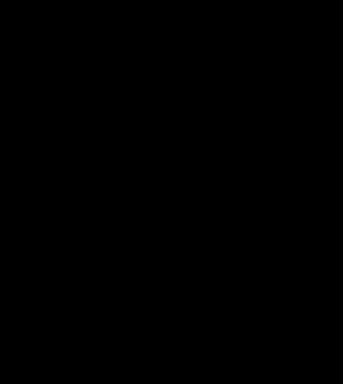
1. **gluLookAt(0.0f, 0.0f, 0.0f, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f, 0.0f);**



1. **gluLookAt(0.0f, 0.0f, 0.0f, 0.0f, 0.0f, -1.0f, 1.0f, 1.0f, 0.0f);**



1. **gluLookAt(0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, 0.0f);**



gluLookAt的调用应该在场景绘制初glLoadIdentity函数调用之后，在所有的glTrandlate、glRorate、glScale函数调用之前调用，以确保首先作用于物体的是模型变换

**openGL中的图像**

在位图中，一块内存中的每一位正好对应于屏幕上一个像素的状态

GLubyte rasters[24] = {

0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,

0xff, 0x00, 0xff, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,

0xff, 0xc0, 0xff, 0xc0

};

void SetRC()

{

glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

glColor3f(1.0f, 0.0f, 0.0f);

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1);

}

void RenderScene()

{

glClear(GL\_COLOR\_BUFFER\_BIT);

glRasterPos2i(20, 20);

glBitmap(10, 12, 0, 0, 11, 0, rasters);

glBitmap(10, 12, 0-, 0, 11, 0, rasters);

glBitmap(10, 12, 0, 0, 11, 0, rasters);

glutSwapBuffers();

}

下图显示的是一个200 \* 200的窗口显示结果



**组成位图的位是从左下角开始绘制的**：首先绘制的是最底部一行，然后是倒数第二行，接下来以此类推，最后绘制第一行。注意F字符可见部分的最大宽度是10位，位图数据总是成块存储的（8位的倍数），但实际位图的最宽部分并不一定是8的倍数

当前光栅位置就是开始绘制下一幅位图（或图像）的屏幕位置，采用glRasterPos或者glWindowPos调用，其中glWindowPos为glRasterPos替代函数，glWindowsPos用窗口坐标指定当前光栅坐标，而不必把它的x和y坐标通过模型视图和投影矩阵进行变换，也不会被裁剪出视口区域，glWindowsPos使我们更容易混合使用2D文字和3D图形，而不必在各种变换状态之间反复切换，为opengl1.4版本引用

注意：位图字体无法进行旋转，因为位图在绘制时总是与帧缓冲区的x和y轴对齐，另外，位图也无法进行缩放

设置位图颜色时候，如果改为下面的这段代码：

void RenderScene()

{

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(0.0f, 1.0f, 0.0f); // 设置为绿色

glRasterPos2i(20, 20);

glColor3f(0.0f, 0.0f, 1.0f); // 设置为蓝色

glBitmap(10, 12, 0, 0, 11, 0, rasters);

glBitmap(10, 12, 0-, 0, 11, 0, rasters);

glBitmap(10, 12, 0, 0, 11, 0, rasters);

glutSwapBuffers();

}



最终结果显示为绿色，，也就是说光栅颜色状态变量是在调用glRasterPos时根据当前颜色设置的，在这段代码中，当程序调用glRasterPos时，GLL\_CURRENT\_RASTER\_COLOR被设置为绿色，第二个glColor3f调用修改了GL\_CURRENT\_COLOR的值，用于以后的几何图形渲染，但用于渲染位图的颜色并没有发生变化

**注意：位图的颜色是在glRasterPos或glWindowsPos被调用时设置的，这意味着先前glColor所设置的当前颜色将作用于后面的位图操作中，但是，在光栅位置已经设置后，再调用glColor不会对颜色产生影响**

位图和像素图在内存中很少是以紧密包装的形式存在的，在许多情况下，为了存取效率都会按字节对齐。在默认情况下，opengl采用4个字节对齐，但是我们可以使用下面的函数修改位图或像素图存储和提取像素的方式：

Void glPixelStorei(GLenum pname, Glint param);

Void glPixelStoref(GLenum pname, Glfloat param);

例如，如果希望改为紧密包装的像素数据，可以进行下面这个函数的调用：

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1);

GL\_UNPACK\_ALIGNMENT指定了opengl如何从数据缓冲区中对图像进行解包，类似地可以使用GL\_PACK\_ALIGNMENT来告诉如何从颜色缓冲区读取的数据进行包装，并把它放置在一个用户指定的内存缓冲区中。

与位图相比，像素图更为有趣，像素图在内存中的布局与位图相似，但是，它的每一个像素可以由超过1个位的存储空间来表示，每个像素所多出来的这些位可以存储这个像素的强度（有时候称为亮度值）或颜色成分。和位图一样，我们也是在当前的光栅位置上绘制像素图，但需要使用glDrawPixels来绘制它，位图的绘制需要glBitmap函数

void PixelTest::render()

{

cout << "PixelTest render()" << endl;

// 按一个字节对齐

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1);

// 加载tga文件，获取图片宽度、高度和成份/格式信息

GLint width, height, component;

GLenum format;

GLbyte \*image = gltLoadTGA("fire.tga", &width, &height, &component, &format);

// 使用窗口坐标设置光栅位置，投影映射空间为(-200, 200, -200, 200, -200, 200)

glRasterPos2i(-200, -200);

// 绘制像素图

if(image == NULL) return;

glDrawPixels(width, height, format, GL\_UNSIGNED\_BYTE, image);

free(image);

}



注意：glRasterPos2i所指定的点的位置和glVertex2f一样，要经过变换

同时我们还可以从颜色缓冲区读取像素数据，甚至把数据从颜色缓冲区的一部分赋值到另一部分，这个用来读取像素数据的函数与glDrawPixels非常相似，只是它的逆操作而已。

glReadPixels(Glint x, Glint y, GLsizei width, GLsizei height, GLenum format, GLenum type, ocnst \*pixels)

x与y为左下角的坐标，format和type参数是所需的数据格式和类型，如果颜色缓冲区所存储的数据与请求的数据不同，opengl会负责进行必要的转换。另外，如果指定了位于边界之外的窗口坐标，则只能获取位于实际opengl帧缓冲区的像素。同时如果颜色缓冲区错存储的数据与请求的数据不同，opengl会负责进行必要的转换

将像素从颜色缓冲区的一部分复制到另一部分也非常容易，并且在这个操作过程中不需要分配任何临时存储空间。首先，使用glRasterPos或glWindowsPos函数设置希望图像数据被复制的目标角落（记住是左下角），然后使用下面这个函数进行复制：

Void glCopyPixels(GLint x, GLint y, GLsizei width, GLsizei height, GLenum type)

在这里type参数应该是GL\_COLOR，对于这个参数也可以是GL\_DEPTH或GL\_STENCIL。如果是这样，复制将分别在深度和模板缓冲区中进行。

void PixelTest::render()

{

cout << "PixelTest render()" << endl;

// 按一个字节对齐

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1);

// 加载tga文件，获取图片宽度、高度和成份/格式信息

GLint width, height, component;

GLenum format;

GLbyte \*image = gltLoadTGA("fire.tga", &width, &height, &component, &format);

// 使用窗口坐标设置光栅位置，投影映射空间为(-200, 200, -200, 200, -200, 200)

glRasterPos2i(-200, -200);

// 绘制像素图

if(image == NULL) return;

glDrawPixels(width, height, format, GL\_UNSIGNED\_BYTE, image);

free(image);

**GLbyte \*pixels = new GLbyte[width \* height \* 24];**

**glRasterPos2i(20, 20);**

**glReadPixels(100, 100, width / 2, height / 2, format, GL\_BYTE, pixels);**

**glDrawPixels(width / 2, height / 2, format, GL\_BYTE, pixels);**

**delete[] pixels;**

}



在默认情况下，在双缓冲区渲染环境中，所有这些像素操作都是在后缓冲区进行的；在单缓冲区渲染环境中，它们是在前缓冲区进行的。我们可以使用下面这两个函数修改这些像素操作的源或目标。

Void glDrawBuffer(GLenum mode);

Void glReadBuffer(GLenum mode);

glDrawBuffer函数会影响glDrawPixels或glCopyPixels操作中像素的绘制地点，可以使用任何有效缓冲区常量：GL\_NONE、GL\_FRONT、GL\_BACK、GL\_FRONT\_AND\_BACK、GL\_FRONT\_LEFT、GL\_FRONT\_RIGHT等

glReadBuffer函数接受相同的常量，并对作为glReadPixels或glCopyPixels函数所执行的读取操作目标颜色缓冲区进行设置

像素中更有趣的操作（像素缩放、像素转移、像素映射）：

1）像素反转（使用了负数的放缩因子）

glPixelZoom(-2.0f, 1.0f);

注意：如果使用负数，不仅会反转图像的像素顺序，而且会反转像素根据光栅位置绘制在屏幕上的方向。例如，在正常情况下，图像在绘制时将放置在作为当前光栅位置的左下角，如果两个缩放因子都为负值，那么光栅位置就称为最终图像的右上角



2）只保留红色（任意一种颜色均可）

glPixelTransferf(GL\_RED\_SCALE, 1.0f);

glPixelTransferf(GL\_GREEN\_SCALE, 0.0f);

glPixelTransferf(GL\_BLUE\_SCALE, 0.0f);



新值 = （旧值 \* 缩放值） + 偏转值 （默认情况：缩放值=1.0 偏转值=0.0）

3）反转颜色（像素映射）

除了缩放和偏转操作之外，像素转移操作还支持颜色映射。颜色映射就是一张表，用于查找一种颜色值（以表索引的方式使用）可以转换为另外哪一种颜色（存储在这个索引位置的颜色值）

GLfloat invertMap[256];

for(int i = 0, size = 256; i < size; i++)

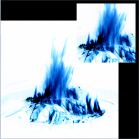
invertMap[i] = 1.0f - (1.0f / 255.0f \* (GLfloat)i);

glPixelMapfv(GL\_PIXEL\_MAP\_R\_TO\_R, 255, invertMap);

glPixelMapfv(GL\_PIXEL\_MAP\_G\_TO\_G, 255, invertMap);

glPixelMapfv(GL\_PIXEL\_MAP\_B\_TO\_B, 255, invertMap);

glPixelTransferi(GL\_MAP\_COLOR, GL\_TRUE);



为了设置颜色映射，必须调用一个函数glPixelMap

GL\_PIXEL\_MAP\_R\_TO\_R：重新映射红色成分

GL\_PIXEL\_MAP\_G\_TO\_G：重新映射绿色成分

GL\_PIXEL\_MAP\_B\_TO\_B：重新映射蓝色成分

GL\_PIXEL\_MAP\_A\_TO\_A：重新映射alpha成分

例如，假设图片中某个像素的颜色是RGB(255,127,127)，那么它映射的计算过程如下：

先把颜色从[0,255]（颜色取值范围）映射到mapsize这个大小上，也就是R=round(255/255)\*255 G=round(127/255)\*255 B=round(127/255)\*255，然后利用得到的索引值去mapArray数组中查询对应地方的颜色得到最终RGB(0.0,0.5,0.5)，最后将得到的结果进行处理绘制出来。

颜色矩阵：颜色矩阵是一种矩阵堆栈，它的工作方式类似于其他的opengl矩阵堆栈（GL\_MODELVIEW，GL\_PROJECTION，GL\_TEXTURE），我们可以调用glMatrixMode函数并使用GL\_COLOR把颜色矩阵作为当前的活动矩阵，所有的矩阵操作函数（glLoadIdentiry、glLoadMatrix等）都可以作用域颜色矩阵，颜色矩阵也看而已进行压入和弹出，但opengl实现只要求支持两个元素深度的颜色矩阵

正常模式：



调用颜色矩阵之后：

glMatrixMode(GL\_COLOR);

glScalef(2.0f, 2.0f, 2.0f);

glMatrixMode(GL\_MODELVIEW);



卷积

卷积是一种功能强大的图形处理技巧，具有许多应用，例如模糊、锐化以及其他特殊效果，是一种过滤器，它根据一些权重模式（称为卷积核）在一幅图像中进行像素处。卷积用这个像素以及邻近像素的加权平均值对它进行替换，每个像素的颜色值根据卷积核的权值进行缩放。

图像与位图相似，但是屏幕矩形区域中的每个像素并不是由1个位表示的，在图像中，每个像素可以包含更多的信息。例如，图像的每个像素可以存储完整的颜色

在正常情况下，我们想到的图像就是来自颜色缓冲区的图片，但是我们也可以从深度缓冲区和模板缓冲区读取（或写入）矩形区域的像素数据

读取、写入和复制像素数据：

glReadPixels：从帧缓冲区读取一个矩形像素数组，并把数据保存在内存中

glDrawPixels：把内存中所保存的一个矩形像素数据写入到帧缓冲区中由glRasterPos所指定的当前位置

glCopyPixels：把一个矩形像素数组从帧缓冲区的一个部分复制到另一个部分。这个函数的行为类似于在调用glReadPixels之后再调用glDrawPixels，但数据并不会写入到内存中

1）glReadPixels

glReadPixels(Glint x, Glint y, GLsizei width, GLsizei height, GLenum format, GLenum type, GLvoid\* pixels)

x，y：左下角位置

format：函数所读取的像素数据元素类型（索引值或R、G、B、A成分值）

type：每个元素的数据类型

pixels：读取的像素数据保存在pixels所指向的数组中

2）glDrawPixels

glDrawPixels (GLsizei width, GLsizei height, GLenum format, GLenum type, GLvoid\* pixels)

void makeCheckImage() // 绘制棋盘数据

{

for(int i = 0; i < checkImageHeight; i++)

{

for(int j = 0;j < checkImageWidth; j++)

{

int c = (((i & 0x8) == 0) ^ ((j & 0x8) == 0)) \* 255;

checkImage[i][j][0] = c;

checkImage[i][j][1] = c;

checkImage[i][j][2] = c;

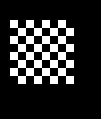
}

}

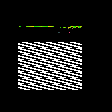
}

glRasterPos2i(0, 0);

glDrawPixels(checkImageWidth, checkImageHeight, GL\_RGB, GL\_UNSIGNED\_BYTE, checkImage);



如果我们将上面的改为GL\_RGBA得到的结果为：



3）glCopyPixels

glCopyPixels(Glint x, Glint y, GLsizei width, GLsizei height, GLenum buffer)

在帧缓冲区内部进行赋值像素数据，从帧缓冲区中的一个矩形（左下角为x与y，宽高分别为width与height）复制像素数据。数据被复制到帧缓冲区中的一个新位置，它的左下角就是当前光栅位置。Buffer是GL\_COLOR、GL\_STENCIL或GL\_DEPTH，指定了这个函数所使用的帧缓冲区。

注意：glCopyPixels函数并不需要format或data数据，因为数据绝不会复制到内存中，glCopyPixels中用于读取的源缓冲区以及目标缓冲区分别是由glReadBuffer函数和glDrawBuffer函数指定的

**OpenGL中的图像管线与坐标变换**

<http://blog.sina.com.cn/s/blog_4bce4aa301011ebe.html>

当glDrawPixels函数被调用时，数据首先根据当前生效的像素存储模式从内存进行解包，接着执行像素传输操作。然后，最终的像素进行光栅化。在光栅化阶段，像素矩形可能会根据当前的状态进行放大或缩小。最后应用的是片段操作，像素被写入到帧缓冲区

当glReadPixels函数被调用时。数据从帧缓冲区读回，并执行像素传输操作，最终的数据被包装到处理器内存中



当glCopyPixels函数指定所有的像素传输操作（相当于glReadPixels函数所执行的操作），然后像glReadPixels函数一样写入最终图像，但是不需要进行第二次传输，下面显示了glCopyPixels函数如何在帧缓冲区内部移动数据





从上面我们可以看出渲染位图要比渲染图像简单一些，因为前者不需要执行像素传输操作和像素缩放操作

注意：像素存储模式和像素传输操作对纹理的应用是在从纹理内存读取纹理或者吧纹理写入到纹理内存时进行的

**像素的包装和解包**

包装和解包是指像素数据写入到处理器内存以及从处理器内存读取的方式

当像素存储在内存中时，每个像素有1至4块数据（称为元素）组成。像素数据有可能只包括颜色索引或亮度（亮度是由红、绿、蓝值的加权之和），也可能包括了每个像素的红、绿、蓝和alpha成分。像素数据可能出现的排列（或称为格式）决定了每个像素所存储的的元素数量以及它们的存储顺序

**控制像素存储模式**

openGL所支持的所有像素存储模式都是由glPixelStore函数控制的

void glPixelStore(GLenum pname, TYPE param)

设置像素存储模式。这些模式将影响glDrawPixels、glReadPixels、glBitmap、glPolygonStipple、glTexImage1D、glTexImage2D、glTexImage3D、glTexSubImage1D、glTexSubImage2D、glTexSubImage3D、glGetTexImage函数所执行的处理

**像素传输操作**

在像素传输期间所执行的转换称为像素传输操作，它们是由glPixelTransfer和glPixelMap函数控制的

**放大、缩小或反转图像**

在应用了像素存储模式和像素传输操作后，图像和位图就进行光栅化。正常情况下，图像中的每个像素被写入到屏幕中的一个像素。但是我们可以使用glPixelZoom函数对图像进行任意的放大、缩小甚至翻转（反射）

Void glPixelZoom(GLfloat zoomX, GLfloat zoomY)

默认情况下zoomX雨zoomY均为1.0，如果他们是2.0，图像的每个像素被绘制成4个屏幕像素，缩放因子也可以为负数，如果为负数则根据当前的光栅位置对图像进行翻转

**Opengl中的帧缓冲区**

在光栅化阶段（包括纹理和雾）之后，数据就不再是像素，而成为片元，每个片元都具有与像素对应的坐标数据以及颜色值和深度值。当每个像素的数据是按照统一的方式存储时，存储所有像素的存储空间就称为缓冲区，不同缓冲区为每个像素所存储的数据量可能不同，但是，在一个特定的缓冲区内，为每个像素所存储的数据量是相同的。为每个像素存储了1位信息的缓冲区又称为位平面

颜色缓冲区只是用于存储像素信息的许多缓冲区之一，其他还有很多缓冲区，颜色缓冲区本身也可能包括几个子缓冲区，系统的帧缓冲区是由所有这些缓冲区组成的，除了颜色缓冲区之外，其他缓冲区都无法直接查看

为什么要分为多个缓冲区？

（个人理解）每个像素的数据量不一定相同，但是对于一个特定的缓冲区而言，它为屏幕上每个像素所存储的数据量是相同的，这种方式显得更加灵活

**颜色缓冲区：**

就是通常用于绘图的缓冲区，它们包括了颜色索引或RGB颜色数据，还可能包括alpha数据，支持立体画面的OpenGL实现提供了左、右颜色缓冲区，分别包含了左、右立体图像，如果opengl实现不支持立体图像，那就只使用左缓冲区，类似，双缓冲系统提供了前、后缓冲区，而单缓冲系统只提供了前缓冲区，每个opengl实现都必须提供一个前-左缓冲区

另外，openGL实现还可能支持不可显示的辅助颜色缓冲区，openGL并没有指定这类缓冲区的特定用途，因此可以按照自己的想法定义和使用它们，例如，我们可以使用它们保存一幅想要重复使用的图像，这样，我们就不必对这幅图像进行重绘，而是只需要把它冲辅助缓冲区复制到普通颜色缓冲区就可以了

我们可以再glGetBooleanv函数中使用GL\_STEREO或GL\_DOUBLEBUFFER参数来判断自己所使用的系统是否支持立体画面（即提供左右缓冲区）或双缓冲（即提供前后缓冲区），为了判断系统支持多少个辅助缓冲区（如果有的话），可以使用GL\_AUX\_BUFFERS为参数条用glGetIntegerv函数

GLboolean b;

glGetBooleanv(GL\_STEREO, &b);

if(b)

{

cout << "support sterro: true" << endl;

}

else

{

cout << "support sterro: false" << endl;

}

glGetBooleanv(GL\_DOUBLEBUFFER, &b);

if(b)

{

cout << "support double buffer: true" << endl;

}

else

{

cout << "support double buffer: false" << endl;

}

GLint bufferCount;

glGetIntegerv(GL\_AUX\_BUFFERS, &bufferCount);

cout << "support buffers count: " << bufferCount << endl;



**清除缓冲区：**

清除缓冲区是开销最大的操作之一，为了解决这个问题，有些计算机提供了能够同时清除多个缓冲区的硬件，openGL充分利用了这类硬件的优势。首先，指定需要写入到每个将要被清除的缓冲区的值，然后就可以发布一条命令执行清除操作。在这个命令中，需要传递将要清除的缓冲区列表，如果硬件支持同时清除多个缓冲区，这些清除操作就会同时进行，否则，这些缓冲区就会按顺序逐个清除

glClearColor glClearIndex glClearDepth glClearStencil glClearAccum

指定RGBA模式下的颜色缓冲区，颜色索引模式下的颜色缓冲区，深度缓冲区，模板缓冲区和累积缓冲区的当前清除值

**选择用于读取和写入的颜色缓冲区**

绘图和读取操作的目标都可以是任何颜色缓冲区，我们可以选择一个单独的缓冲区作为绘图或读取操作的目标，对于绘图操作，还可以通过设置把绘图结果同时写入到多个缓冲区。我们可以使用glDrawBuffer函数选择需要写入的缓冲区，也可以使用glReadBuffer函数选择用于读取的缓冲区，作为glReadPixels、glCopyPixels、glCopyTexImage和glCopyTexSubImage的数据来源

**多重采样**

抗锯齿处理的最大优点之一就是它能够使多边形的边缘更为平滑，使渲染效果显得更为自然和逼真。点和直线的平滑处理是得到广泛支持的，但遗憾的是多边形的平滑处理并没有在所有平台上都得到实现。即使在可以使用GL\_POLYGON\_SMOOTH的时候，对整个场景进行抗锯齿处理并没有想象中的那么方便。这是因为抗锯齿处理是基于混合操作的，这就需要从前导后对所有的图元进行排序，这是非常麻烦的。

Opengl新增了一个特性，称为多重采样，可以用来解决这个问题。如果读者所使用的opengl实现支持这个特性（这个是在opengl1.3的特性），已经包含了颜色、深度和模板值的帧缓冲区就会添加一个额外的缓冲区，所有的图元在每个像素上都进行多次采样，，其结果就存在于这个缓冲区中，每次当这个像素进行更新时，这些采样值进行解析，以产生一个单独的值。自然这种处理会带来额外的内存开销和处理器开销，有可能对性能造成影响。因此，有些opengl实现可能并不支持多渲染环境中的多重采样

为了进行多重采样，首先必须获得一个支持多重采样帧缓冲区的渲染环境

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_DEPTH | GLUT\_ACCUM | GLUT\_RGBA | **GLUT\_MULTISAMPLE**);

可以使用glEnable(GL\_MULTISAMPLE)或glDisable(GL\_MULTISAMPLE)打开或关闭多重采样

关于多重采样需要注意的一点，当它被启动时，点、直线和多边形的平滑特性都将被忽略，这就意味着在使用多重采样时，就不能同时使用点和直线的平滑处理。在一定的特定opengl实现中，点和直线如果采用平滑处理可能会比使用多重采样效果更好，因此，当我们绘制点和直线时，可以关闭多重采样，在绘制其他实心几何图形时再打开多重采样。

当然，如果没有多重采样缓冲区，opengl就当GL\_MULTISAMPLE是被禁用的

指定法线

Opengl中可以为每个多边形只指定一条法线（多边形各个顶点的法线方向相同），也可以为每个多边形指定多条法线（多边形顶点的法线方向有不同），除了顶点之外，不能为多边形其他地方分配法线

法线向量只表示方向，不表达大小（即与长度无关）。理论上可以为顶点指定任意大小的法线向量，但在opengl执行光照操作时，会将顶点的法线向量规范化（单位化），而这样必然会降低程序的性能，所以一般应由我们自己提供各个顶点的规范化法线向量

如果只是对模型进行移动、旋转操作，法线向量的长度不会发生变化，而如果对模型进行缩放操作，长度则将变化。如果需要opengl对法线向量进行规范化，就需要启动这个功能，glENable(GL\_NORMALIZE)，默认这个功能是关闭的。

如果所进行的缩放是均匀缩放，则可以使用glEnable(GL\_RECSALE\_NORMAL)，这样性能相比于GL\_NORMAL更优，默认情况下，GL\_RESCALE\_NORMAL也是被禁用的。

法线向量一旦被指定，除非再指定其他点的法线向量，否则以后指定的所有顶点都将以这一向量作为自己的法线向量。

法线均衡

早前我们曾经提到过通过对法线进行“扭曲”，可以用平面多边形产生外光平滑的表面，这种技巧称为法线均衡，它可以产生一些有趣的光学幻觉

其实主要是处理一些共享顶点，在球体中，计算真实法线比较简单，球体中心和多边形顶点的连线即是，但是在一些复杂的物体中就没那么简单了，需要取得共享一个顶点的多边形的法线，对其进行平均，来获得更加真实地效果

为什么需要单位法线？

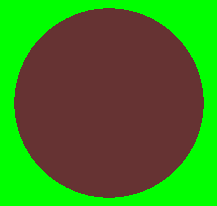
个人理解，根据phong光照模型，方便求cos值，仅仅只需要求两个向量的点积

例子：

glTranslatef(0.0f, 0.0f, -100.0f);

glScalef(3.0f, 3.0f, 3.0f);

gluLookAt(-10.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);



glTranslatef(0.0f, 0.0f, -100.0f);

glScalef(3.0f, 3.0f, 3.0f);

gluLookAt(-10.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);

**glEnable(GL\_NORMALIZE);**



求任意多边形的法向量

可以从多边形平面上的三个点来计算它的法向量，可以产生两个向量，并求出向量的叉乘

**辅助颜色**

在正常情况下，opengl执行光照计算，并根据标准光照模式计算单个片断的颜色，然后，这些片断颜色与应用到几何图形的过滤后的纹理单元相乘，但是，这个过程会产生一个副作用，它会大大削弱几何图形表面镜面亮点的可见性。一般情况下，任何纹理颜色与1（白色亮点）相乘的结果仍然是同一种纹理颜色，我们无法通过把纹理颜色与一个小于或等于1的值相乘来使它变得更亮

这个纹理的解决方案就是在纹理贴图治后再应用镜面亮点（通过加法而不是乘法），这个技巧称为辅助镜面颜色，它可以通过手动应用或者通过光照模型自动计算，我们可以通过简单地设置glLightModel打开它：

glLightMode(GL\_LIGHT\_MODEL\_COLOR\_CONTROL, GL\_SEPARATE\_SPECULAR\_COLOR);

我们可以通过把光照模型参数设置为GL\_COLOR\_SINGLE，把它切换回常规的光照模型

glLightMode(GL\_LIGHT\_MODEL\_COLOR\_CONTROL, GL\_COLOR\_SINGLE);

如果没有使用光照（光照被禁用），也可以使用glSecondaryColor函数，在纹理贴图之后直接指定一种辅助颜色。和glColor一样，这个函数也有很多变型。 我们还应该注意到，如果指定了一种辅助颜色，必须显式启用辅助颜色，这是通过启用GL\_COLOR\_SUM标志实现的：

glEnable(GL\_COLOR\_SUM)

注意：只有当光照被禁用时，才能手动设置辅助颜色

**万向节锁**

<http://blog.csdn.net/kesalin/article/details/7168967>

<http://blog.csdn.net/kesalin/article/details/2161254>

http://blog.csdn.net/huazai434/article/details/6458257

3D数学基础：图形与游戏开发

欧拉角的缺点：

1）欧拉角是不可传递的，旋转的顺序影响旋转的结果，不同的应用又可能使用不同的循环顺序，旋转顺序无法统一

2）2个旋转的角度可以不受限制，既可以是100000度，也可以是-10000度

3）可能造成万向节死锁

实际上，透视矩阵要与观察矩阵合并，然后将组合矩阵应用于场景的世界坐标描述以生成齐次坐标，在经过规范化变换和裁剪等其他处理后，用参数h去除齐次坐标可得到真实的变换坐标位置。

相机空间的一个顶点v，经过透视变换后进入了CVV中，这个变换实际上完成了两个工作：

1）将顶点从3D空间投影到2D的投影平面（Projection Plane）上

2）将投影平面上的2D投影点通过线性插值变换到齐次裁剪空间CVV中

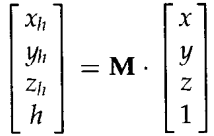
这两个变换都是通过透视矩阵一次完成的，顶点进入齐次裁剪空间并经过CVV裁剪，最终进行透视除法从4D齐次形式变回3D形式

视觉坐标系中的顶点在做投影变化后实际上得到的是一个齐次坐标系，其中的w=-z

计算机图形库将空间位置当做四维齐次坐标来处理，因此所有变换均可以用4\*4矩阵来表示，每一坐标位置进入观察流水线时，它被转换成四维表示



在一个位置经过几何、观察和投影变换后，称为一个齐次形式：



这里的M表示从世界坐标到规范化、齐次投影坐标的所有各种变换的组合，而齐次参数h可能不再是1。事实上，h可以有任意实数值，依赖于我们如何在场景中表达对象及使用的投影类型。

如果该齐次参数h确实为1，则该齐次坐标与笛卡尔坐标相同。对于平行投影变换来说常常如此。但是，透视投影生成的齐次坐标是任一空间点z坐标的函数。透视投影的齐次参数甚至可能为负值。这在坐标位置位于投影参考点之后时发生，同样，对象曲面的有条样条表示常在齐次坐标中形成，其中齐次坐标可正可负。因此，如果裁剪在投影坐标系中除以齐次参数h后再完成，则可能失去某些坐标信息而使裁剪不正确。（比如z为0的点？？？）

处理所有可能的投影变换和对象表示的一个有效方法，是在空间位置的齐次坐标系中进行裁剪，而由于所有观察体可转换到一个规范化立方体，因此可以用硬件来完成单一的参见过程，在齐次坐标系中用规范化参见平面来裁剪对象。

**纹理映射**

纹理映射是将纹理空间中的纹理像素映射到屏幕空间中的像素的过程，简单来讲，纹理就是矩形的数据数组，例如，颜色数据、亮度数据、颜色和alpha等，纹理数组中的单个值称为纹理单元，也叫纹素，这里让它区别与像素。通过uv来表述

注意：opengl中v轴从上向下是正方向，u轴从左向右是正方向

与纹理映射有关的一个特性是，当模型进行变换时，纹理坐标仍然会跟着模型的顶点，它们并不进行变换（当然也有其他方法可以改变纹理坐标），就好像粘着顶点一样。这样，当模型进行旋转、拉伸和放缩时，纹理也会跟着变化

1）加载纹理

要把纹理映射到几何图形中，首先我们需要加载纹理到内存中，加载之后这个纹理就称为opengl当前纹理状态的一部分。Opengl提供了下面三个方法从内存缓冲区中加载纹理：

Void glTexImage1D() Void glTexImage2D() Void glTexImage3D()

这三个方法告诉了opengl加载的纹理数据信息。使用一致的根函数调用（glTexImage）加载纹理，使其成为当前纹理，使用上面的函数时，opglGL会拷贝data参数所指向的位置的纹理信息。这种数据复制可能代价很高，opengl中可以使用纹理对象来缓解性能问题

2）颜色缓冲区中读取

跟从颜色缓冲区中读取像素一样，纹理数据一样可以从颜色缓冲区中读取，使用如下方法：

Void glCopyTexImage1D Void glCopyTexImage2D

注意：我们无法从2维的颜色缓冲区中读取三维的纹理数据，所以没有glCopyTexImage3D这个方法

3）更新纹理

如果我们只需要修改纹理中的一部分数据，为不想重新加载数据，可以使用glTexSubImage方法，这个方法比每次都去重新加载纹理数据要快得多

Opengl不论是旧版本还是新版本，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，可以使用下面的代码获取opengl支持的最大纹理

// 获取opengl版本支持的最大纹理尺寸

GLint openglMaxSize = 0;

glGetIntegerv(GL\_MAX\_TEXTURE\_SIZE, &openglMaxSize);

**纹理矩阵**

纹理坐标也可以通过纹理矩阵来进行变换，纹理矩阵的工作方式与投影矩阵，模型视图矩阵类似。通过glMetrixMode(GL\_TEXTURE)来开启纹理矩阵模式，在此函数调用后面的矩阵变换将被应用于纹理坐标。纹理坐标可以进行移动、放缩、旋转。纹理矩阵的栈最多只能容纳两个纹理矩阵，通过glPushMatrix和glPopMatrix来进行栈操作

**着色器内部原理（通过经典管线说明）**

着色器程序在gpu上执行，opengl主程序在cpu执行，主程序向显存输入顶点等数据，启动渲染过程，并对渲染过程进行控制，了解这一点就可以知道显示列表（Display Lists）以及glFinish这种函数存在的必要了，前者（显示列表）将一组绘制指令放到GPU上，CPU只要发一条“执行这个显示列表”这些指令就执行，而不必CPU每次渲染都发送大量指令到GPU，从而节约PCI（因为PCI总线比显存慢）；后者（glFinish）让CPU等待GPU将已发送的渲染执行执行完。

GPU提供了大规模的并行机制，特别适合于执行高度并行的渲染过程，这个“并行”的概念可能要超出我们平常在CPU上开的几十个线程，GPU的线程数可以达到上百万个或更多（每个线程可以对应于每个顶点、图元、片段的处理过程）

忽略一下高大上的新着色器，管线可以总结为：

顶点数据-顶点着色器-图元装配-几何着色器-光栅化-片段着色器-后片断处理-帧缓冲，再经过双缓冲交换，渲染内容就显示到了屏幕上

管线中的主要部分将分为 顶点处理，图元装配裁剪等（加“等”是包括几何着色器）、光栅化、片断处理四个部分。顶点处理基本对应顶点着色器，片断处理基本对应片断着色器。我们先看看常用的固定管线功能都包括哪些部分。顶点处理包括固定管线的顶点坐标变换、光照（也即逐顶点光照）等；图元装配裁剪等包括图元装配、裁剪。透视除法、视口变换等；光栅化包括点线光栅化、多边形填充、纹理、雾等；片断处理包括各种测试（Scissor、Alpha、Stencil、Depth Test）、混合（Blending）等

顶点变换：包括模型视图变换、投影变换以及光照（固定管线只能逐顶点光照）

图元装配裁剪等：顶点处理或者顶点着色器的输出是一些变换后的位于Clip坐标空间的点，这些点首先根据顶点之间的连接关系（点、线、多边形）进行图元装配，图元装配之后是裁剪，裁剪之后是透视除法，透视除法之后是视口变换，视口变换也将原来的z坐标缩放到（0-1）变为Depth，视口变换完成后的图元将进入光栅化

光栅化：到目前为止，管线里的数据都是顶点，经过图元装配之后，哪些顶点就是一个点、哪两个顶点是直线段、哪三个或更多顶点是一个三角形或多边形，这些图元都已经知道了，但它们也还只是顶点而已，顶点都还没有“像素点”、直线段之间还是空的、多边形内部也是空的，光栅是的任务就是构造这些。由于已经经过了视口变换，光栅化是在二维（带深度值）的屏幕坐标中进行的

光栅化两个任务：1 确定图元包含哪些由整数坐标确定的“小方块”（和屏幕像素对应，现在还不能叫片断，光栅化完成之后才能叫片断） 2 确定这些小方块的Depth和Color（从图元顶点的Depth和Color插值获取），这些颜色后来可能被其他如纹理操作修改，最为复杂的纹理在光删化阶段进行

片断处理：光栅化之后是一些片断，片断被称为“准像素”，要能想象出屏幕坐标系的一个整数坐标上只有一个像素，但可以前后“堆叠”多个片断。这些片断进入片断处理或者片断着色器，进行各种测试，包括Scissor、Alpha、Depth。Stencil等，通过测试的片断将被写入FrameBuffer，包括RGBA缓冲、Depth缓冲、Stencil缓冲。还有一个叫做Accumulation Buffer，多用于运动模糊。景深模糊等，但是不能直接写入，而是将RGBA缓冲整体累积，最终在屏幕上显示出来。

在opengl中，除非另有所指，否则每次条用一个绘图函数时，指定的物体就会被绘制。这看起来是显而易见的，但是在有些系统中，首先要列出需要绘制的物体清单，在完成这个清单后，再告诉图形硬件绘制这个清单中的所有物体。第一种风格称为立即模式的图形编程，它也是默认的OpglGL风格。除了立即模式之外，还可以选择把一些绘图命令保存在一个列表（称为显示列表）中，以后再一起执行。一般而言，立即模式更易编程，但显示列表常常具有更高的效率。

特殊的清除窗口函数效率远远高于普通的绘图函数

glClear的唯一参数表示需要清除的缓冲区，例如:

glClear(GL\_COLOR\_BUFFER\_BIT)值清除颜色缓冲区

一般情况下，只要在程序的早期设置一次清除颜色就可以了，以后可以根据需要随时清除缓冲区，opengl把当前的清除颜色作为一个状态变量，这样就不必在每次清除缓冲区时重新指定清除颜色

OpenGL允许同时清除多个缓冲区，这是因为清除通常是一种相对较慢的操作，涉及窗口中的每个像素（可能数以百万计）。有些图形硬件允许同时清除一组缓冲区。如果硬件不支持同时清除多个缓冲区，它就会线性地执行这些清除操作，下面这段代码：

glClear(GL\_COLOR\_BUFFER\_BIT) | GL\_DEPTH\_BUFFER\_BIT)

和

glClear(GL\_COLOR\_BUFFER\_BIT)

glClear(GL\_DEPTH\_BUFFER\_BIT)

它们在功能上是等价的，但在许多机器上，前者的执行速度要快得多，在任何机器上，它肯定不会比后者慢

glFlush -- 强制以前发出的opengl命令开始执行，因此保证它们能够在有限的时间内完成

glFinish -- 强制以前发出的opengl命令完成执行，在以前的命令完成执行之前，这个函数并不会返回

opengl多边形必须是凸多边形，也就是不存在内陷的部分。Opengl在合法多边形构成方面施加这些限制的原因是，这些限制有利于生产商提供快速的多边形渲染硬件来渲染符合条件的多边形。简单多边形的渲染速度非常快，而那些困难的情况就难以快速检测。

**标记多边形的边界边**

openGL只能渲染凸多边形，但是实际中可以看到很多非凸多边形。为了绘制这些非凸多边形，一般将他们分解为几个凸多边形（通常是三角形），然后再分别绘制这些三角形。遗憾的是，如果把一个普通的多边形分解为几个三角形，然后再分别绘制这些三角形，就无法使用glPolygonMode函数绘制多边形的真正轮廓，我们所看到的是它内部的这些三角形轮廓，为了解决这个问题可以调用glEdgeFlag(GLboolean flag)来进行顶点标记。该函数只作用于那些为多变形、三角形和四边形所指定的顶点，对哪些为三角形带或者四边形带所指定的顶点无效。如果参数为GL\_TRUE，在此之后创建的所有顶点都认为是边界边的起点，知道用GL\_FALSE为flag参数的值再次调用了这个函数

glEdgeFlag()函数表示一个顶点是否应该被认为是多边形的一条边界变的起点，默认值为GL\_TRUE

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);

glBegin(GL\_POLYGON);

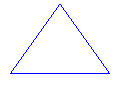
glColor3f(0.0f, 0.0f, 1.0f);

glVertex3f(300.0f, 100.0f, 0.0f);

glVertex3f(400.0f, 100.0f, 0.0f);

glVertex3f(350.0f, 170.0f, 0.0f);

glEnd();



glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);

glBegin(GL\_POLYGON);

glColor3f(0.0f, 0.0f, 1.0f);

glVertex3f(300.0f, 100.0f, 0.0f);

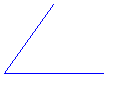
**glEdgeFlag(GL\_FALSE);** // 将以400为起点的线设置为非起点

glVertex3f(400.0f, 100.0f, 0.0f);

**glEdgeFlag(GL\_TRUE);**

glVertex3f(350.0f, 170.0f, 0.0f);

glEnd();



**法线向量**

法线向量是一条垂直于某个表面的方向向量，对于平表面而言，它上面每个点的垂直方向都是相同的，但是，对于普通曲面而言，表面上每个点的法线方向可能各不相同。在opengl中，既可以为每个多边形指定一条法线，也可以为多边形的每个顶点分别指定一条法线。同一个多边形的顶点可能共享同一条法线（平表面），也可能具有不同的法线（曲面）。除了顶点之外，不能为多边形的其他地方分配法线。

顶点属性就是顶点shader里面声明的一个attribute变量，有两种类型attribute变量。一种是const属性，一种是顶点属性数组。

1）如果是const顶点属性，表示所有的顶点的这个属性都是一样的，使用函数：

glVertexAttribnf(GLuint index, GLfloat x, GLfloat y)

glVertexAttribnfv(GLuint index, GLfloat\* data)

它只支持float和vec4类型

2）如果是顶点属性数组，使用函数：

glEnableVertexAttribArray(顶点属性的location)

glVertexAttribPointer这个函数制定顶点属性的位置、数据指针等

在opengl设计中，必须提供不少于16个vec4的变量来存储属性

**投影纹理映射**

http://www.cnblogs.com/mazhenyu/p/4088983.html

**批处理**

Opengl被描述为一种软件接口，因此，我们可以想象opengl命令可以由驱动程序以某种方式转换为一些特定的硬件命令或操作，然后发送到图形卡并立即执行，这种想法在绝大多数情况下是正确的，事实上，绝大多数opengl渲染命令会转换为一些硬件特定的命令，但这些命令并不是立即发送到硬件，反之，它们累积在一个本地缓冲区中，在达到某个门槛值后，才被一起刷新到硬件。**使用这种类型的安排主要是与图形硬件的通信需要相当长的时间，把大量数据通过总线一次性地传输到硬件显然比分批次传输要快得多**

有3个事件可以出发当前一组渲染命令的刷新。**第一个事件是驱动程序的命令缓冲区被填满时**，我们不能访问这个缓冲区，也不能对缓冲区大小施加任何控制。**另外，在执行缓冲区交换时也会触发命令缓冲区的刷新**，在所有未处理命令被执行前，缓冲区交换是不会发生的，，因此要启动刷新操作，然后执行缓冲区交换的命令，单缓冲可以采用glFlush，

但是，有些opengl命令并不会进行缓冲以便以后执行，例如，glReadPixels和glDrawPixels函数，这些函数直接访问帧缓冲区并直接读取和写入数据，这些函数实际上引入了一个管线堆，因为当前队列中的命令必须被刷新，并在我们直接更改颜色缓冲区之前执行，**我们可以调用glFinish来强制刷新命令缓冲区**，并等待图形硬件完成它的所有渲染任务。glFinish一般很少使用，一般情况下，它是为了满足平台特定的需求，例如多线程和多环境渲染

**glFlush与glFinish的区别：**

glFlush将gl命令队列中的命令发送给显卡并清空队列，发送完立即返回

glFinish将gl命令队列中的命令发送给显卡并清空队列，显卡完成这些命令后返回