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);

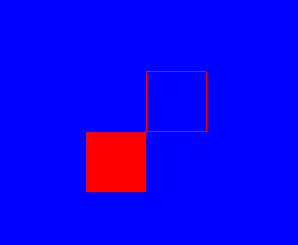
glVertex2f(0.0f, 0.0f);

glVertex2f(0.0f, 25.0f);

glVertex2f(25.0f, 25.0f);

glVertex2f(25.0f, 0.0f);

glEnd();



**剔除多边形表面：**

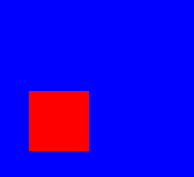
首先要使用glEnable(GL\_CULL\_FACE)启动剔除功能，然后使用glCullFace来进行剔除，glCullFace的参数可以是GL\_FRONT、GL\_BACK或者GL\_FRONT\_AND\_BACK

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

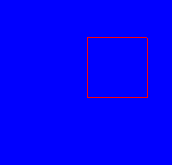
对上面的代码添加：

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,

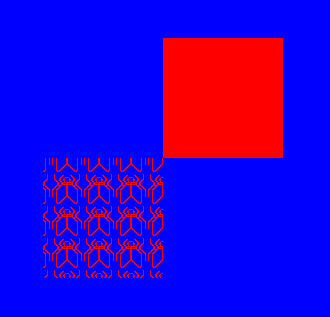
0x10, 0x30, 0x0C, 0x08,

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;

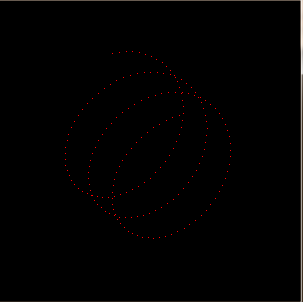
}

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);

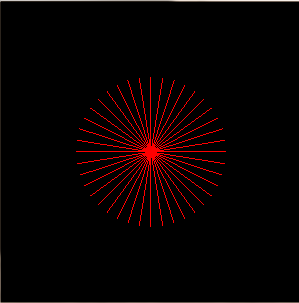
glVertex2f(x, y);

}

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;

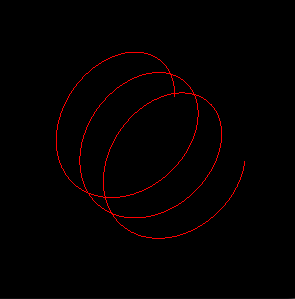
}

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);

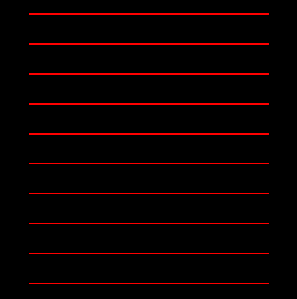
glEnd();

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);

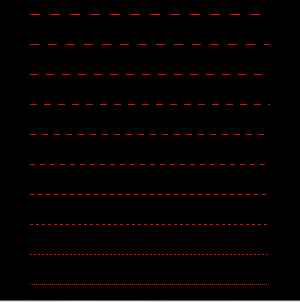
glEnd();

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);

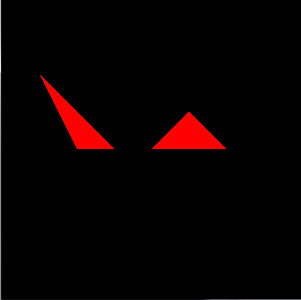
glVertex2f(50.0f, 0.0f);

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);

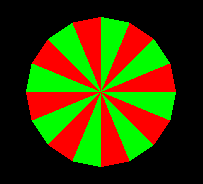
ipvort++;

}

glEnd();

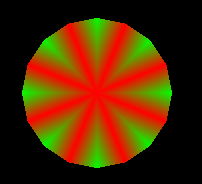
glutSwapBuffers();

}



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);

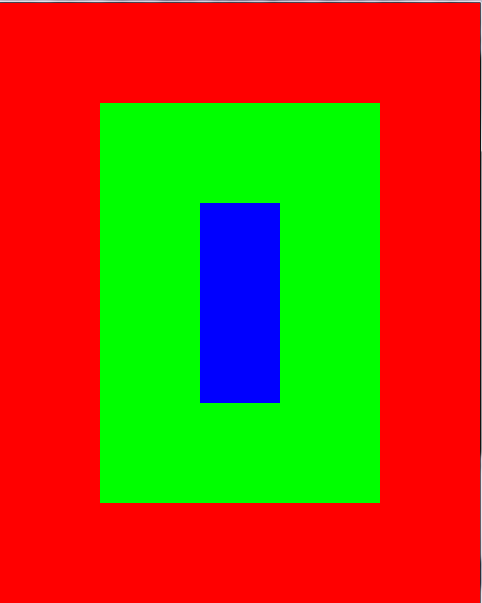
glScissor(200, 200, 80, 400);

glClear(GL\_COLOR\_BUFFER\_BIT);

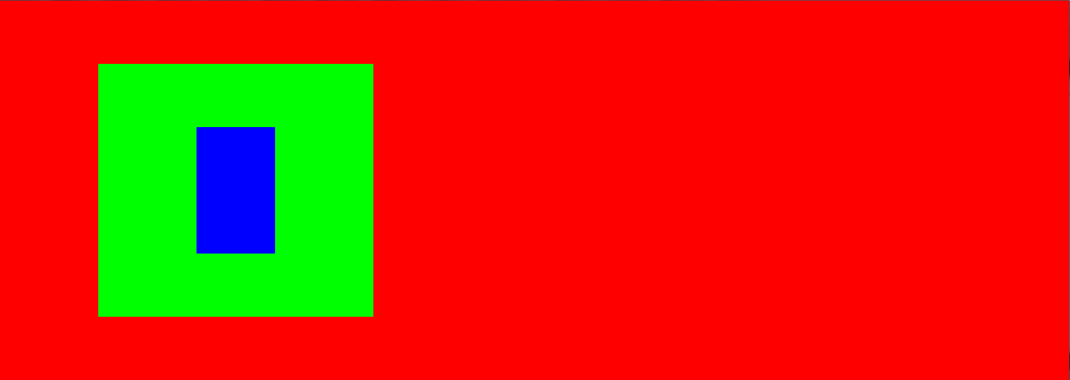
glDisable(GL\_SCISSOR\_TEST);

glutSwapBuffers();

}



实际上当我们改变实际窗口大小时候，该区域仍然不变



**模板测试**

为了使用模板测试，我们必须首先使用平台特定的OpenGL设置过程请求一个模板缓冲区，当我们使用GLUT时，我们在初始化显示模式时请求一个模板缓冲区，例如下面的代码：

glutInitDisplayMode(GLUT\_RGB | GLUT\_DOUBLE | GLUT\_STENCIL);

我们可以采用glEnable(GL\_STENCIL\_TEST)启动模板测试

打开了模板测试之后，绘图就只在那些通过了模板测试的位置进行，我们可以使用下面的函数进行模板测试：

Void glStencilFunc(GLenum func, Glint ref, GLuint mask)

模板函数的func可以取下面的任意一个值：GL\_NEVER、GL\_ALWAYS、GL\_LESS、GL\_LEUUAL、GL\_EQUAL、GL\_GEQUAL和GL\_NOTEQUAL，这些值告诉OpenGL如何把已经存储在模板缓冲分区中的值与ref参数所指定的值进行比较，另外我们可以指定一个掩码，在比较之前，将这个掩码与参考值和摹本缓冲区中的值进行位AND操作

**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();

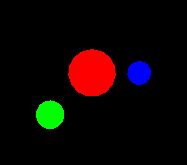
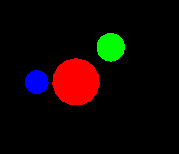
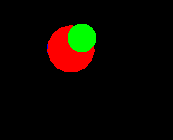
angle += 5.0f;

if(angle >= 360.0f)

angle = 0.0f;

glutSwapBuffers();

}

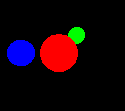
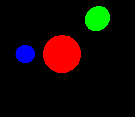
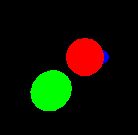
  

**透视投影**

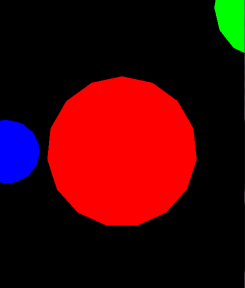
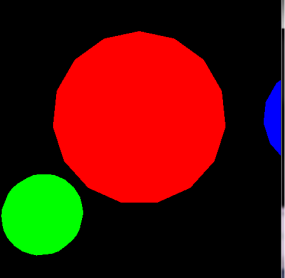
透视投影执行透视除法，对距离观察者较远的物体进行缩短和收缩。在投影到屏幕之后，可视区域后端和前端的度量单位并不相同，因此，如果两个逻辑大小相同的物体分别位于可视区域的前面和后面，前者看上去要比后者大

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));

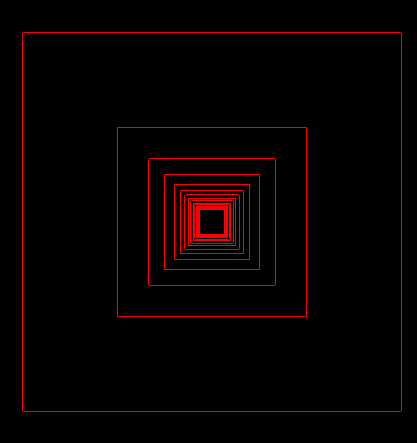
glVertex3f(50.0f, 50.0f, -25.0f \* (i + 1));

}

glEnd();

在上面的例子中有以下几种情况：

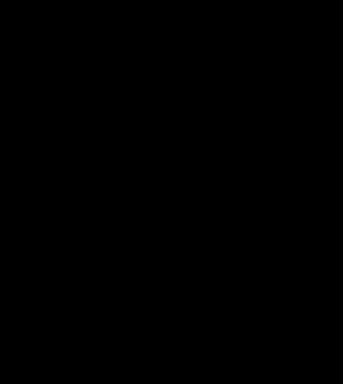
1. **gluLookAt(0.0f, 0.0f, 0.0f, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f, 0.0f);**



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坐标系之间的变换：**

本地坐标系—>世界坐标系->眼坐标系->裁剪坐标系

一般gluLookAt用于从世界坐标系到眼坐标系的转换，但是由于OpenGL里面模型视图矩阵直接将本地坐标系转换为眼坐标系，所以gluLookAt应该被用来设置模型视图矩阵

在openGL中进行几何变换的方式为，首先通过glTranslatef、glRoratef、glScalef等函数设置好几何变换矩阵（相当于对坐标系进行了变换），然后进行绘制，那么图形的投影坐标将受到设置好的几何变换矩阵所影响而显示出几何变换的效果，而不不是首先进行绘图然后再通过几何变换函数对已经存在的图形进行变换

在OpenGL中，顶点计算方法：

**屏幕坐标点 = 3D模型点 \* 几何变换栈矩阵(n…1) \* 投影变换栈矩阵(n…1)**

openGL坐标系可以分为：世界坐标系和当前绘图坐标系

**世界坐标系：**以屏幕中心为原点，你面对屏幕，你的右边是x正轴，上面是y正轴，上面是正轴，屏幕指向你的为z正轴

**当前绘图坐标系：**是绘制物体时的坐标系，程序刚初始化时，世界坐标系和当前绘图坐标系是重合的，当用glTranslatef、glScakef、glRotatef对当前绘图坐标系进行评议、伸缩、旋转变换之后，世界坐标系和当前绘图坐标系不再重合，改变之后，再用过来Verte3f等绘图函数绘图时候， 都是在当前绘图坐标系进行绘图，所有的函数参数也都是相对当前绘图坐标系来讲的

**openGL纹理过滤**

1 最近点采样（GL\_NEAREST）

2 线性纹理过滤（双线性过滤）GL\_LINEAR

3 mipmap纹理过滤（三线性过滤）GL\_LINEAR\_MIPMAP\_LINEAR

4 各项异性过滤（与前面三种是不一样的）

双线性过滤：

采用2\*2的矩阵进行加权平滑，这四个点依次是本身，右边，下面与右下角

在纹理缩减到一半或者放大一倍的范围内，双线性过滤都能有非常好的精度。这也就是说，如果纹理在每个方向都有256个像素，那么将它缩减到128以下或者放大到512以上的时候，由于会丢掉过多的像素或者进行了过多的平滑处理，纹理看起来就会很差。通常在缩减过程中使用Mipmap来实现较好的性能

对于锯齿，本身是客观存在的（由于屏幕的像素点都为整数，且画线算法是一定的，也就是说我们通过一定的画线算法始终会绘制对应的那些像素点），我们只是通过某种过滤方式来减少锯齿发生的几率，即通过改变那些对应点的颜色

Mipmap中每一个层级的小图都是主图的一个特定比例的缩小细节的复制品。虽然在某些必要的视角，主图仍然会被使用，来渲染完整的细节。但是当贴图被缩小或者只需要从远距离观看时，mipmap就会转换到适当的层级。事实上，在三线性过滤（trilinear filtering）起作用时，会在两个相近的层级之间切换。

因为mipmap贴图需要被读取的像素远少于普通贴图，所以渲染的速度得到了提升。而且操作的时间减少了，因为mipmap的图片已经是做过抗锯齿处理的，从而减少了实时渲染的负担。放大和缩小也因为mipmap而变得更有效率。+

如果贴图的基本尺寸是256x256像素的话,它mipmap就会有8个层级。每个层级是上一层级的四分之一的大小，依次层级大小就是：128x128;64x64;32x32;16x16;8x8;4x4;2x2;1x1(一个像素)。例如在一个场景中，渲染贴图需要填满的空间大小是40x40像素的话，如果没有三线性过滤，那32x32 会被放大显示，或者有三线性过滤，会在64x64和32x32之间切换。最简单的生成贴图的方法就是依次做平均，当然也可以用更加高级的算法。

三线性过滤相对的比较复杂，它只能用于纹理被缩小的情况，需要先构造纹理图像的mipmap，mip的意思是“在狭窄的地方里的许多东西”，mipmap就是对最初的纹理图像构造的一系列分辨率减少并且预先过滤的纹理图。对于一个8 x 8的纹理来说需要为它构造4 x 4、2 x 2、1 x 1这三个mipmap。如果正方形被缩小到在屏幕上占6 x 6的象素矩阵，一个象素的采样过程就变成这样，首先是到8 x 8的纹理图中进行对最接近它2 x 2的纹理单元矩阵进行采样（也就是上面的线性过滤）；其次是到4 x 4的纹理图中重复上面的过程；接着把上面两次采样的结果进行加权平均，得到最后的采样数据。可以看出整个过程一共进行了三次的线性过滤，所以这种方法叫做三线性过滤，它的效果是三种纹理过滤方法里面最好的。  
这是一种更复杂材质影像插补处理方式，会用到相当多的材质影像，而每张的大小恰好会是另一张的四分之一。例如有一张材质影像是512×512个图素，第二张就会是256×256个图素，第三张就会是128×128个图素等等，总之最小的一张是1×1.凭借这些多重解析度的材质影像，当遇到景深极大的场景时（如飞行模拟），就能提供高品质的贴图效果。一个“双线过滤”需要三次混合，而“三线过滤”就得作七次混合处理，所以每个像素就需要多用21/3倍以上的计算时间。还需要两倍大的存储器时钟带宽。但是“三线过滤”可以提供最高的贴图品质，会去除材质的“闪烁”效果。对于需要动态物体或景深很大的场景应用方面而言，只有“三线过滤”才能提供可接受的材质品质。

**openGL矩阵推导**

<http://blog.sina.com.cn/s/blog_a401a1ea0101f1k3.html>

齐次坐标表示是计算机图形学的重要手段之一，它既能够用来明确区分向量和点，同时也更易用于进行仿射（线性）几何变换

对于任意一个向量v以及基oabc

v = v1 \* a + v2 \* b + v3 \* c

对于一个点p以及基oabc

p – o = p1 \* a + p2 \* b + p3 \* c

即p = p1 \* a + p2 \* b + p3 \* c + o

也就是说虽然都是用代数分量的形式表达向量和点，但表达一个点比一个向量需要额外的信息。如果我们写一个代数分量表达（1,4,7），就不知道它是一个向量还是一个点了

V = (a, b, c, o) \* {v1, v2, v3, 0}

P = (a, b, c, o) \* {p1, p2, p2, 1}

这样，向量和点在同一个基下就有了不同的表达：3D向量的第4个代数分量为0，而3D点的第4个代数分量是1

这样，上面的（1,4,7）如果写成（1,4,7,0）就是向量，如果写成（1,4,7,1）就表示一个点

齐次坐标与普通坐标的相互转换：

1）普通坐标🡪齐次坐标

如果(x,y,z)是个点，则变为(x,y,z,1)

如果(x,y,z)是个向量，则变为(x,y,z,0)

2）齐次坐标🡪普通坐标

如果(x,y,z,1)，则知道它是一个点，变为(x,y,z)

如果(x,y,z,0)，则为一个向量，变为(x,y,z)

**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++)

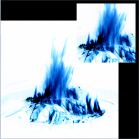
invertMap[i] = 1.0f - (1.0f / 255.0f \* (GLfloat)i);

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;

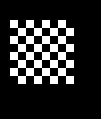
}

}

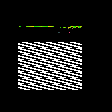
}

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中的纹理贴图**

上面的过程都是将图像直接在屏幕上画出来，但是当我们把图像数据应用到一个几何图元时，就称为纹理或纹理贴图

一幅纹理图像被加载之后，它就具有和像素图像（pixmap）相同的成分和排列，但此时纹理单元和屏幕上的像素之间很少存在一对一的对应关系。

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数据，纹理数组中的单个值常常称为纹理单元

纹理贴图步骤：

1 创建纹理对象，并为它指定一个纹理

2 确定纹理如何应用到每个像素上

3 启动纹理贴图功能

4 绘制场景，提供纹理坐标和几何图形坐标

加载纹理：

glTexImageXD：opengl支持一维、二维和三维的纹理贴图，它使用对应的函数来进行加载，并将其作为当前纹理，注意，opengl从data参数所指向的位置复制纹理信息，这种数据复制可能代价很高，在后面，我们将采用一些方法来缓解这个问题。同时必须要启动适当的纹理状态，否则纹理在加载之后不会应用到几何图形上，必须采用调用glEnable。一经加载，这个纹理就将成为当前的纹理状态

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1);

pBytes = gltLoadTGA("..\\logo.tga", &iWidth, &iHeight, &iComponents, &eFormat);

if(pBytes == NULL) cout << "load tga picture failed" << endl;

cout << "width = " << iWidth << ", height = " << iHeight << endl;

glTexImage2D(GL\_TEXTURE\_2D, 0, iComponents, iWidth, iHeight, 0, eFormat, GL\_UNSIGNED\_BYTE, pBytes); // 将pBytes中的数据复制了一份

free(pBytes);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexEnvi(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_MODULATE);

glEnable(GL\_TEXTURE\_2D); // 此句屏蔽发现纹理就不会显示

加载纹理最后一个值得注意的地方是：

通过glTexImage函数所加载的纹理数据将经理和前面所描述的像素和图像处理管线相同的过程，这意味着当纹理数据被加载时，它将会经理像素包装、像素缩放、颜色表、卷积等操作

**纹理环境**：决定了纹理单元的颜色与底层几何图形颜色进行组合

GL\_MODULATE：调整，将纹理单元颜色与几何图形颜色相乘（在光照计算之后），这就是我们仍然能够看到金字塔原先的颜色以及纹理看上去像是被着色了一样的原因，这种模式下，可以用几何图形的颜色调整纹理的色调，这样，只要一种纹理就可以实现不同的效果

GL\_REPLACE：替换，将纹理单元的颜色替换几何图形的片断颜色（即像素的颜色）

GL\_ADD：添加，简单的将纹理颜色添加到几何图形片段的颜色上，简单地吧颜色值添加到底层几何图形片段的颜色上。所有超过1.0的颜色成分都被截取为1.0，使用这种方法，可能会得到过度饱和的颜色值（基本上是一种白色或者比预想的颜色更白一些

GL\_DECAL：贴花，如果纹理没有alpha属性，和GL\_REPLACE一样，如果纹理有alpha属性，在纹理的alpha属性将会和底层几何图形的alpha成分混合在一起

GL\_BLEND：混合，如果选择这种纹理模式，还需要设置纹理环境颜色

GLfloat fColor[4] = {1.0f, 1.0f, 0.0f, 0.0f};

glTexEnvi(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_BLEND);

glTexEnvfv(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_COLOR, fColor);

**纹理过滤：**决定了纹理与像素的映射方式

<http://www.cnblogs.com/funny-world/p/3162003.html>

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

查看双线性过滤算法，在理解下面的纹理环绕

**纹理环绕：**主要是处理边缘像素点

了解下面的几种参数选择（几种参数的区别）：

<http://blog.csdn.net/lixiang996/article/details/6859575>

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

设置纹理border的颜色：

GLfloat color[4] = {1.0f, 0.0f, 0.0f, 1.0f};

glTexParameterfv(GL\_TEXTURE\_2D, GL\_TEXTURE\_BORDER\_COLOR, color);

在GL\_NEAREST过滤模式中，环绕模式并不起作用，因为纹理坐标总是对齐到纹理贴图中一些特定的纹理单元，但是，GL\_LINEAR过滤则需要取纹理坐标周围像素的平均值，对于那些位于纹理贴图边缘的纹理单元，这就会出现问题。

GL\_REPEAT：纹理采样简单地从接下来的行或列提取，在重复模式中，这相当于环绕到纹理的另一边，对于沿物体环绕并与另一边吻合的纹理（例如球体），这种模式是相当完美的,

比如，1.1处的纹理单元与0.1出的纹理单元是相同的



GL\_MIRRORED\_REPEAT：相当于将纹理坐标1.1变为0.9，达到镜面反射的效果

GL\_CLAMP：所需要的纹理单元取自纹理边界或TEXTURE\_BORDER\_COLOR（用glTexParameterfv函数进行设置），截取纹理坐标到 [0,1] 。将导致纹理坐标处于[1-1/2n, 1]的像素，在纹理滤镜为线性滤镜时，与border融合，最终纹理坐标为1的像素，将为border和边界像素的中值



GL\_CLAMP\_TO\_BORDER：截取纹理坐标到[-1/2n,1+1/2n]。将导致纹理坐标处于[1-1/2n,1+1/2n]范围内的像素，在纹理滤镜为线性滤镜时，与border融合，最终纹理坐标为1+1/2n的像素将于border同色。



GL\_CLAMP\_TO\_EDGE：截取纹理坐标到[1/2n,1-1/2n]。将导致永远不会与border融合



Mip贴图

Mip贴图主要解决两个问题：

1 闪烁问题：当屏幕上被渲染物体的表面与它所应用的纹理图像相比显得非常小时，就会出现闪烁，尤其当照相机或者物体处于运动状态时，更容易看到闪烁现象

2 性能问题：加载了大量的纹理内存并对它们进行了过滤处理，但屏幕上显示的只是很小的一部分，纹理越大，所造成的性能影响就越大

Mip贴图由一系列图像组成，每个图像的大小都是前面那个图像的一半。Mip贴图层并不一定是正方形的，但每个图像的大小都依次减半，直到最后一个图像的大小是1x1，当其中一维大小达到1时，接下来的减半处理就只发生在其他维上，使用正方形集合（即各个维的大小相等）的mip贴图所要求的内存比不使用mip贴图要多出1/3

Mipmap有多少个层级是由glTexImage的第二个参数level决定的，层级从0开始，0,1,2,3这样递增。仅仅使用glTexImage函数加载mip层并不能启动mip贴图功能，如果纹理过滤设置为GL\_NEAREST和GL\_LINEAR，则只有纹理贴图基层会被使用，也就是说，如果没有使用mipmap技术，只有第0层的纹理会被加载，在默认情况下，为了使用mipmap，所有层级都会被加载，但是我们可以通过纹理参数来控制要加载的层级范围，使用glTexParameteri，第二个参数GL\_TEXTURE\_BASE\_LEVEL来指定最低层级的level，第二个参数为GL\_TEXTURE\_MAX\_LEVEL指定最高层级的level，其他所有加载的mip测过都将被忽略，我们必须指定其中一个**mip贴图过滤器**，这样才能使用所有已加载的mip层，这个mipmap过滤器的常量是**GL\_FILTER\_MIPMAP\_SELECTOR**的形式，其中FILTER指定了过滤模式，SELECTOR指定了如何选择mipmap层，例如，GL\_NEAREST\_MIPMAP\_LINEAR，它的SELECTOR是GL\_LINEAR，它会在两个最邻近的mip层中执行线性插值，然后得出的结果又由被选择的过滤器GL\_NEAREST进行过滤

其中GL\_NEAREST\_MIPMAP\_NEAAREST具有很好的性能，也能够解决闪烁的问题，但在视觉效果上会比较差。其中GL\_LINEAR\_MIPMAP\_NEAREST常用于游戏加速，使用了质量较高的线性过滤，和快速的选择的方式(最邻近方式）。

使用最邻近的方式作为mipmap选择器的效果依然不能令人满意。从某一个角度去看，常常可以看到物体表面从一个mip层到另一个mip层的转变。GL\_LINEAR\_MIPMAP\_LINEAR和GL\_NEAREST\_MIPMAP\_LINEAR过滤器在mip层之间执行一些额外的线性插值，以消除不同层之间的变换痕迹，但也需要一些额外的性能开销。GL\_LINEAR\_MIPMAP\_LINEAR具有最高的精度。

Mipmap可以通过软件和硬件生成，软件生成的质量不高，硬件生成速度快，但它只是一个扩展，在opengl1.4才被纳入opengl核心api中

glTexParameteri(GL\_TEXTURE\_2D, GL\_GENERATE\_MIPMAP, GL\_TRUE)

当这个参数被设置为GL\_TRUE时，所有调用glTexImage或者glTexSubImage都会自动更新纹理贴图（第0级）和所有更底层的纹理

纹理对象

glTexImage、glTexSubImage和gluBuildMipmaps这些函数的调用消耗的时间特别多，这些函数大量的移动内存，有时需要重新调整数据的格式来使用一些内部表示，在纹理之间切换或者重新加载不同的纹理图片会带来较大的性能开销。为了减少这些开销，我们可以使用纹理对象。纹理对象允许你一次性加载多个纹理状态（包括纹理图像），然后在他们之间快速切换。纹理状态由当前绑定的纹理对象来维护。

glGenTexture生成纹理对象

glBindTexture绑定纹理对象，在此函数之后，纹理图像的加载和纹理参数的设置都只影响当前绑定的纹理对象

glDeleteTextures删除纹理对象，多次调用glGenTextures的开销较小，但是多次调用glDeleteTextures会导致有一些延时，原因是需要释放大量的内存空间，当不再需要此纹理对象时，要把该纹理对象删除，放置内存泄露。

gllsTexture判断纹理对象名称是否可用

void MipMap::setRC()

{

glEnable(GL\_DEPTH\_TEST);

// 图像信息

GLint width, height, component;

GLenum format;

// 设置纹理环境

glTexEnvi(GL\_TEXTURE\_2D, GL\_TEXTURE\_ENV, GL\_REPLACE);

// 生成纹理对象

glGenTextures(NUMBER, texture);

cout << "texture[0] = " << texture[0] << ", texture[1] = " << texture[1] << ", texture[2] = " <<

texture[2] << endl;

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

{

void \*pImage = gltLoadTGA(fileName[i], &width, &height, &component, &format);

if(pImage != nullptr)

{

std::cout << fileName[i] << " load success!"<< std::endl;

// 绑定纹理对象，生成mipmap

glBindTexture(GL\_TEXTURE\_2D, texture[i]);

gluBuild2DMipmaps(GL\_TEXTURE\_2D, component, width, height, format, GL\_UNSIGNED\_BYTE, pImage);

g**lTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);**

**glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);**

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

}

free(pImage);

}

glEnable(GL\_TEXTURE\_2D);

}

void MipMap::render()

{

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

glPushMatrix();

//移动和旋转

glTranslatef(0.0f, 0.0f, zPos);

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

for(GLfloat z = -60.0f; z <= 0.0f; z += 10.0f)

{

//绑定地板纹理绘制地板，注意glBindTexture在glBegin和glEnd中是无效的

glBindTexture(GL\_TEXTURE\_2D, texture[FLOOR]);

glBegin(GL\_QUADS);

glTexCoord2f(0.0f, 0.0f);

glVertex3f(-10.0f, -10.0f, z);

glTexCoord2f(1.0f, 0.0f);

glVertex3f(-10.0f, -10.0f, z + 10.0f);

glTexCoord2f(1.0f, 1.0f);

glVertex3f(10.0f, -10.0f, z + 10.0f);

glTexCoord2f(0.0f, 1.0f);

glVertex3f(10.0f, -10.0f, z);

glEnd();

//绑定天花板纹理

glBindTexture(GL\_TEXTURE\_2D, texture[CEILING]);

glBegin(GL\_QUADS);

glTexCoord2f(0.0f, 0.0f);

glVertex3f(-10.0f, 10.0f, z);

glTexCoord2f(1.0f, 0.0f);

glVertex3f(-10.0f, 10.0f, z + 10.0f);

glTexCoord2f(1.0f, 1.0f);

glVertex3f(10.0f, 10.0f, z + 10.0f);

glTexCoord2f(0.0f, 1.0f);

glVertex3f(10.0f, 10.0f, z);

glEnd();

//绑定砖墙的纹理

glBindTexture(GL\_TEXTURE\_2D, texture[BRICK]);

glBegin(GL\_QUADS);

glTexCoord2f(0.0f, 0.0f);

glVertex3f(-10.0f, -10.0f, z);

glTexCoord2f(1.0f, 0.0f);

glVertex3f(-10.0f, 10.0f, z);

glTexCoord2f(1.0f, 1.0f);

glVertex3f(-10.0f, 10.0f, z + 10.0f);

glTexCoord2f(0.0f, 1.0f);

glVertex3f(-10.0f, -10.0f, z + 10.0f);

glTexCoord2f(0.0f, 0.0f);

glVertex3f(10.0f, -10.0f, z);

glTexCoord2f(1.0f, 0.0f);

glVertex3f(10.0f, 10.0f, z);

glTexCoord2f(1.0f, 1.0f);

glVertex3f(10.0f, 10.0f, z + 10.0f);

glTexCoord2f(0.0f, 1.0f);

glVertex3f(10.0f, -10.0f, z + 10.0f);

glEnd();

}

}

在上面的例子中，设置mip贴图纹理过滤器时，它被选择为只用于缩小过滤器，这是最为典型的情况，因为在 opengl选择了最大可用的mip层后，就没有更大mip层可以使用，从本质上来，这相当于一旦传递了一个门槛值，实际使用的就是最大纹理图像，不再有其他的mip贴图层可以选择。

**常驻纹理：**绝大多数的opengl实现支持有限数量的高性能纹理内存，位于这种内存中的纹理访问速度非常快，性能非常高。一开始，所有加载的纹理都存储在这块内存中，但是，在典型情况下，这种内存的数量是有限的，在有些时候，纹理可能需要存储在较慢的内存中，这种较慢的内存常常位于opengl硬件的外部（例如存储在PC的系统内存，而不是存储在**图形卡或AGP内存**中）。

为了优化渲染性能，opengl会自动把经常访问的纹理移动到高性能内存中，位于这种高性能内存中的纹理就称为常驻纹理。我们可以通过函数glAreTexturesResident来判断某个纹理是否是常驻纹理

**纹理优先级：**在默认情况下，绝大多数的opengl实现使用最常使用（MFU，Most Frequently used）算法来决定哪个纹理应该作为常驻纹理，但是如果几个较小的纹理的使用频率只是稍微高于一个大得多的纹理，按照这种算法，纹理处理的性能可能会受到相当程度的影响，我们可以通过glPrioritizeTextures这个函数设置纹理的优先级。

mipMap如何绑定纹理（参考cocos2dx）

**opengl中的压缩纹理**

纹理贴图可以在3D渲染中获取难以置信的效果，它在顶点处理上需要付出的代价非常少，但是，纹理存在一个缺点，他们需要大量的内存来进行存储和处理。早起对纹理压缩的尝试就是简单地把纹理作为jpg文件存储，在加载之后进行解压（在调用glTexImage之前），这种做法可以节省磁盘空间并减少在网络上传输图形所需要的时间，但对于加载到图形硬件内存的纹理图像的存储要求则没有多大的帮助

Opengl所支持的纹理压缩远远不止简单地允许加载经过压缩的纹理。在绝大多数实现中，纹理数据在图形硬件内存中仍然保持压缩状态，这就允许我们在较小的内存中加载更多的纹理，从而显著地改善纹理处理的性能，这是由于在纹理过滤时减少了纹理交换（移动纹理）并使用了更少的内存。

压缩纹理：

为了获得opengl对压缩纹理的支持，纹理数据一开始并不需要进行压缩，我们可以在加载一幅纹理图像时请求opengl对它进行压缩，这是在glTexImage函数中把internalFormat参数设置为下面值之一实现的。

压缩格式 基本内部格式

GL\_COMPRESSED\_ALPHA GL\_ALPHA

GL\_COMPRESSED\_LUMINANCE GL\_LUMINANCE

GL\_COMPRESSED\_LUMINANCE\_ALPHA GL\_LUMINANCE\_ALPHA

GL\_COMPRESSED\_INTENSITY GL\_INTENSITY

GL\_COMPRESSED\_RGB GL\_RGB

GL\_COMPRESSED\_RGBA GL\_RGBA

当使用上面的值进行压缩之后，opengl会选择最适当的纹理压缩格式，我们可以使用glHint来指定希望opengl根据最快速度还是最佳质量算法来选择压缩格式。

加载压缩纹理

在前面的应用中，可以让opengl用一种内部支持的格式对纹理进行压缩，用glGetCompreddedTexImage函数（相当于未压缩纹理的glTexImage函数）提取经过压缩的数据并将它保存在磁盘中。

为了加载压缩纹理，可以使用glCompressTexImage，这个实际上与glTexImage相同，仅有的区别是这些函数internalFormat参数必须制定一种受到支持的压缩纹理图像格式。

**多重纹理**

现代的opengl硬件实现支持在几何图形上同时应用两个或更多的纹理，也就是一个顶点可以对应多个纹理单元。

首先，如果一个显卡支持多重纹理，表示它可以将几次渲染合为一次来进行，在这个渲染过程中，你可以在不同的纹理单元分别放入第0次、第1次、第2次需要渲染的对象，当程序向屏幕绘图时，将这些渲染结果叠加起来，将最终结果显示到屏幕。

每个纹理单元对应有其纹理环境，这个纹理环境致命了渲染的纹理图片、渲染参数、过滤参数等，其中有一个重要的项目指明了程序将渲染结果进行逐个叠加的过程中，当前的纹理单元内的内容如何与前面的内容进行混合。

在默认情况下，第一个纹理单元是活动纹理单元，所有的纹理命令（除了glTexCoord之外）都会影响当前的活动纹理单元，我们可以使用glActiveTexture函数（把纹理单元标识符作为参数）修改当前的纹理单元。例如，为了切换到第二个纹理单元并在这个纹理单位上启用2D纹理，可以使用下面的代码：

glActiveTexture(GL\_TEXTURE1);

glEnable(GL\_TEXTURE\_2D);

为了在第二个纹理单位上禁用纹理，并切换到第一个纹理单位（基纹理单位），可以私用下面的代码：

glDisable(GL\_TEXTURE\_2D);

glActiveTexture(GL\_TEXTURE0);

向glTextParameter、glTexEnv、glTexGen、glTexImage和glBindTexture这样的所有纹理函数调用都只作用域当前的纹理单元。当几何图形被渲染时，纹理贴图根据以前指定的纹理环境和参数，从所有已启用的纹理单元进行应用。

在使用多重纹理时，我们仍然可以使用glTexCoord指定纹理坐标，但是，这些坐标只用于第一个纹理单位（GL\_TEXTURE0）。如果想分别为每个纹理单元指定纹理坐标，需要使用新的纹理坐标函数之一：

glMultiCoord1f(GLenum textUnit, GLfloat s)

glMultiCoord2f(GLenum textUnit, GLfloat s, GLfloat t)

glMultiCoord3f(GLenum textUnit, GLfloat s, GLfloat t, GLfloat r)

注意：每一个纹理单元都有自己的纹理矩阵

[**glActiveTexture与glBindTexture**](http://blog.csdn.net/zhuyingqingfen/article/details/19968341)**区别**

<http://blog.csdn.net/zhuyingqingfen/article/details/19968341>

可以这样简单的理解为：显卡中有N个纹理单元（具体数目依赖你的显卡能力），每个纹理单元（GL\_TEXTURE0、GL\_TEXTURE1等）都有GL\_TEXTURE\_1D、GL\_TEXTURE\_2D等，如下代码：

**[cpp]** [view plain](http://blog.csdn.net/zhuyingqingfen/article/details/19968341) [copy](http://blog.csdn.net/zhuyingqingfen/article/details/19968341)

[print](http://blog.csdn.net/zhuyingqingfen/article/details/19968341)[?](http://blog.csdn.net/zhuyingqingfen/article/details/19968341)[在CODE上查看代码片](https://code.csdn.net/snippets/205456)

1. **struct** TextureUnit
2. {
3. GLuint targetTexture1D;
4. GLuint targetTexture2D;
5. GLuint targetTexture3D;
6. GLuint targetTextureCube;
7. ...
8. };
10. TextureUnit textureUnits[GL\_MAX\_TEXTURE\_IMAGE\_UNITS]
11. GLuint currentTextureUnit = 0;

默认情况下当前活跃的纹理单元为0.

**[cpp]** [view plain](http://blog.csdn.net/zhuyingqingfen/article/details/19968341) [copy](http://blog.csdn.net/zhuyingqingfen/article/details/19968341)

[print](http://blog.csdn.net/zhuyingqingfen/article/details/19968341)[?](http://blog.csdn.net/zhuyingqingfen/article/details/19968341)[在CODE上查看代码片](https://code.csdn.net/snippets/205456)

1. **void** glActiveTexture(GLenum textureUnit)
2. {
3. currentTextureUnit = textureUnit  - GL\_TEXTURE0 ;
4. }

glActiveTextue 并不是激活纹理单元，而是选择当前活跃的纹理单元。

**[cpp]** [view plain](http://blog.csdn.net/zhuyingqingfen/article/details/19968341) [copy](http://blog.csdn.net/zhuyingqingfen/article/details/19968341)

[print](http://blog.csdn.net/zhuyingqingfen/article/details/19968341)[?](http://blog.csdn.net/zhuyingqingfen/article/details/19968341)[在CODE上查看代码片](https://code.csdn.net/snippets/205456)

1. **void** glBindTexture(GLenum textureTarget, GLuint textureObject)
2. {
3. TextureUnit \*texUnit = &textureUnits[currentTextureUnit];
4. **switch**(textureTarget)
5. {
6. **case** GL\_TEXTURE\_1D: texUnit->targetTexture1D = textureObject; **break**;
7. **case** GL\_TEXTURE\_2D: texUnit->targetTexture2D = textureObject; **break**;
8. **case** GL\_TEXTURE\_3D: texUnit->targetTexture3D = textureObject; **break**;
9. **case** GL\_TEXTURE\_CUBEMAP: texUnit->targetTextureCube = textureObject; **break**;
10. }
11. }    
    从示例代码中可以看到：当绑定纹理目标时，所作用的是当前活跃的纹理单元。

**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;



**深度缓冲区：**

深度缓冲区存储每个像素的深度值， 通常是根据物体和观察点的距离来测量的，因此具有较大深度值的像素有可能被具有较小深度值的像素所覆盖

**模板缓冲区：**

与深度缓冲区大小相同，通过设置模板缓冲每个像素的值，可以指定在渲染的时候只渲染某些像素，从而可以达到一些特殊的效果

**积累缓冲区：**

与RGBA模式下的颜色缓冲区一样，累积缓冲区也用于存储RGBA颜色数据（在颜色索引模式下使用累积缓冲区的结果是未定义的）。累积缓冲区通常用于把一系列的图像合成为一幅图像。通过这种方法，可以对图像进行超量采样，然后对样本求平均值，并将结果写入到颜色缓冲区中，从而实现场景的抗锯齿处理。我们不能把数据直接写入到累积缓冲区，累积操作总是以矩形块为单位对数据进行操作，冗长是把数据移入或移出颜色缓冲区

累积缓冲区通过void glAccum(GLenum op, GLfloat value)控制，第一个参数表示对累积缓冲区所进行的操作，第二个参数是浮点数用于指定缩放因子

OpenGL支持的累积缓冲区的操作如下表：

GL\_ACCUM 将颜色缓冲区的颜色值进行放缩后，累加到累积缓冲区

GL\_LOAD 将颜色缓冲区的颜色值进行放缩后，替换掉累积缓冲区的颜色值

GL\_RETURN 将累积缓冲区的颜色值进行放缩后，拷贝会颜色缓冲区

GL\_MULT 将累积缓冲区的颜色值进行放缩后，替换掉原累积缓冲区的颜色值

GL\_ADD 将累积缓冲区的颜色值进行放缩后，累加到累积缓冲区

由于累积缓冲区会带来大内存的开销，因袭在实时应用程序中比较少用，但在非实时应用程序中，可以产生实时应用程序无法做到的效果。例如，你可以多次渲染场景，并在每次渲染时进行抖动零点几个像素，这样就可以产生整个场景的反走样效果，比多重采样的效果还要好。还可以模糊前景或背景，然后清洗的渲染一个物体来模拟照相机景深的效果

void drawMotionBlur()

{

GLfloat rot = 35.0f;

GLfloat pass = 10.0f;

for(GLsizei i = 0; i < 10; i++)

{

rot += 0.75f;

drawGemerotry(rot);

// 颜色缓冲区复制到累积缓冲区

if(i == 0)

{

glAccum(GL\_LOAD, 0.5f);

}

else

{

// 颜色缓冲区累加到累积缓冲区

glAccum(GL\_ACCUM, (0.5f \* 1 / pass));

}

}

// 累积缓冲区复制到颜色缓冲区，在之后会调用glutSwapBuffers进行缓冲区的交换

glAccum(GL\_RETURN, 1.0f);

}

void drawGemerotry(GLfloat rot)

{

glPushMatrix();

{

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

drawGround();

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

glTranslatef(0.0f, 0.3f, -3.5f);

glRotatef(-rot \* 2.0f, 0.0f, 1.0f, 0.0f);

glTranslatef(1.0f, 0.0f, 0.0f);

glutSolidSphere(0.1f, 17, 13);

}

glPopMatrix();

}

void drawGround()

{

GLsizei colorIndex = 0;

GLfloat range = 20.0f;

GLfloat step = 0.5f;

GLfloat zMin = 0.0f, zMax = 40.0f;

glShadeModel(GL\_SMOOTH);

glEnable(GL\_DEPTH\_TEST);

for(GLfloat x = -range; x <= range; x += step)

{

colorIndex = 0;

glBegin(GL\_LINE\_STRIP);

for(GLfloat z = zMin; z <= zMax; z += step)

{

if(colorIndex % 2 == 0)

{

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

}

else

{

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

}

glVertex3f(x, 0, -z);

colorIndex++;

}

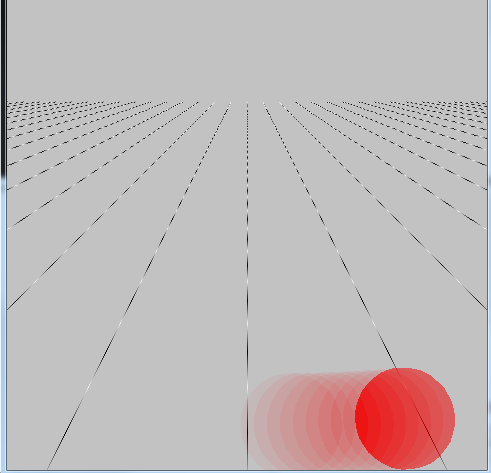
glEnd();

}

}

注意：一定要开启累积缓冲区

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



**Alpha测试**

Alpha测试允许我们告诉opengl丢弃那些alpha值无法通过alpha比较测试的片元，被丢弃的片元不写入颜色、深度、模板或累计缓冲区，这个特性允许丢弃一些值来改善性能，并消除深度缓冲区中那些可能在颜色缓冲区中不可见的几何图形（由于他们具有非常低的alpha值），alpha测试值和比较函数是通过glAlphaFunc函数指定的。

在上面的例子中：

glEnable(GL\_ALPHA\_TEST);

glAlphaFunc(GL\_NEVER, 0.5f);

当加入上面的代码后，就发现什么都画不出来了。

**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

**OpenGL的工作流程**

http://blog.163.com/zhaoxin851055@126/blog/static/81129298200927102644306/

**1）几何顶点被组合为图元**

包括矩阵变换、光照、颜色以及纹理坐标等，一旦这些计算完成，几何顶点被组合为图元（点、线或多边形），还包括相关边界标识、法线等

输入数据、矩阵变换、光照和颜色、生成纹理坐标、图元组装

**2）图元被合成片元**

图元分为几个步骤转换为像素片元：图元被适当的裁剪、颜色和纹理数据也相应作出调整，相关的坐标被转换为窗口坐标。最后，光删化将裁剪好的图元转换为像素片元

裁剪、转换到窗口坐标、光栅化

**3）片源被转换为帧缓冲区中的像素数据**

**片元测试**

在opengl确定了应该生成的片元及绘制颜色后，仍然需要几个处理过程来控制如何将该片元作为一个像素绘制到帧缓存中。

**OpenGL光照**

光源的一般属性包括：镜面反射光线颜色、漫反射光线颜色、环境光线颜色与光源位置

光线的衰减：光源发出的光线强度会随着传播距离越来越大而变弱（无穷远光源除外）。光线强度会乘以一个衰减因子

衰减因子=1 / (K1 + K2 \* d + k3 \* d ^ 2) 其中d为光源距离

（无穷远光源的衰减因子为1）

方向光源发出的光线会随着偏移中心方向的角度增大而减弱

漫反射光：如果它从正面照射表面，看起来就更亮一些，反之，如果它写着掠过表面，看起来就暗一些，但是，当它撞击表面时，它会均与地向所有方向发散，因此，不管眼睛在哪个位置，漫反射光看上去总是一样亮

材质：材质是光照效果中的重要属性。材质描述了物体表面的光学物理属性，决定了光线在该表面光线反射的具体情况

物体表面的反射分为漫反射和镜面反射。物体反射各种类型光源的情况都可以分为漫反射和镜面反射两种

材质的属性包括：镜面反射颜色、漫反射颜色、环境光颜色、光洁度、自发光颜色

镜面反射颜色、漫反射颜色、环境光颜色分别与光源的镜面反射光颜色、光源的漫发射颜色、光源的环境光颜色共同决定物体表面的镜面发射颜色、漫反射颜色、环境光颜色。3种类型的结果分别计算，然后叠加共同确定发射表面像素值的颜色

Opengl确定某个顶点颜色可以分为两种情况：开启光照渲染和关闭光照渲染的情况。当关闭光照渲染的时候，顶点的颜色由opengl状态机绘制该物体时的颜色决定；当开启光照渲染时，绘制该物体的opengl状态机的颜色将对该物体上顶点的颜色没有任何影响，此时物体顶点的颜色由物体材质、光照叠加效果以及物体表面的纹理贴图（如果有纹理贴图的话）共同决定。

// 设置材质

GLfloat material\_ambient[] = {0.3f, 0.2f, 0.7f, 1.0f};

GLfloat material\_diffuse[] = {0.2, 0.8f, 1.0f, 1.0f};

glMaterialfv(GL\_FRONT, GL\_AMBIENT, material\_ambient);

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, material\_diffuse);

// 设置光照

GLfloat light\_ambient[] = {1.0f, 0.2f, 0.2f, 0.0f};

GLfloat light\_diffuse[] = {1.0f, 0.7f, 1.0f, 1.0f};

GLfloat light\_specular[] = {0.0f, 0.0f, 1.0f, 1.0f};

GLfloat light\_position[] = {-10.0f, 0.0f, 0.0f};

glLightfv(GL\_LIGHT0, GL\_AMBIENT, light\_ambient);

glLightfv(GL\_LIGHT0, GL\_DIFFUSE, light\_diffuse);

glLightfv(GL\_LIGHT0, GL\_POSITION, light\_position);

glLightfv(GL\_LIGHT0, GL\_SPECULAR, light\_specular);

glEnable(GL\_LIGHTING); // 开启光照渲染

glEnable(GL\_LIGHT0); // 开启光源

如果我们要打开全局光照，也就是在没有指定光源的时候，让场景中的物体也可以被照亮，可以使用下面的代码（已经测试）：

// 设置全局光照，是的物体在没有光照的情况下也可以被照亮

GLfloat global\_ambient[] = {0.5f, 0.2f, 0.2f, 1.0f};

glLightModelfv(GL\_AMBIENT\_AND\_DIFFUSE, global\_ambient);

**注意，在这种情况下也是需要开启光照渲染的**glEnable(GL\_LIGHTING); **在启动混合之前，这些颜色中的alpha成分是不会使用的，在此之前，可以完全忽略alpha段，在opengl中，被光照的物体并不会向其他物体发射光线**

由于物体有的表面相对于观察者来说是背向观察者的，即使光源照到这些表面上的顶点也不能被观察者看到，但是这些顶点也在渲染场景的时候，消耗了大量的计算时间，即使对最后的观察效果没有任何影响。所以可以设置背面的这些顶点不进行计算，添加如下代码：

// 关闭背面顶点的计算

glLightModeli(GL\_LIGHT\_MODEL\_TWO\_SIDE, GL\_FALSE);

一个光源对场景中3D物体表面一个顶点的颜色贡献如下：

顶点颜色=全局环境光颜色+衰减因子\*（环境+漫反射+镜面反射）

考虑到材质自发光颜色属性和多个光源后，3D物体表面一个顶点的颜色为：

顶点颜色=材质自发光颜色+所有光源之和（全局环境光颜色+衰减因子\*（环境+漫反射+镜面反射））

环境光不依赖于光源的方向

漫反射光依赖于光源的方向和法线的方向

镜面反射光依赖于光源的方向、法线的方向和视角方向

**注意：在光照计算中，顶点法线必须是经过规范化的（normalize）**

光照中的纹理

当场景中的3D物体表面有纹理贴图时，纹理图像也将对点的最终颜色产生影响。在这种情况下，物体表面点的颜色由场景中的所有光源（包括环境光源），物体的材质，物体表面的纹理共同确定。渲染过程是先进行光照处理，再进行纹理处理，纹理处理会影响到最终3D物体表面像素点的颜色。

但是在这种情况下会发现镜面反射区可能被覆盖，为了解决这个问题，可以

glLightModefv(GL\_LIGHT\_MODEL\_COLOR\_CONTROL, GL\_SEPARATE\_SPECULAR\_COLOR),

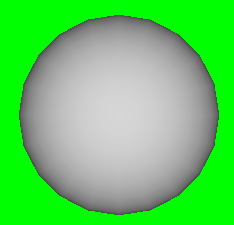
这样，每个顶点光照计算将产生两种颜色，主颜色和辅助颜色，前者包含所有非镜面发射光照的贡献，后者是所有镜面发射光照的总贡献。纹理映射的时候只将主颜色和纹理颜色混合起来，执行完纹理映射之后，再将主颜色和纹理颜色的混合结果与辅助颜色混合起来

光源GL\_LIGHT0与其他几个光源不同，它的

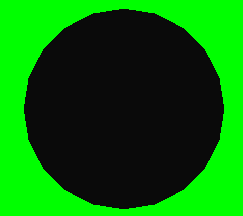
GL\_DIFFUSE、GL\_SPECULAR的默认值是（1.0，1.0,1.0，1.0）白光

而其他的光源默认值是（0.0,0.0,0.0,1.0）

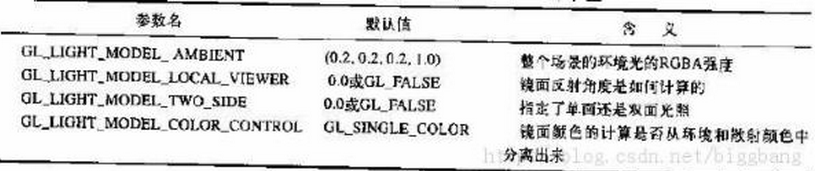
（已经测试）发现直接调用glEnable(GL\_LIGHT0);



glEnable(GL\_LIGHT1);



glLightModelfv(GLenum pname, const GLfloat \*params)



全局环境光是一种零成本的在opengl的光照计算结果上添加一个偏移量的方式，其默认的RGBA值是（0.2,0.2,0.2,1.0）

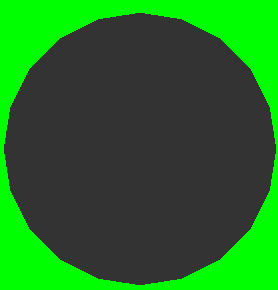
Gouraud Shading与Phong Shading

在图形渲染中有两种着色模式，高洛德着色与冯氏着色。高洛德着色也被称为per-vertex着色，它是在顶点着色阶段对顶点进行颜色计算，然后在光栅化阶段对这些顶点颜色进行线性插值形成片元的颜色；冯氏着色也被称为per-pixel像素着色，它是在片元着色阶段对每个片元（像素）进行颜色计算，无疑，插值的颜色效果没有针对每个片元进行颜色计算的效果好（除非你的图元切分到像素近似大小，不过这样GPU肯定吃不消，计算量巨大）。

Gouraud的消耗与模型的精细度相关，phong的消耗与render target分辨率相关



如果只开启了全局环境光，发现下面图的每个像素值均为51，



实验得出最终顶点的环境光=全局环境光\*顶点环境光+光源环境光\*顶点环境光

GLfloat globalAmebient[] = {1.0f, 1.0f, 1.0f, 1.0f};

glLightModelfv(GL\_LIGHT\_MODEL\_AMBIENT, globalAmebient);

// 添加光源

GLfloat lightPosition[] = {-100.0f, 0.0f, 10.0f, 1.0f};

GLfloat lightAmbientColor[] = {1.0f, 0.0f, 0.0f, 1.0f};

GLfloat lightDiffuseColor[] = {0.0f, 0.0f, 0.0f, 1.0f};

GLfloat lightSpecularColor[] = {1.0f, 1.0f, 1.0f, 1.0f};

glLightfv(GL\_LIGHT0, GL\_AMBIENT, lightAmbientColor);

glLightfv(GL\_LIGHT0, GL\_DIFFUSE, lightDiffuseColor);

glLightfv(GL\_LIGHT0, GL\_SPECULAR, lightSpecularColor);

glLightfv(GL\_LIGHT0, GL\_POSITION, lightPosition);

//glLightModeli(GL\_LIGHT\_MODEL\_LOCAL\_VIEWER, GL\_TRUE);

//glLightModeli(GL\_LIGHT\_MODEL\_TWO\_SIDE, GL\_TRUE);

// 设置材质

GLfloat ambientColor[] = {0.2f, 0.2f, 0.2f, 1.0f};

GLfloat diffuseColor[] = {0.0f, 0.0f, 0.0f, 1.0f};

GLfloat specularColor[] = {1.0f, 1.0f, 1.0f, 1.0f};

glMaterialfv(GL\_FRONT, GL\_AMBIENT, ambientColor);

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, diffuseColor);

glMaterialfv(GL\_FRONT, GL\_SPECULAR, specularColor);

**glMateriali(GL\_FRONT, GL\_SHININESS, 10); // 镜面指数,这个特别重要**

glEnable(GL\_LIGHTING);

glEnable(GL\_LIGHT0);



**注意：上面有一个参数特别重要GL\_SHININESS，表示镜面指数，取值范围为0-128，该值越小，表示材质越粗糙，点光源发射的光线照射到上面，也可以产生较大的亮点，该值越大，表示材质类似于镜面，光源照射到上面后，产生较小的亮点**

**将上面的10改为100**

**从上面的表中可以看出，该值的默认值为0**

**设置材质属性的方式有两种：**

1 在指定每个多边形或每组多边形之前使用glMaterial函数

GLfloat ambientColor[] = {0.2f, 0.2f, 0.2f, 1.0f};

GLfloat diffuseColor[] = {0.0f, 0.0f, 0.0f, 1.0f};

GLfloat specularColor[] = {1.0f, 1.0f, 1.0f, 1.0f};

glMaterialfv(GL\_FRONT, GL\_AMBIENT, ambientColor);

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, diffuseColor);

glMaterialfv(GL\_FRONT, GL\_SPECULAR, specularColor);

glMateriali(GL\_FRONT, GL\_SHININESS, 100);

glBegin(GL\_TRIANGLES);

glVertex3f(-15.0f, 0.0f, 30.0f);

glVertex3f(0.0f, 15.0f, 30.0f);

glVertex3f(0.0f, 0.0f, -56.0f);

glEnd();

在glMaterial调用之后所指定的所有图元都将受到最后一次所设置的值影响，直到再次调用glMaterial函数

2 颜色追踪方法（推荐）

可以告诉opengl仅仅通过调用glColor来设置材料属性，为了启动颜色追踪，需要调用

glEnable(GL\_COLOR\_MATERIAL);

接着，glColorMaterial函数根据glColor所设置的值来指定材料参数

GLfloat globalAmbient[] = {1.0f, 1.0f, 1.0f, 1.0f};

glLightModelfv(GL\_LIGHT\_MODEL\_AMBIENT, globalAmebient);

glEnable(GL\_COLOR\_MATERIAL);

glColorMaterial(GL\_FRONT, GL\_AMBIENT\_AND\_DIFFUSE);

采用：

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

glutSolidSphere(10.0f, 50, 50);



glColor4f(0.5f, 0.5f, 0.5f, 1.0f);

glutSolidSphere(10.0f, 50, 50);



关于光照中光源的位置：

1）有待验证：顶点（基于局部坐标系） Model（基于世界坐标系） View（基于世界坐标系） Projection（基于眼坐标系） LightPosition（基于世界坐标系也可能基于眼坐标，取决于模型视图矩阵是先设置还是后设置？？？如果是之后设置是否应该是没有起到变化，而导致一直是用的默认的视点）

光源的坐标也会受到Model变换的影响（已经验证）

GLfloat lightPosition[] = {150.0f, 0.0f, 0.0f, 1.0f};



glTranslatef(-300.0f,0.0f, 0.0f);

GLfloat lightPosition[] = {150.0f, 0.0f, 0.0f, 1.0f};



在opengl中，至少支持8种独立的光源，它们可以出现在场景中的任何地方或者视景体之外，光源位置的控制也是通过模型视图矩阵实现的，包括平移、旋转、缩放（因为它会改变法向量，慎用）等，这样，光源可以看做一个定义在世界坐标系中的普通的几何对象。

在光照程序中：

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

glTranslatef(-300.0f,0.0f, 0.0f);

GLfloat lightPosition[] = {150.0f, 0.0f, 0.0f, 1.0f};

glLightfv(GL\_LIGHT0, GL\_POSITION, lightPosition);



gluLookAt(-120.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);

glTranslatef(-300.0f,0.0f, 0.0f);

GLfloat lightPosition[] = {150.0f, 0.0f, 0.0f, 1.0f};

glLightfv(GL\_LIGHT0, GL\_POSITION, lightPosition);



对于普通的几何对象，我们在世界坐标系中定义它们，平移、旋转等也是在改变它们的世界坐标，这些坐标在提交给投影过程之前，opengl会根据当前世界坐标系和摄像机坐标系的关系把世界坐标系换换为摄像机坐标系，然后提交给投影过程，每个几何对象的坐标都会经过这种坐标变换

lightPosition包含了光的位置，这个数组最后一个值是1.0，表示指定的坐标是光源的位置，如果这个数组的最后一个值是0。0，则表示光是从无限远地方沿这个数组所指定的向量照射过来的

方向光源与聚光灯效果

在设置lightPosition的时候，第四个分量w为0，则表示光线从无穷远处向指定的位置发射而来，这就是方向光源。如果不为0，在x/w，y/w，z/w表示了光源的位置

使用方向光源的时候会发现光源的模型变换基本上没有什么影响

GLfloat lightPosition[] = {150.0f, 0.0f, 0.0f, 0.0f};



glTranslatef(-30000000.0f,0.0f, 0.0f);

GLfloat lightPosition[] = {150.0f, 0.0f, 0.0f, 0.0f};



方向性光源会均匀地照射在物体的表面，也就是说所有的光线都是平行的，反之，对于位置性光源，光线从光源发散开来，通常太阳可以近似地被认为是方向光源

聚光灯属性：

GL\_SPOT\_DIRECTION：表示一个向量，即光源发射的方向

GL\_SPOT\_EXPONENT：一个值，表示聚光的程度，为0时表示光照范围内向各个方向发射的光线强度相同，为正数时表示光照向中央集中，正对发射方向的位置受到更多光照，其他位置受到较少光照，数值越大，聚光效果就越明显

GL\_SPOT\_CUTOFF：一个值，表示一个角度，它是光源发射光线所覆盖角度的一半，其取值范围在0到90之间，也可以取180这个特殊值，取值为180时表示光源发射光线覆盖360度，即不适用聚光灯，向全周围发射。

GLfloat spotDirection[] = {0.0f, 0.0f, -1.0f};

glLightf(GL\_LIGHT0, GL\_SPOT\_CUTOFF, 60);

glLightf(GL\_LIGHT0, GL\_SPOT\_EXPONENT, 10.0f);

glLightfv(GL\_LIGHT0, GL\_SPOT\_DIRECTION, spotDirection);



同时经过测试发现，聚光灯也受到模型视图矩阵变换的影响

发光体

**颜色和材料的更多细节**

1）混合

我们知道，在正常情况下，opengl在渲染时把颜色值放在颜色缓冲区，我们还知道，每个片段（即像素）的深度值是放在深度缓冲区中的。当深度测试被关闭时，新的颜色值简单的覆盖颜色缓冲区中已经存在的颜色，当深度测试开启时，新的颜色片段只有当它们比原来的颜色更接近邻近的裁剪平面时才会替换原来的颜色片段。当然，这是在正常情况下，如果opengl打开了混合功能，这些规则就会突然失效。

glEnable(GL\_BLEND);

用专业术语表示则为，已经存储在颜色缓冲区的颜色称为目标颜色，这个颜色值包含了单独的红、绿、蓝以及一个可选的alpha值。作为当前渲染命令的结果进入颜色缓冲区的颜色值称为源颜色，它可能与目标颜色进行交互，也可能不与之进行交互，也就是说用源颜色去混合目标颜色，在默认情况下，混合方程式像下面这样：

Cf = (Cs \* S) + (Cd \* D)

其中Cf为最终计算产生的颜色，Cs为源颜色，Cd为目标颜色。S与D分别是源和目标混合因子，这些混合因子是用下面这个函数进行设置的。

glBlendFunc(GLenum S, GLenum D) 其中S与D都是枚举类型

比如glBlendFunc(GL\_SRC\_ALPHA,GL\_ONE\_MINUS\_SRC\_ALPHA)

这个函数表明源颜色的alpha成分越高，添加的源颜色成分就越多，目标颜色所保留的成分就越少。

除了采用默认的混合方程式以外，还可以修改混合方程式：

glBlendEquation(GLenum mode)

可选择为GL\_FUNC\_ADD（默认），GL\_FUNC\_SUBTRACT，GL\_FUNC\_REVERSE\_SUBTRACT，GL\_MIN，GL\_MAX

混合的另外一个功能就是抗锯齿,为了消除**图元**之间的锯齿状边缘，opengl使用混合功能来混合片断的颜色，也就是把像素的目标颜色与周围像素的颜色进行混合。从本质上来说，在任何图元的边缘上，像素颜色都会稍微延伸到相邻的像素。

打开抗锯齿功能非常简单，首先开启混合

glEnable(GL\_BLEND);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

我们还需要确保把混合方程式修改为GL\_ADD，不过由于是默认的方式，因此不需要设置（并不是所有的opengl实现都支持修改混合方程式），在设置之后，可以选择调用glEnable函数对点、直线或多边形进行抗锯齿处理

glEnable(GL\_POINT\_SMOOTH);

glEnable(GL\_LINE\_SMOOTH);

glEnable(GL\_POLYGON\_SMOOTH);

glHint是opengl的反走样函数，opengl实现反走样需要满足两个条件，一是启动混合，二是启用针对几何图元的反走样处理。

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

glEnable(GL\_BLEND);

glEnable(GL\_POLYGON\_SMOOTH);

glHint(GL\_POLYGON\_SMOOTH\_HINT, GL\_NICEST);

Blend混合是将源色和目标色以某种方式混合生成特效的技术。混合常用来绘制透明或半透明的物体，在混合中其关键作用的a值实际上是将源色和目标色按给定比率进行混合，以达到不同程度的透明，混合操作只能在RGBA模式下进行，颜色索引模式下无法指定a值，物体的绘制顺序会影响到opengl的混合处理

glBlendFunc(GLenum sfactor , GLenum dfactor); // 混合函数

sfactor：源混合因子

dfactor：目标混合因子

为了方便颜色混合，我们往往采用RGBA这种颜色模式。其中RGB表示色彩分量，而A就是混合因子（blend factor）。A，我们在图形、图像处理中常常表示为：alpha，它在图像处理中常用作为透明系数。  
我们指定了源和目标的混合因子后，OpenGL会对绘制对象的最终颜色做如下计算：  
设：源对象的某个顶点的颜色为(Rs, Gs, Bs, As)  
   目的对象对应此源对象顶点的颜色为(Rd, Gd, Bd, Ad)  
   源混合因子为：(Sr, Sg, Sb, Sa)  
   目的混合因子为：(Dr, Dg, Db, Da)  
那么，该顶点最终目标颜色为：  
(Rs \* Sr  <op>  Rd \* Dr,  Gs \* Sg  <op>  Gd \* Dg,  Bs \* Sb  <op>  Bd \* Db,  As \* Sa  <op>  Ad \* Da)  
其中，<op>可以是加法（＋），减法（－）， 逆向减法，最小值，最大值或按位逻辑操作，并且其优先级小于乘法（\*）。

然后我们使用glBlendEquation()来指定混合操作，参数可以是：GL\_FUNC\_ADD, GL\_FUNC\_SUBTRACT, GL\_FUNC\_REVERSE\_SUBTRACT, GL\_MIN, GL\_MAX。  
但这里要注意的是，OpenGL ES1.1没有glBlendEquation接口，因此只能做加法操作

注意： 所谓源颜色和目标颜色，是跟绘制的顺序有关的。假如先绘制了一个红色的物体，再在其上绘制绿色的物体。则绿色是源颜色，红色是目标颜色。如果顺序反过来，则 红色就是源颜色，绿色才是目标颜色。在绘制时，应该注意顺序，使得绘制的源颜色与设置的源因子对应，目标颜色与设置的目标因子对应。不要被混乱的顺序搞晕 。

// 采用顶点数组

GLfloat vertices1[] = {-0.3f, 0.5f, -2.0f, 1.0f,

-0.1f, -0.4f, -2.0f, 1.0f,

0.4f, -0.5f, -2.0f, 1.0f,

0.2f, 0.3f, -2.0f, 1.0f

};

GLfloat colors1[] = {0.0f, 1.0f, 0.0f, 1.0f,

0.5f, 0.5f, 0.5f, 1.0f,

1.0f, 0.0f, 0.0f, 1.0f,

0.0f, 0.0f, 1.0f, 1.0f

};

GLfloat vertices2[] = {-0.8f, 0.5f, -1.0f, 1.0f,

-0.2f, -0.4f, -1.0f, 1.0f,

0.6f, -0.5f, -1.0f, 1.0f,

0.4f, 0.3f, -1.0f, 1.0f

};

GLfloat colors2[] = {0.0f, 1.0f, 0.0f, 0.5f,

0.5f, 0.5f, 0.5f, 1.0f,

1.0f, 0.0f, 0.0f, 1.0f,

0.0f, 0.0f, 1.0f, 1.0f

};

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

glEnable(GL\_BLEND);

glEnable(GL\_POLYGON\_SMOOTH);

glHint(GL\_POLYGON\_SMOOTH\_HINT, GL\_NICEST);

glShadeModel(GL\_SMOOTH);

glPolygonMode(GL\_BACK, GL\_LINE);

glPolygonMode(GL\_FRONT, GL\_FILL);

glFrontFace(GL\_CCW);

glEnableClientState(GL\_VERTEX\_ARRAY);

glEnableClientState(GL\_COLOR\_ARRAY);

glVertexPointer(4, GL\_FLOAT, 0, vertices1);

glColorPointer(4, GL\_FLOAT, 0, colors1);

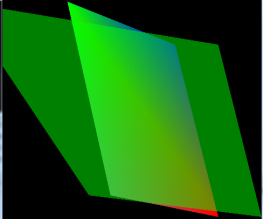
glDrawArrays(GL\_POLYGON, 0, 4);

**glShadeModel(GL\_FLAT);**

glVertexPointer(4, GL\_FLOAT, 0, vertices2);

glColorPointer(4, GL\_FLOAT, 0, colors2);

glDrawArrays(GL\_POLYGON, 0, 4);



注意上面的着色模式为单调模式，所以都是采用的第一个顶点的颜色和alpha值，如果将

注意：在顶点着色模式种，除了多边形是以第一个顶点颜色为标准外，其他的都是以最后一个顶点的颜色为标准

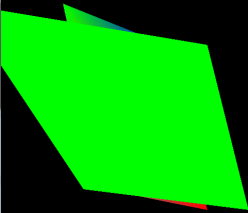
GLfloat colors2[] = {0.0f, 1.0f, 0.0f, 1.0f,

0.5f, 0.5f, 0.5f, 0.5f,

1.0f, 0.0f, 0.0f, 1.0f,

0.0f, 0.0f, 1.0f, 1.0f

};



如果将着色模式修改为glShadeModel(GL\_SMOOTH);

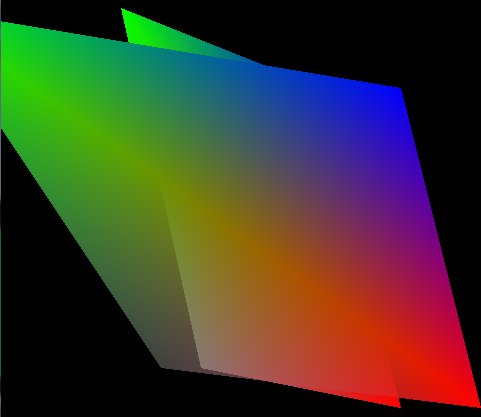
GLfloat colors2[] = {0.0f, 1.0f, 0.0f, 1.0f,

0.5f, 0.5f, 0.5f, 0.5f,

1.0f, 0.0f, 0.0f, 1.0f,

0.0f, 0.0f, 1.0f, 1.0f

};



现在改变两个物体的坐标位置：

// 采用顶点数组

GLfloat vertices1[] = {-0.3f, 0.5f, -2.0f, 1.0f,

-0.1f, -0.4f, -2.0f, 1.0f,

0.4f, -0.5f, -2.0f, 1.0f,

0.2f, 0.3f, -2.0f, 1.0f

};

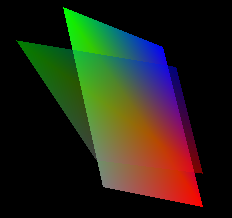
GLfloat colors2[] = {0.0f, 1.0f, 0.0f, 0.5f,

0.5f, 0.5f, 0.5f, 0.5f,

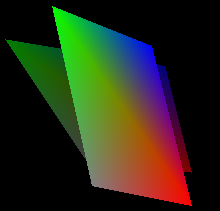
1.0f, 0.0f, 0.0f, 0.5f,

0.0f, 0.0f, 1.0f, 0.5f

};

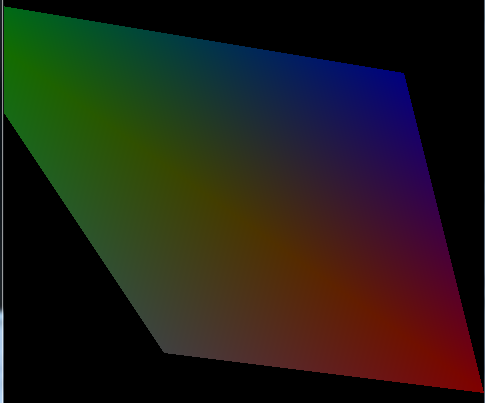


如果添加了：glEnable(GL\_DEPTH\_TEST);

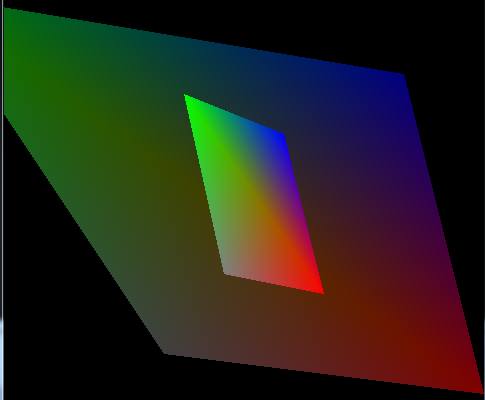


就会发现如果我们先绘制一个比较近的不透明物体，然后绘制一个比较远的透明物体，在调用深度测试的情况下，发现被挡住的不透明物体没有出现混合了（实际上是被遮挡的那一部分通过深度测试被舍弃了）

或者说我们先绘制一个近的透明的，后绘制一个远的不透明的，也会发现远的没有显示出来，原因就是在绘制近的透明的时候在深度缓冲区保存了深度值，后面的直接舍弃了



解决办法是：在绘制近的透明物体时设置深入缓冲区为只读的glDepthMask(FALSE);



实现三维混合

也许你迫不及待的想要绘制一个三维的带有半透明物体的场景了。但是现在恐怕还不行，还有一点是在进行三维场景的混合时必须注意的，那就是深度缓冲。  
深度缓冲是这样一段数据，它记录了每一个像素距离观察者有多近。在启用深度缓冲测试的情况下，如果将要绘制的像素比原来的像素更近，则像素将被绘制。否则，像素就会被忽略掉，不进行绘制。这在绘制不透明的物体时非常有用——不管是先绘制近的物体再绘制远的物体，还是先绘制远的物体再绘制近的物体，或者干脆以混乱的顺序进行绘制，最后的显示结果总是近的物体遮住远的物体。  
然而在你需要实现半透明效果时，发现一切都不是那么美好了。如果你绘制了一个近距离的半透明物体，则它在深度缓冲区内保留了一些信息，使得远处的物体将无法再被绘制出来。虽然半透明的物体仍然半透明，但透过它看到的却不是正确的内容了。  
要解决以上问题，需要在绘制半透明物体时将深度缓冲区设置为只读，这样一来，虽然半透明物体被绘制上去了，深度缓冲区还保持在原来的状态。如果再有一个物体出现在半透明物体之后，在不透明物体之前，则它也可以被绘制（因为此时深度缓冲区中记录的是那个不透明物体的深度）。以后再要绘制不透明物体时，只需要再将深度缓冲区设置为可读可写的形式即可。嗯？你问我怎么绘制一个一部分半透明一部分不透明的物体？这个好办，只需要把物体分为两个部分，一部分全是半透明的，一部分全是不透明的，分别绘制就可以了。  
即使使用了以上技巧，我们仍然不能随心所欲的按照混乱顺序来进行绘制。必须是先绘制不透明的物体，然后绘制透明的物体。否则，假设背景为蓝色，近处一块红色玻璃，中间一个绿色物体。如果先绘制红色半透明玻璃的话，它先和蓝色背景进行混合，则以后绘制中间的绿色物体时，想单独与红色玻璃混合已经不能实现了。  
总结起来，绘制顺序就是：首先绘制所有不透明的物体。如果两个物体都是不透明的，则谁先谁后都没有关系。然后，将深度缓冲区设置为只读。接下来，绘制所有半透明的物体。如果两个物体都是半透明的，则谁先谁后只需要根据自己的意愿（注意了，先绘制的将成为“目标颜色”，后绘制的将成为“源颜色”，所以绘制的顺序将会对结果造成一些影响）。最后，将深度缓冲区设置为可读可写形式。**（先做深度测试了之后才会做混合操作）**调用glDepthMask(GL\_FALSE);可将深度缓冲区设置为只读形式。调用glDepthMask(GL\_TRUE);可将深度缓冲区设置为可读可写形式。  
一些网上的教程，包括大名鼎鼎的NeHe教程，都在使用三维混合时直接将深度缓冲区禁用，即调用glDisable(GL\_DEPTH\_TEST);。这样做并不正确。如果先绘制一个不透明的物体，再在其背后绘制半透明物体，本来后面的半透明物体将不会被显示（被不透明的物体遮住了），但如果禁用深度缓冲，则它仍然将会显示，并进行混合。NeHe提到某些显卡在使用glDepthMask函数时可能存在一些问题，但可能是由于我的阅历有限，并没有发现这样的情况。

那么，实际的演示一下吧。我们来绘制一些半透明和不透明的球体。假设有三个球体，一个红色不透明的，一个绿色半透明的，一个蓝色半透明的。红色最远，绿色在中间，蓝色最近。根据前面所讲述的内容，红色不透明球体必须首先绘制，而绿色和蓝色则可以随意修改顺序。这里为了演示不注意设置深度缓冲的危害，我们故意先绘制最近的蓝色球体，再绘制绿色球体。  
为了让这些球体有一点立体感，我们使用光照。在(1, 1, -1)处设置一个白色的光源。代码如下：  
void setLight(void)  
{  
     static const GLfloat light\_position[] = {1.0f, 1.0f, -1.0f, 1.0f};  
     static const GLfloat light\_ambient[]   = {0.2f, 0.2f, 0.2f, 1.0f};  
     static const GLfloat light\_diffuse[]   = {1.0f, 1.0f, 1.0f, 1.0f};  
     static const GLfloat light\_specular[] = {1.0f, 1.0f, 1.0f, 1.0f};  
  
     glLightfv(GL\_LIGHT0, GL\_POSITION, light\_position);  
     glLightfv(GL\_LIGHT0, GL\_AMBIENT,   light\_ambient);  
     glLightfv(GL\_LIGHT0, GL\_DIFFUSE,   light\_diffuse);  
     glLightfv(GL\_LIGHT0, GL\_SPECULAR, light\_specular);  
  
     glEnable(GL\_LIGHT0);  
     glEnable(GL\_LIGHTING);  
     glEnable(GL\_DEPTH\_TEST);  
}  
每一个球体颜色不同。所以它们的材质也都不同。这里用一个函数来设置材质。  
void setMatirial(const GLfloat mat\_diffuse[4], GLfloat mat\_shininess)  
{  
     static const GLfloat mat\_specular[] = {0.0f, 0.0f, 0.0f, 1.0f};  
     static const GLfloat mat\_emission[] = {0.0f, 0.0f, 0.0f, 1.0f};  
  
     glMaterialfv(GL\_FRONT, GL\_AMBIENT\_AND\_DIFFUSE, mat\_diffuse);  
     glMaterialfv(GL\_FRONT, GL\_SPECULAR,   mat\_specular);  
     glMaterialfv(GL\_FRONT, GL\_EMISSION,   mat\_emission);  
     glMaterialf (GL\_FRONT, GL\_SHININESS, mat\_shininess);  
}  
有了这两个函数，我们就可以根据前面的知识写出整个程序代码了。这里只给出了绘制的部分，其它部分大家可以自行完成。  
void myDisplay(void)  
{  
     // 定义一些材质颜色  
     const static GLfloat red\_color[] = {1.0f, 0.0f, 0.0f, 1.0f};  
     const static GLfloat green\_color[] = {0.0f, 1.0f, 0.0f, 0.3333f};  
     const static GLfloat blue\_color[] = {0.0f, 0.0f, 1.0f, 0.5f};  
  
     // 清除屏幕  
     glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  
  
     // 启动混合并设置混合因子  
     glEnable(GL\_BLEND);  
     glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);  
  
     // 设置光源  
     setLight();  
  
     // 以(0, 0, 0.5)为中心，绘制一个半径为.3的不透明红色球体（离观察者最远）  
     setMatirial(red\_color, 30.0);  
     glPushMatrix();  
     glTranslatef(0.0f, 0.0f, 0.5f);  
     glutSolidSphere(0.3, 30, 30);  
     glPopMatrix();  
  
     // 下面将绘制半透明物体了，因此将深度缓冲设置为只读  
     glDepthMask(GL\_FALSE);  
  
     // 以(0.2, 0, -0.5)为中心，绘制一个半径为.2的半透明蓝色球体（离观察者最近）  
     setMatirial(blue\_color, 30.0);  
     glPushMatrix();  
     glTranslatef(0.2f, 0.0f, -0.5f);  
     glutSolidSphere(0.2, 30, 30);  
     glPopMatrix();  
  
     // 以(0.1, 0, 0)为中心，绘制一个半径为.15的半透明绿色球体（在前两个球体之间）  
     setMatirial(green\_color, 30.0);  
     glPushMatrix();  
     glTranslatef(0.1, 0, 0);  
     glutSolidSphere(0.15, 30, 30);  
     glPopMatrix();  
  
     // 完成半透明物体的绘制，将深度缓冲区恢复为可读可写的形式  
     glDepthMask(GL\_TRUE);  
  
     glutSwapBuffers();  
}  
  
大家也可以将上面两处glDepthMask删去，结果会看到最近的蓝色球虽然是半透明的，但它的背后直接就是红色球了，中间的绿色球没有被正确绘制。

**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

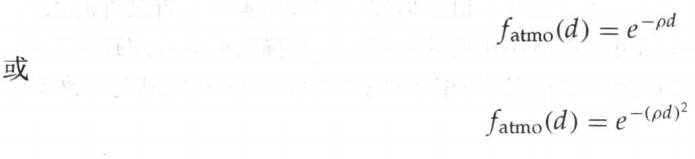
关于多重采样需要注意的一点，当它被启动时，点、直线和多边形的平滑特性都将被忽略，这就意味着在使用多重采样时，就不能同时使用点和直线的平滑处理。在一定的特定opengl实现中，点和直线如果采用平滑处理可能会比使用多重采样效果更好，因此，当我们绘制点和直线时，可以关闭多重采样，在绘制其他实心几何图形时再打开多重采样。

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

**雾**

在使用雾时，opengl把一种指定的雾颜色与完成了所有其他颜色计算的几何图形进行混合

对象颜色上的雾气效果是光照模型中有时会考虑的另一因素。雾气使颜色变淡、使对象变模糊，因此我们可以按照模拟空气的灰尘、烟或雾的多少来指定一个函数秀该表面颜色。雾气效果常用指数衰减函数来模拟：



赋给d的值是从观察位置到对象的距离，我们还在这两个指数函数中使用参数p来为雾气设定一个正密度。较高的p值生成较稠密的雾气且导致表面更柔和。在以对象的表面颜色计算出来后，将该颜色与雾气函数之一相乘来减少其强度，减少量依赖于雾气密度设定的值。

我们也可以使用线性的深度提示函数来简化雾气衰减函数从而代替指数函数。这使远距离对象的表面颜色强度减弱，但这样就失去了改变雾气密度的控制。

我们有时要模拟雾气的颜色。例如，有烟房间的空气可以用蓝灰色或用淡灰色建模，混合雾气颜色和对象颜色的计算如下：



这里的是一个指数或线性雾气衰减函数

即最终的颜色=对象颜色\*衰减指数+雾颜色\*(1-衰减指数)

float fogColor[] = {1.0f, 1.0f, 1.0f, 1.0f};

glEnable(GL\_FOG); // 开启雾

glFogfv(GL\_FOG\_COLOR, fogColor); // 设置雾颜色

glFogf(GL\_FOG\_START, 0.0f); // 雾从多远开始

glFogf(GL\_FOG\_END, 50.0f); // 雾从多远停止

glFogi(GL\_FOG\_MODE, GL\_LINEAR); // 采用哪种方程式

glFogf(GL\_FOG\_DENSITY, 0.5f); // 雾的稠密度

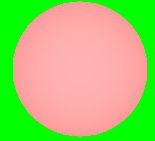
glPushMatrix();

glTranslatef(0.0f, 0.0f, -30.0f);

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

glutSolidSphere(5.0f, 50, 50);

glPopMatrix();



我们看到的球体边缘的像素点是(255,153,153)

实际上衰减因子=(50-30)/(50-0)=0.4

Final color = (1.0f, 0.0f, 0.0f) \* 0.4f + (1.0f, 1.0f, 1.0f) \* 0.6f

= (1.0f, 0.6f, 0.6f) = (255, 153, 153)

雾模式 雾方程式

GL\_LINEAR f = (end - c) / (end - start)

GL\_EXP f = exp(-d \* c)

GL\_EXP2 f = exp(-(d \* c) ^ 2)

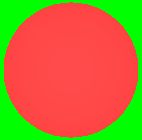
其中d表示稠密度，c表示距离

也就是说对于GL\_LINEAR而言，稠密度对最终颜色没有任何影响

但是对于GL\_EXP与GL\_EXP2而言，抽密度越大，衰减因子就越小，物体颜色所占比例就越小

glFogi(GL\_FOG\_MODE, GL\_EXP); // 采用哪种方程式

glFogf(GL\_FOG\_DENSITY, **0.01f**); // 雾的稠密度



glFogi(GL\_FOG\_MODE, GL\_EXP); // 采用哪种方程式

glFogf(GL\_FOG\_DENSITY, **0.05f**); // 雾的稠密度



注意：GL\_FOG\_START和GL\_FOG\_END仅对GL\_LINEAR雾有效，GL\_FOG\_DENSITY对于GL\_LINEAR雾无效

片断与眼睛的距离可以通过两种方法进行计算。有些实现（以NVIDIA硬件著称）将使用实际的片断深度。其他实现（多见于ATM芯片）使用顶点与顶点之间的插值。前者有时称为片断雾，后者有时称为顶点雾。片断雾比顶点雾更为复杂，但它长航具有更好的质量

雾坐标

我们并不一样要让opengl计算雾距离，也可以自己完成这种计算，这种值称为雾坐标。可以采用glFogCoordf函数手动设置。除非用下面这个函数更改了雾坐标的来源，否则雾坐标是被忽略的

glFogi(GL\_FOG\_COORD\_SRC, GL\_FOG\_COORD)

**累积缓冲区**

除了颜色、模板和深度缓冲区外，opengl还支持累积缓冲区。这个缓冲区允许在渲染到颜色缓冲区之后不是把结果显示到窗口，而是把颜色缓冲区的内容复制到累积缓冲区。Opengl所支持的一些复杂操作允许我们按照不同的方法将颜色缓冲区的内容与累积缓冲区的内容进行反复混合（这也是累积缓冲区名字的由来）。完成一幅图像的累积之后，就可以把累积缓冲区的内容复制回颜色缓冲区，并执行缓冲区交换以显示结果。

累积缓冲区是为合成多幅图像而设计的，它不是简单的用引入像素片元来代替像素值，而是将片元进行缩放，然后加到已有的像素值上。

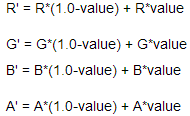
我们可以像其他缓存一样清空累积缓存，可以用glClearAccum()来设置红、绿和蓝色分量的清空值，按位顺序清空累积或以GL\_ACCUM\_BUFFER\_BIT调用，你不能直接渲染进累积缓存，而是应该渲染到一个选定的缓存，然后用glAccum来将在那缓存中的当前图像累积进累积缓存。glAccum用当前选择的读取缓存

累积缓存主要是为了辅助累积计算。例如：运动模糊和全局反走样

Void glAccum(GLenumm op, GLfloat value)

第一个参数指定了希望使用哪种累积操作，第二个参数是个浮点值，用于对操作进行缩放

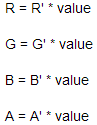
GL\_ACCUM：根据参数值对颜色缓冲区的值进行缩放，并将它们添加到累积缓冲区的当前 内容中



GL\_LOAD：根据参数值对颜色缓冲区的值进行缩放，并将它们替换累积缓冲区的当前内容



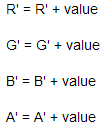
GL\_RETURN：根据累积缓冲区的值对颜色进行缩放，然后把结果复制到颜色缓冲区



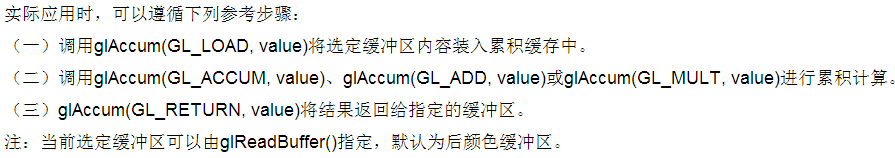
GL\_MULT：根据累积缓冲区的值对颜色进行缩放，然后把结果存储到累积缓冲区



GL\_ADD：根据累积缓冲区的值对颜色值进行缩放，然后把结果添加到累积缓冲区的当前内容上



由于累积缓冲区操作必须复制大量的内存并进行大量的处理，所以实时应用极少使用这个功能。对于非实时的渲染，opengl可以产生一些令人惊异的效果。



**抖动技术**

片断测试

1）裁剪测试

2）alpha测试

3）模板测试

4）深度测试

5）混合

6）抖动

7）逻辑操作

Opengl中的光照计算在哪个阶段进行？

如果采用Gouraud shading，光照计算在世界坐标系中进行；如果采用Phong shading，光照计算在屏幕坐标系中进行。Back-face culling是在观察坐标系中进行的。

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

固定管线整体变换：

1 顶点坐标---视觉坐标

2 视觉坐标---投影坐标

3 投影坐标---屏幕坐标

4 屏幕坐标---片元

5片元---像素

1）模型视图变换，即顶点变换

2）投影变换

3）透视除法、裁剪、隐藏表面消除

4）光栅化

5）片断测试

**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

**各向异性过滤**

Opengl的纹理过滤方式：

1 最邻近采样 GL\_NEAREST

2 线性纹理过滤（双线性过滤） GL\_LINEAR

3 mipmap纹理过滤（三线性过滤） GL\_LINEAR\_MIPMAP\_LINEAR

4 各向异性过滤

其中一个双线性过滤需要三次混合（四个顶点，三次混合）

一个三线性过滤需要七次混合（两次四个顶点就是六次混合，最后在加一次）

如果我们在纹理过滤的时候考虑了观察角度，那么这种过滤方法就称为各向异性过滤

<http://blog.csdn.net/kkk328/article/details/7055934/>

**I.纹理过滤:**

当三维空间里面的多边形经过坐标变换、投影、光栅化等过程，变成二维屏幕上的一组象素的时候，对每个象素需要到相应纹理图像中进行采样，这个过程就称为纹理过滤。  
  
**II.纹理过滤通常分为2种情况：**   
a) 纹理被缩小 GL\_TEXTURE\_MIN\_FILTER   
比如说一个8 x 8的纹理贴到一个平行于xy平面的正方形上，最后该正方形在屏幕上只占4 x 4的象素矩阵，这种情况下一个象素对应着多个纹理单元。  
b) 纹理被放大 GL\_TEXTURE\_MAG\_FILTER   
纹理被放大这种情况刚好跟上面相反，假如我们放大该正方形，最后正方形在屏幕上占了一个16 x 16的象素矩阵，这样就变成一个纹理单元对应着多个象素。  
  
**III.几种不同的纹理过滤方式：**   
1.最近点采样 GL\_NEAREST  
2.线性纹理过滤(双线性过滤)GL\_LINEAR  
3.mipmap纹理过滤(三线性过滤) GL\_LINEAR\_MIPMAP\_LINEAR  
4.各向异性过滤  
  
a) 最近点采样  
最近点采样,不进行任何过滤操作的速度最快也最简单，只是针对每一个象素对最接近它的纹理单元进行采样，可用于上面两种情况。但是这种纹理过滤方法的效果最差，在屏幕显示的图像会显得十分模糊。  
   
  
b)双线性过滤 Bilinear Interpolation   
线性过滤也比较简单，每个象素要对最接近它的2 x 2的纹理单元矩阵进行采样，取4个纹理单元的平均值，也可用于上面的两种情况。这种纹理过滤方法的效果比上面的要好很多。  
这是一种较好的材质影像插补的处理方式，会先找出最接近像素的四个图素，然后在它们之间作差补效果，最后产生的结果才会被贴到像素的位置上，这样不会看到“马赛克”现象。这种处理方式较适用于有一定景深的静态影像，不过无法提供最佳品质。其最大问题在于，当三维物体变得非常小时，一种被称为Depth Aliasing artifacts（深度赝样锯齿），也不适用于移动中的物件。  
   
  
c)三线性过滤 Trilinear Interpolation   
三线性过滤相对的比较复杂，它只能用于纹理被缩小的情况，需要先构造纹理图像的mipmap，mip的意思是“在狭窄的地方里的许多东西”，mipmap就是对最初的纹理图像构造的一系列分辨率减少并且预先过滤的纹理图。对于一个8 x 8的纹理来说需要为它构造4 x 4、2 x 2、1 x 1这三个mipmap。如果正方形被缩小到在屏幕上占6 x 6的象素矩阵，一个象素的采样过程就变成这样，首先是到8 x 8的纹理图中进行对最接近它2 x 2的纹理单元矩阵进行采样（也就是上面的线性过滤）；其次是到4 x 4的纹理图中重复上面的过程；接着把上面两次采样的结果进行加权平均，得到最后的采样数据。可以看出整个过程一共进行了三次的线性过滤，所以这种方法叫做三线性过滤，它的效果是三种纹理过滤方法里面最好的。  
这是一种更复杂材质影像插补处理方式，会用到相当多的材质影像，而每张的大小恰好会是另一张的四分之一。例如有一张材质影像是512×512个图素，第二张就会是256×256个图素，第三张就会是128×128个图素等等，总之最小的一张是1×1.凭借这些多重解析度的材质影像，当遇到景深极大的场景时（如飞行模拟），就能提供高品质的贴图效果。一个“双线过滤”需要三次混合，而“三线过滤”就得作七次混合处理，所以每个像素就需要多用21/3倍以上的计算时间。还需要两倍大的存储器时钟带宽。但是“三线过滤”可以提供最高的贴图品质，会去除材质的“闪烁”效果。对于需要动态物体或景深很大的场景应用方面而言，只有“三线过滤”才能提供可接受的材质品质。  
  
d)各向异性过滤 Anisotropic Interpolation   
各向异性过滤是最新型的过滤方法，它需要对映射点周围方形8个或更多的像素进行取样，获得平均值后映射到像素点上。对于许多3D加速卡来说，采用8个以上像素取样的各向异性过滤几乎是不可能的，因为它比三线性过滤需要更多的像素填充率。但是对于3D游戏来说，各向异性过滤则是很重要的一个功能，因为它可以使画面更加逼真，自然处理起来也比三线性过滤会更慢。  
各异向性纹理过滤不是单独使用而是和前面所述的其他过滤方法结合一起使用的。它在取样时候，会取8个甚至更多的像素来加以处理，所得到的质量最好。  
假设Px为纹理在x坐标方向上的缩放的比例因子；Py为纹理在y坐标方向上的缩放的比例因子；Pmax为 Px和Py中的最大值；Pmin为Px和Py中的最小值。当Pmax/Pmin等于1时，也就是说Px等于Py，纹理的缩放是各同向的；但是如果Pmax /Pmin不等于1而是大于1，Px不等于Py，也就是说纹理在x坐标方向和在y坐标方向缩放的比例不一样，纹理的缩放是各异向的，Pmax/Pmin代表了各异向的程度。  
举个例子来说，64 x 64的纹理贴到一个开始平行于xy平面的正方形上，但是正方形绕y轴旋转60度，最后投影到屏幕上占了16 x 32的象素矩阵。纹理在x坐标方向上缩放的比例因子为64/16等于4，在y坐标方向缩放的比例因子为64/32等于2，Pmax等于4，Pmin等于 2。缩放的各异向程度为2。当把各异向性过滤和线性过滤结合起来的时候，应该是对最接近象素的4 x 2的纹理单元矩阵采样才合理，因为一个象素在x坐标方向上对应了更多的纹理单元（Px > Py）。即使是纹理在一个轴方向上缩小而在另一个轴方向上放大，处理的过程也是一样的（注意的是如果纹理在一个轴方向上缩小而在另一个轴方向上放大，OpenGL仍然把它当作是纹理被缩小的情况，将采用为纹理缩小情况设置的过滤方法为基本过滤方法，然后再加上各异向性过滤）。假设被贴图的正方形最后在屏幕上占了一个128 x 32 的象素矩阵，纹理在x坐标方向上缩放的比例因子为64/128等于0.5，在y坐标方向缩放的比例因子为64/32等于2，由于Py > Px 且 Pmax/Pmin等于4，所以当把各异向性过滤和线性过滤结合起来的时候，应该对最接近象素的2 x 8的纹理单元矩阵进行采样。三线性过滤和各异向性过滤结合的过滤方法的步骤跟前面单独的三线性过滤方法大致是一样的，只是前面两步采用了各异向性过滤和线性过滤结合的方法。  
通常情况下采取线性过滤或者三线性过滤就可以得不错的效果，但是在某些特殊的情况下，特别是把一个都是线状条纹的纹理图贴到一个绕x或者是y轴旋转角度很大的多边形上的时候，比如将人的头发纹理贴到构成人的头顶的多边形，即使是三线性过滤的效果也不能令人满意，只有将各异向过滤方法和三线性过滤或者线性过滤的方法结合起来才能得到完美的效果。

**opengl光照模型与面绘制算法**

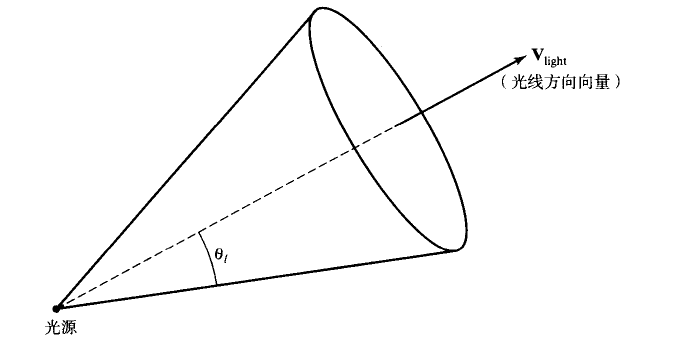
光照模型主要用于对象表面某光照位置的颜色计算。表面绘制算法是使用光照模型为对象的所有投影位置确定像素颜色

1）光源：任一发出辐射能量的对象称为一个光源，大多数的发光体仅作为场景中的光照源，但是在有些应用中，我们可能希望建立既是光源又是光反射体的对象。比如，在一个塑料球内放置一个灯泡，这样在球表面上既发光也反射光

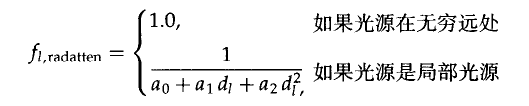
点光源：通过给出其位置和发射光颜色来定义

无穷远光源：使用一个颜色值和从该光源发出的一个固定光线方向来模拟一个无穷远距离光源，在光照计算中，仅需要一个发射方向的向量以光源颜色而不需要光源位置

方向光源：除了位置和颜色之外还需要确定一个向量方向和从该向量方向开始的角度范围



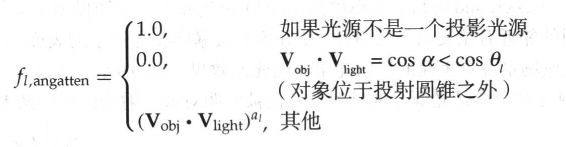
辐射强度衰减：衰减因子为1/(a0+a1\*d+a2\*d^2),但是对于无穷远处的光源不能使用强度衰减来计算，因为场景中的各点离很远的光源位置几乎都相等，因此：



角强度衰减：对于一个方向光源，可以按照从点光源位置出发的光强度角计算衰减，即可模拟一个光锥，沿其圆锥轴光强最大，离开圆锥轴时强度逐渐减弱，常用的方向光源角强度衰减函数是：



这种情况下还需要考虑几个特殊情况，如果光源不是方向光源（即不是投影光源）则没有角强度衰减，同样，如果一个对象位于投影光锥体外的任何地方，则它得不到该光源的光照:



表面光照效果：光照模型使用使用为表面设定的各种光学特性来计算表面的光照效果，这些特性包括透明度、颜色发射系数以及各种表面纹理参数

当光入射到不透明表面时，一部分被发射而一部分被吸收，入射光中北表面反射的总量依赖于材料类型，光亮的表面将大部分的入射光反射，而惠安的表面吸收较多的入射光，对于透明的表面，入射光的补一份穿过该材料

漫反射：对象的漫反射光的颜色称为该对象的颜色。例如，蓝色对象将白色光中的蓝色部分反射而将其他成分吸收，如果蓝色对象在红色光下观察，则它表现为黑色，因为所有入射光全部被吸收

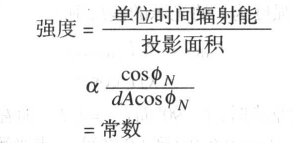
基本光照模型：

精确的光照模型按照入射光能量与对象的材料组成之间的交互结果进行计算，为了简化表面光计算，可使用前面泰伦的光照效果的物理过程近似表示，基本光照模型中的发光体一般限于点光源，然而，许多图形软件提供处理方向光源和扩展光源的函数。

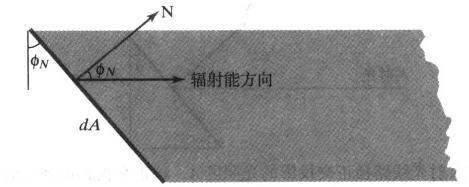
**Lambert漫反射模型**

该模型属于经验模型，主要用来简单模拟粗糙物体表面的光照现象。

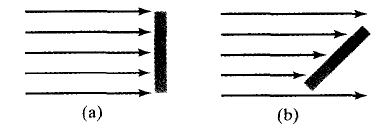
该定律表明：在与对象表面法向量夹角为¢N的方向上，每个面积为dA的平面单位所发散的光线与cos¢N成正比。该方向的光强度可用单位时间辐射能总量除以表面积在辐射方向的投影来计算



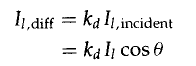
这样，对于朗伯反射，光强度在所有观察方向都相同

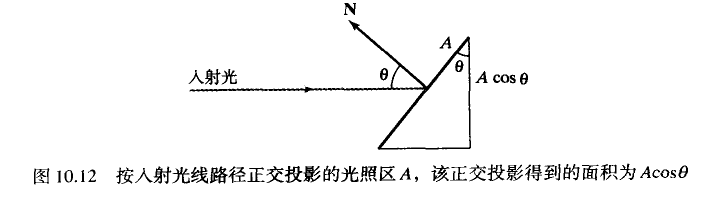


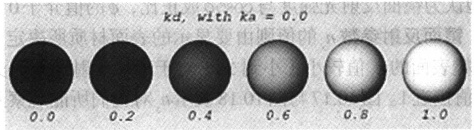
与表面相交的光线数与该表面投影到入射光方向的面积成正比，如下图：



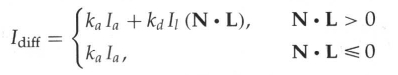
如果把入射光方向与表面法线方向之间的入射角称为θ，则垂直于光线方向的面片投影面积与cosθ成正比，因此一个强度为I的光源入射光总量为

 Kd数与0-1之间的常数





组合环境光和点光源的强度计算，可以得到一个的表面位置的全部漫反射表达。另外，许多图形软件包引入为每一表面指定的环境光发射系数Ka来修改环境光强度Ia，这使得我们能够简单地利用该参数来调节场景中的光照效果。使用参数Ka，可将单个点光源的全部漫反射公式表示为：



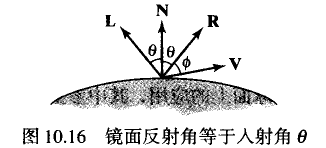
其中Ka和Kd都依赖于表面材质，对单色光照效果而言其值介于0和1之间



把环境光模型添加进来，最后，Lambert光照模型可写为：  
I= IaKa + Id Kdcosθ= IaKa + Id Kd(L·N)  
该模型包含环境光和漫反射光

**Phong镜面反射模型**

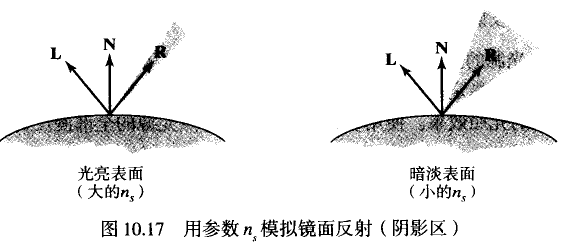
在光滑表面上看到的高光或镜面反射是由接近镜面反射角的一个会聚区域内入射光的全部或绝大部分称为反射光所导致。



对于一个理想的反射体（镜子），入射光仅在镜面反射方向有反射现象，此时仅当V和R重合时才能观察到反射光线（即¢=0）

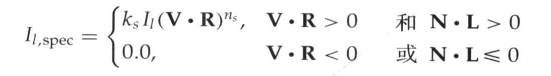
非理想反射体系统的镜面发射方向分布在向量R周围的有限范围内。较光滑表面的镜面反射范围较小，而粗糙的对象表面则有较大的镜面反射范围。

Phong Bui Tuong提出了镜面反射模型，该模型认为镜面反射光照强度与成正比，¢介于0-90之间。**镜面发射参数ns的值则由要显示的表面材质所决定。**光滑表面的ns较大（如100或更大），而粗糙表面的ns较小（小到1）。对于理想反射器ns是无限的。而粗糙对象表面。如粉笔或煤渣的ns值接近1





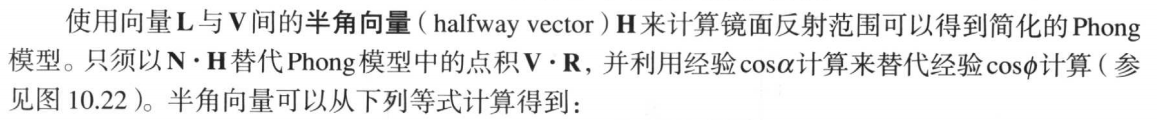


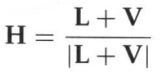


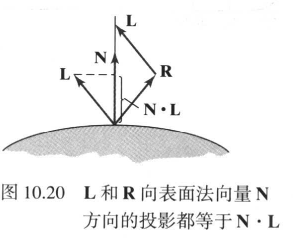
反射向量R的方向可通过向量L和N计算出来，可以通过点积N.L得到向量L在法向量方向上的投影，这也等于单位向量R在N方向上的投影

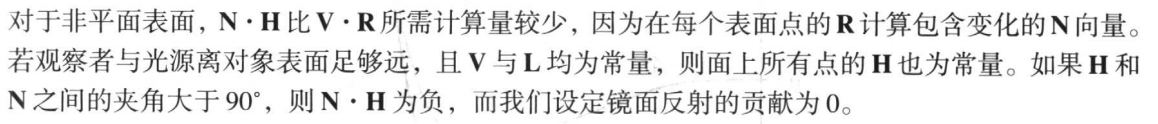


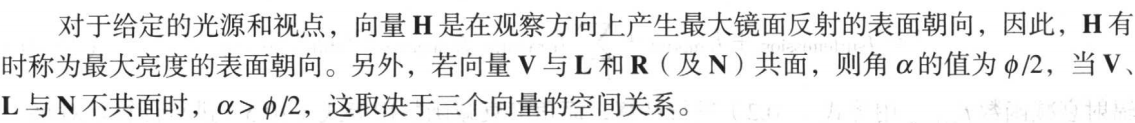












Lambert和Phong两个模型经常同时使用，因为在现实世界中，任何表面都会同时发生漫反射和镜面反射两种现象，因此我们就要使用两种模型分别计算两种发射后的光强（也就是顶点颜色值），使渲染的效率看起来真实一些，但是注意，这样做并不会带来真正真实的效果，毕竟这两种模型都是经验模型，考虑的都是在理想情况下。而Blinn-phong光照模型是基于Phong的修正模型。相比于Phong模型，Blinn-phong模型只是用N.H替代了V.R，但却获得了明显的提高，它能够提供比Phong更柔和。更平滑的高光，而且速度上更快，因此成为许多CG软件中默认的光照渲染方法，同时也被集成到大多数的图形芯片中，而且Opengl和DirecX 3D的渲染管线中，它也是默认的光照模型

漫反射和镜面发射合并：



Blinn-phong光照模型省去了计算发射光线方向向量的两个乘法运算，速度更快

**抖动**

**齐次坐标**

所谓齐次坐标，就是用n+1个分量来表示n维坐标

为什么要引入齐次坐标？

许多图形应用涉及到几何变换，主要包括平移、旋转、缩放，以矩阵表达式来计算这些矩阵时，平移矩阵时相加的，旋转和缩放则是矩阵相乘的，将这些变换综合起来可以表示为

P1=P\*M1+M2，由于多次使用会效率很低，因此希望转化为P1=P\*M的形式，因而产生了齐次坐标

**3D图形学的学习策略**

<http://www.docin.com/p-218474381.html>

**OpenGL中的MVP矩阵**

<http://www.tuicool.com/articles/2IJvQfj>

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

二维坐标系之间的变换：

下图给出了一个在笛卡尔坐标系统xy中用坐标原点（x0，y0）及方向角θ指定的笛卡尔坐标系统x'y'，为了将对象描述从xy坐标变换到x'y'坐标，必须建立把x'y' 轴叠加到xy轴的变换，这需要分两步进行（平移和旋转都已经证明）：

1）将x'y'系统的坐标原点（x0，y0）平移到xy系统的原点（0,0）

2）将x'轴旋转到x轴上

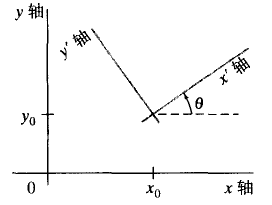
即点从A坐标系转换到B坐标系：

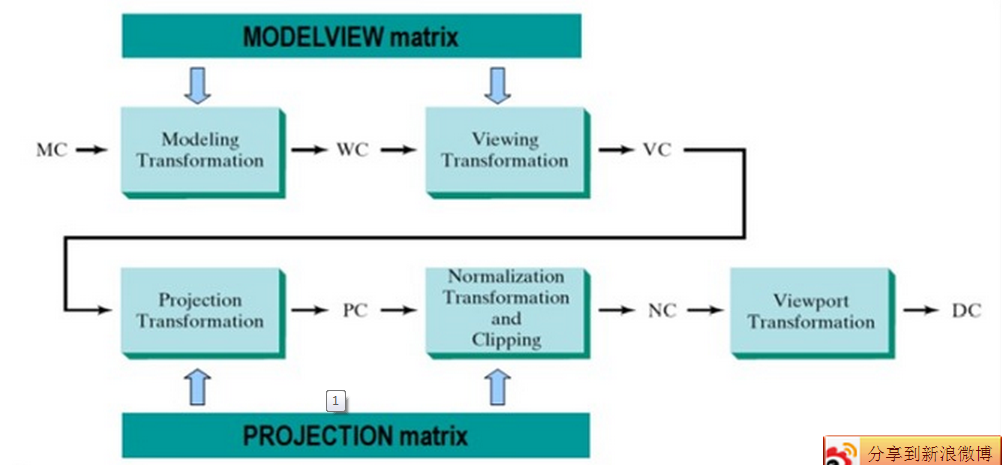
将B坐标系变到与A坐标系重合，比如下图中从x'y'坐标系转换到xy坐标系，则先将xy坐标系平移，然后旋转，即T(x0,y0)\*R(θ)\*v(x,y)，也就是将x'y'坐标系中与xy对应的点先R(θ)然后T(x0,y0)，这一切的变换都是在xy坐标系上做的

为了使观察处理独立与输出设备，图形系统将对象描述转换到规范化设备坐标系并提供裁剪程序。有些系统的规范化设备坐标范围是0到1，而有些是-1到1，裁剪工作通常在规范化坐标系中进行。这使得我们可以在此之前合并变换矩阵从而减少计算时间

应用程序要得到特殊的裁剪效果，可通过选择裁剪窗口的不同形状、大小和方向来实现、例如，可以使用椭圆、星型等形状作为裁剪窗口，但是使用凹多边形或非线性边界裁剪窗口来比用矩形裁剪要花费更多的时间。确定对象与圆的焦点比确定它与直线的交点需要更多的计算。最简单的用户裁剪的窗口边界是用坐标轴平行的直线段。因此，图形软件一般仅允许使用平行于x和y轴的矩形裁剪窗口。

世界坐标系裁剪窗口，图形系统程序库中一般均提供定义标准矩形裁剪窗口的函数。我们可以指定世界坐标系中的两点作为标准矩形的两个对角顶点。一旦建立了裁剪窗口，观察函数就对场景描述进行处理并将结果送到输出设备



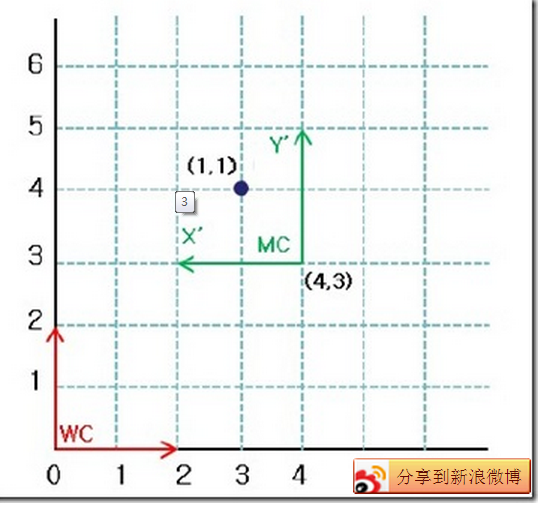
****

1）模型视图变换

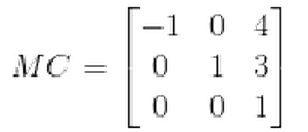
图中显示了opengl图形管线的主要部分，也是我们进行图形编程时常常要用到的部分。一个顶点数据从左上角（MC）进行管线，最后从图的右下角（DC）输出。MC是Model Coordinate的简写，表示模型坐标，DC是Device Coordinate的简写，表示设备坐标。当然DC有很多，比如显示器、打印机等，这里的坐标我们理解为屏幕坐标。MC是3D坐标（注意：这里所说的是3D坐标，而不是世界坐标），这个3D坐标就是模型坐标，也说成本地坐标（相对于世界坐标）。MC要经过模型变换（Modeling Transformation）才变换到世界坐标。

变换到世界坐标WC（World Coordinate）简单点说就是如何用世界坐标系来表示本地坐标系中的坐标

举例：

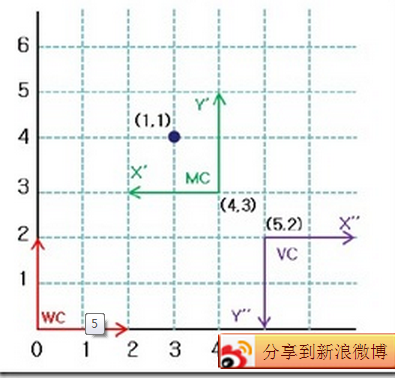


上图中红色的是世界坐标（WC），绿色的是模型坐标（MC）,首先要将模型坐标系MC在世界坐标系中表示出来，得到的变换矩阵为：（先将xy坐标系仿射，后平移，参考上面的二维坐标系之间的变换）



其中，矩阵的第一列为MC中的X轴在WC中的向量表示，第二列为MC中Y轴在WC中的向量表示，第三列为MC中的原点在WC中的表示

现在我们将相机坐标也加进去，相机坐标也成为观测坐标（View Coordinate）

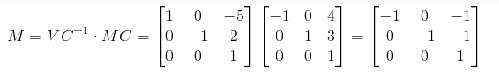


可以得到世界坐标转化为眼坐标的矩阵为（先将x''y''平移，后仿射）：



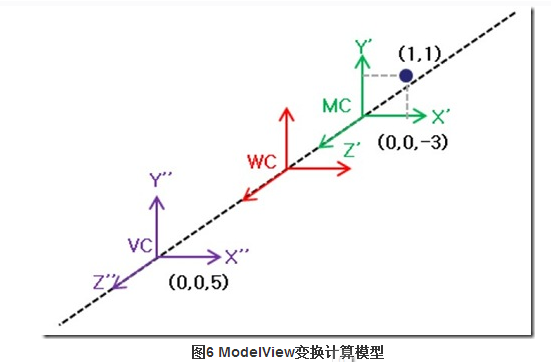
**记住一点：所有的平移、旋转、放缩都必须基于世界坐标做变化**

在opengl中，当我们申明顶点的时候，有时候说的是世界坐标，这是因为初始化的时候世界坐标系、模型坐标系和相机坐标系是一样的，重合在一起。所以，opengl中提供了模型观测变换，它是把模型坐标系直接转换为相机坐标系，在上面的例子中，就是将上面的两个矩阵相乘。这就是opengl中的ModelView变换矩阵。这也是ModelView变换的名字的由来，就是通过场面的两个步骤得到的。

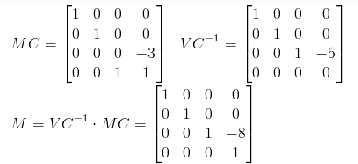


现在只要用上面的模型观测矩阵M乘以模型坐标系MC中的坐标就可以得到相机坐标系中的坐标了。模型观测变换的关键就是要得到相机坐标系中的坐标，**因为光照等计算都是在这个坐标系中完成的**

以下图为例进行验证：



通过上图可以得出



void testModelViewMatrix()

{

GLfloat matrix[16] = {0};

int size = GET\_ARRAY\_LENGTH(matrix);

glMatrixMode(GL\_MODELVIEW\_MATRIX);

glLoadIdentity();

glGetFloatv(GL\_MODELVIEW\_MATRIX, matrix);

showMatrix(matrix, size);

gluLookAt(0.0f, 0.0f, 5.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);

glGetFloatv(GL\_MODELVIEW\_MATRIX, matrix);

showMatrix(matrix, size);

glTranslatef(0.0f, 0.0f, -3.0f);

glGetFloatv(GL\_MODELVIEW\_MATRIX, matrix);

showMatrix(matrix, size);

}



到这里，整个ModelView变换就完成了，通过ModelView变换后得到就是相机坐标系中的坐标。在这个坐标系内典型的计算就是法线了。

实际上也就是project\*view\*model\*vertex

在默认的情况下，glulookAt位于原点，也就是一个单位矩阵，model矩阵也是一个单位矩阵，因此最初的模型视图矩阵也可以看做是View\*Model，接下来我们要先设置View，在设置Model，实验证明如果将两者调用顺序颠倒，所得的模型视图矩阵不一样，最后得到的效果也不一样

**坐标变换：**

<http://www.niubb.net/a/2015/04-29/329440.html>

<http://www.360doc.com/content/14/1028/09/19175681_420513219.shtml>

左右手坐标系：大拇指-x轴 食指-y轴 中指-z轴

Opengl 右手坐标系 direct 左手坐标系

**glLookAt矩阵推导**

<http://blog.csdn.net/augusdi/article/details/20450065>

<http://www.360doc.com/content/14/0822/11/13726687_403784935.shtml>

把物体从世界坐标系转换到视点坐标系的矩阵称为视图矩阵

gluLookAt通过指定一个视点、表面场景中的参考点以及up向量来构造一个视图变换矩阵，这个矩阵将代表场景中心的参考点映射到-Z轴，视点映射为原点。视图变换就是在相机坐标系下解释世界坐标系下的点，而这个变换矩阵，可以看做是将相机坐标系变换到与原来的世界坐标系

重点理解uvn坐标系统

Libgdx的矩阵推导

**public** Matrix4 setToLookAt (Vector3 position, Vector3 target, Vector3 up) {

*tmpVec*.set(target).sub(position);

setToLookAt(*tmpVec*, up);

**this**.mul(*tmpMat*.setToTranslation(position.tmp().mul(-1)));

**return** **this**;

}

**public** Matrix4 setToLookAt (Vector3 direction, Vector3 up) {

*l\_vez*.set(direction).nor();

*l\_vex*.set(direction).nor();

*l\_vex*.crs(up).nor();

*l\_vey*.set(*l\_vex*).crs(*l\_vez*).nor();

idt();

val[*M00*] = *l\_vex*.x;

val[*M01*] = *l\_vex*.y;

val[*M02*] = *l\_vex*.z;

val[*M10*] = *l\_vey*.x;

val[*M11*] = *l\_vey*.y;

val[*M12*] = *l\_vey*.z;

val[*M20*] = -*l\_vez*.x;

val[*M21*] = -*l\_vez*.y;

val[*M22*] = -*l\_vez*.z;

**return** **this**;

}

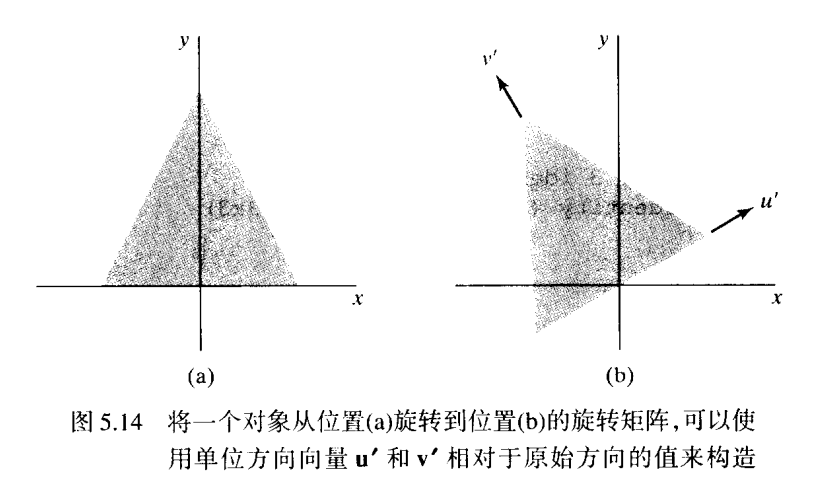
上面的公式在up={-(float)Math.sqrt(2.0f)/2, (float)Math.sqrt(2.0f)/2,0.0f}的情况下已经验证是正确的，实际上是先求得了一个旋转矩阵，然后再与一个平移矩阵相乘的结果，l\_vex,l\_vey,l\_vez实际上是观看位置的三维笛卡尔坐标系，注意坐标系的得到是先确定z轴（通过direction方向得到），在确定x轴（direction与up叉乘得到，右手法则），最后确定y轴（x轴与z轴叉乘得到，右手法则）

三维空间的旋转，绕z轴旋转很好求，接下来的绕x和y轴旋转可以通过轴的取代获取：

绕x轴，则之前的z轴由x轴取代，之前的x轴由-z取代

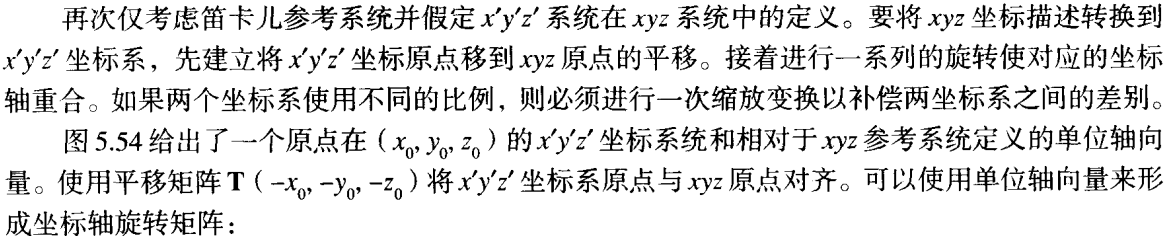
绕y轴，则之前的z轴由y轴取代，之前的y轴由x取代，之前的x轴由z轴取代

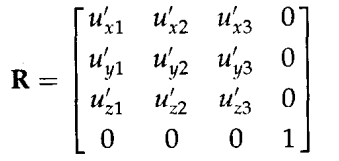
构造二维旋转矩阵（引用计算机图形学的原话）

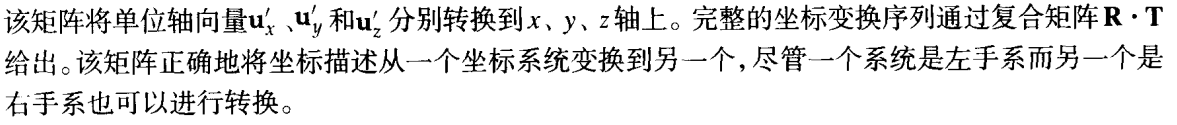


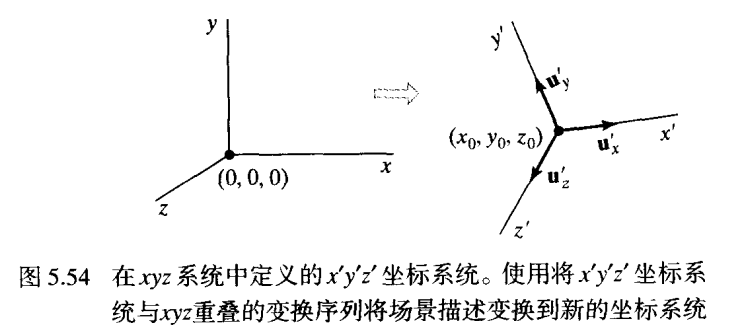
我们可以将u'的元素位置设置给旋转矩阵的第一行，将v'的元素设置给旋转矩阵的第二行，从而构造所期望的变换。任何旋转矩阵的元素可以表示为一组正交单位向量的元素

由于只有正弦函数受到旋转角符号变化的影响，因此逆矩阵也可以通过交换原矩阵中的行和列来得到，即可以利用旋转矩阵R的转置来取代逆矩阵计算（R^-1=R^T）









用过渡矩阵M乘以基向量(1,0,0)^T，发现得到的结果是过度矩阵M的第一列

用过渡矩阵M乘以基向量(0,1,0)^T，发现得到的结果是过度矩阵M的第二列

用过渡矩阵M乘以基向量(0,0,1)^T，发现得到的结果是过度矩阵M的第三列

这是一个关键的发现：矩阵的每一列都能解释为转换后的基向量

这个强有力的概念有两条重要性质：

1 有了一种简单的方法来形象化解释矩阵所代表的变换

2 有了反向建立矩阵的可能---给出一个期望的变换（如旋转、缩放等），能够构造一个矩阵代表此变换，我们所有做的一切就是计算基向量的变换，然后将变换后的基向量填入矩阵中

glMatrixMode(GL\_MODELVIEW)该语句指定了一个4\*4建模观察矩阵作为当前矩阵，使用glMatirxMode函数还可以设定另外另个模式：纹理模式和颜色模式。纹理模式用于映射表面的纹理图案，而颜色模式用于从一个颜色模型转换到另外一个颜色模型。glMatrixMode函数的默认变量是GL\_MODELVIEW

opengl有两个函数用来处理栈中的矩阵，这些栈处理函数比单独管理栈矩阵效率要高，尤其是在栈函数使用硬件实现时。例如，硬件实现可同时复制多个矩阵元素，而我们可以在栈中保留一个单位矩阵，从而在当前矩阵初始化时实现得比调用glLoadIdentity更快

这两个函数是glPushMatrix与glPopMatrix

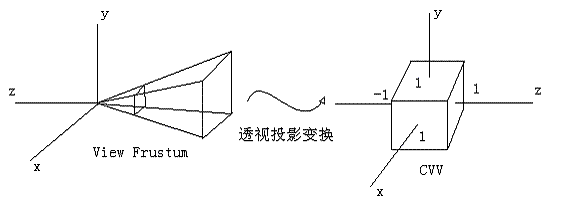
**2）投影变换矩阵推导**

<http://blog.csdn.net/lyx2007825/article/details/8792475>

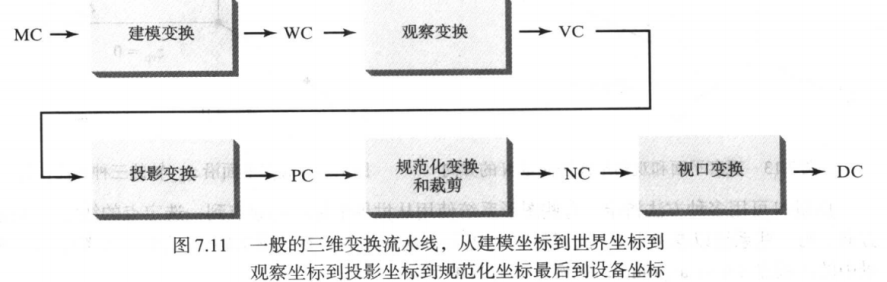
http://blog.csdn.net/popy007/article/details/1797121

经过相机矩阵的变换，顶点被变换到了相机空间，这个时候的多边形也许会被视锥体裁剪，但是在这个不规则体中进行裁剪并非那么容易，所以经过图形学前辈们的精心分析，裁剪被安排到规则观察体（Cannonical View Volume，CVV）中进行，CVV是一个正方体，xyz的范围都是-1到1，多边形的裁剪就是用这个规范体完成的。所以，事实上是头是投影变换由两步组成：

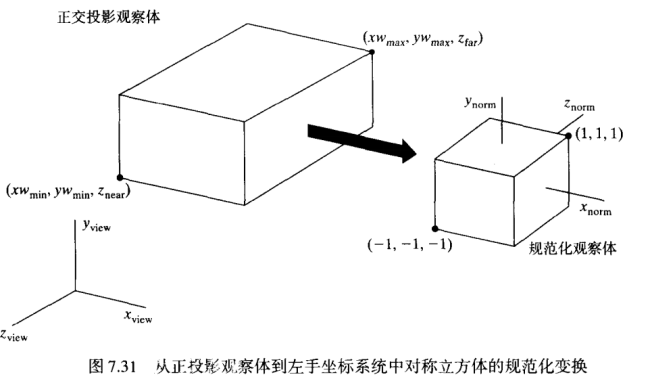
1. 用透视变换矩阵把顶点从视锥体中变换到建材空间的CVV中
2. CVV裁剪完成后进行透视除法

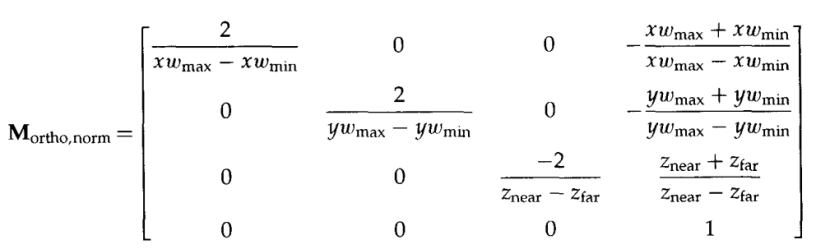


完成向规范化投影坐标的转换后，可高效地对对称立方体（或单位立方体）进行裁剪。裁剪以后，规范化观察体的内容转变到屏幕坐标系，对于x和y的变换和之前的等比变换相同，但三维观察体的所有位置还有一个深度，我们需要将深度信息留给可见性检测和表面绘制时使用。因此，现在可以把视口变换看成向三维屏幕坐标系的映射。



裁剪操作可以在所有与设备无关的坐标变换（从世界坐标系到规范化坐标系）完成之后进行，这样，坐标变换可以合并以便最大限度的提高效率





在规范化变换之后可高效地进行裁剪处理，在观察流水线的这个阶段，所有与设备无关的变换均已完成，并且已经合并到一个组合矩阵中。裁剪后，再应用可见性测试，表面绘制和视视口变换来生成最后的场景屏幕显示

Libgdx的实现：

**public** **void** update (**boolean** updateFrustum) {

projection.setToOrtho(zoom \* -viewportWidth / 2, zoom \* viewportWidth / 2, zoom \* -viewportHeight / 2, zoom

\* viewportHeight / 2, Math.*abs*(near), Math.*abs*(far));

view.setToLookAt(position, tmp.set(position).add(direction), up);

combined.set(projection);

Matrix4.*mul*(combined.val, view.val);

**if** (updateFrustum) {

invProjectionView.set(combined);

Matrix4.*inv*(invProjectionView.val);

frustum.update(invProjectionView);

}

}

获取正交投影归一化矩阵

**public** Matrix4 setToOrtho (**float** left, **float** right, **float** bottom, **float** top, **float** near, **float** far) {

**this**.idt();

**float** x\_orth = 2 / (right - left);

**float** y\_orth = 2 / (top - bottom);

**float** z\_orth = -2 / (far - near);

**float** tx = -(right + left) / (right - left);

**float** ty = -(top + bottom) / (top - bottom);

**float** tz = -(far + near) / (far - near);

val[*M00*] = x\_orth;

val[*M10*] = 0;

val[*M20*] = 0;

val[*M30*] = 0;

val[*M01*] = 0;

val[*M11*] = y\_orth;

val[*M21*] = 0;

val[*M31*] = 0;

val[*M02*] = 0;

val[*M12*] = 0;

val[*M22*] = z\_orth;

val[*M32*] = 0;

val[*M03*] = tx;

val[*M13*] = ty;

val[*M23*] = tz;

val[*M33*] = 1;

**return** **this**;

}

**透视投影矩阵**

即将每一个棱锥体的每个切面都转换到正方体中，可以看做平行于眼坐标切一个面做映射

也可以看作先投影了再做规范化，映射到CVV中（注意：透视投影的裁剪是在齐次坐标系下进行的，也就是说将投影的齐次坐标在CVV立方体中进行裁剪）

先映射到观察面（取近平面），设原始点为p(x,y,z)，投影点为p1(x1,y1,z1)（其实这里的投影点x1,y1,z1会变为齐次坐标）通过三角形的相似性我们可以得到:

x1=-x\*near/z y1=-y\*near/z z1=-near

由于z值以后要进行深度测试，故可以将z1取为之前的z，即z1=z，z值与x和y无关

此时可以将x1,y1,z1用齐次坐标式表示为

near 0 0 0

0 near 0 0 \* (x,y,z,1)^T

0 0 A B

0 0 -1 0

因此可以得到：（注意是除以-z之后映射到了-1-1，而不是直接将near与far映射到-1-1,也就是真实的坐标点要映射到-1-1），**也就是假设了AZ+B已经是z的齐次坐标了，然后归一化，在x和y的求解过程中，是转换为正常坐标-归一化坐标-齐次坐标，而z的求解是假设了齐次坐标，然后将其进行归一化**

z1 = (A\*z+B)/(-z)

此时将z轴方向映射到-1到1

则当z=-near时，z1=-1 当z=-far时，z1=1 求解得到：

A=(near+far)/(near-far) B=(2\*near\*far)/(near-far)

但此时x和y还没有归一化到-1到1之间

即(near\*x-left)/(right-left)=(x2+1)/2

得到x2=(2\*near\*x-right-left)/(right-left)

同理得到y2=(2\*near\*y-top-bottom)/(top-bottom)

最终得到的矩阵为：

2\*near/(right-left) 0 -(right+left)/(right-left) 0

0 2\*near/(top-bottom) -(top+bottom)/(top-bottom) 0

0 0 (near+far)/(near-far) (2\*near\*far)/(near-far)

0 0 -1 0

由常见的公式可得：

Left=-near\*aspect\*tan(a/2)

Right=near\*aspect\*tan(a/2)

bottom=-near\*tan(a/2)

top= near\*tan(a/2)

带入得到：

Cot(a/2)/aspect 0 0 0

0 Cot(a/2) 0 0

0 0 (near+far)/(near-far) (2\*near\*far)/(near-far)

0 0 -1 0

Libgdx的实现

**public** Matrix4 setToProjection (**float** near, **float** far, **float** fov, **float** aspectRatio) {

idt();

**float** l\_fd = (**float**)(1.0 / Math.*tan*((fov \* (Math.*PI* / 180)) / 2.0));

**float** l\_a1 = (far + near) / (near - far);

**float** l\_a2 = (2 \* far \* near) / (near - far);

val[*M00*] = l\_fd / aspectRatio;

val[*M10*] = 0;

val[*M20*] = 0;

val[*M30*] = 0;

val[*M01*] = 0;

val[*M11*] = l\_fd;

val[*M21*] = 0;

val[*M31*] = 0;

val[*M02*] = 0;

val[*M12*] = 0;

val[*M22*] = l\_a1;

val[*M32*] = -1;

val[*M03*] = 0;

val[*M13*] = 0;

val[*M23*] = l\_a2;

val[*M33*] = 0;

**return** **this**;

}

**另外一种理解推导方式（关于x和y）**

首先我们通过映射关系可以得到正常坐标x1=-near\*x/z y1=-near\*y/z

接下来我们需要将正常坐标x1和y1映射到-1-1区域中

即(x1-left)/(right-left)=(x2+1)/2 (x2为我们所求的归一化后的坐标)

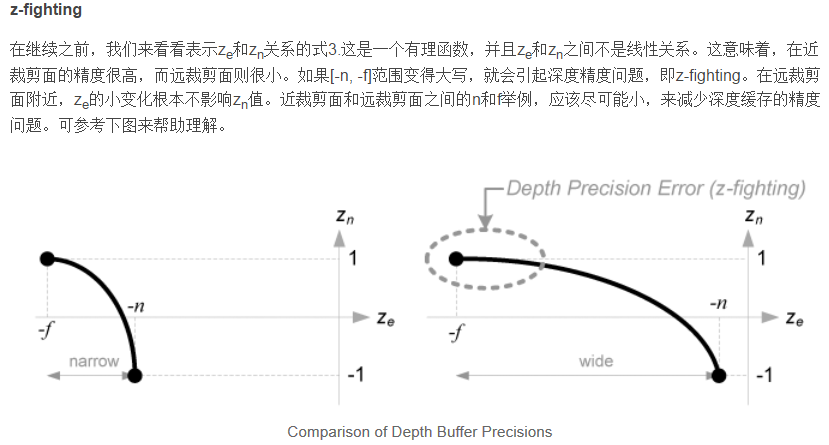
得到x2=(-2\*near)\*x/(-z)\*(right-left)-(right+left)/(right-left);

然后将x2转换为齐次坐标，也就是用-z乘以x2，则可以得到变换矩阵的第一行为：

(-2\*near/(right-left) 0 -(right+left)/(right-left) 0)

**一个问题:Z缓冲冲突**

由于深度缓冲的精度问题，当两个相聚特别近的顶点在投影映射后会出现两者渲染遮挡关系错误的情况，



解决该问题的方法：

<http://blog.csdn.net/iaccepted/article/details/44873517>

opengl中遮挡关系是通过z缓冲区+深度测试来实现的，而z缓冲区一般都有位数限制，通常为24位，也有32位。

这样在进行遮挡查询的时候，如果两个面距离非常近的话，那么这两个面的z值就可能会由于非常小而产生交替出现的现象，称为z-fighting，这就严重影响渲染质量。

通常的解决方法：

1.最常用的就是使用glPolygonOffset函数，对于两个非常近的面或者model我们可以将其分开渲染，先渲染一个然后通过glPolygonOffset函数设置一个偏移之后再渲染另一个，

   这样的结果就是在一个的深度值写入深度缓冲之后，另一个可以加一个偏移量来保证在深度测试的时候两者的深度值达到明显区别的程度，从而正确的绘制两个object。

2.通常我们在计算depth的时候并不使用线性计算，即计算出的depth并非是点的z坐标线性转换而来，而是使用非线性变换，Fdespth = （1/z - 1/near) / (1/far - 1/near),这样就能保证在离近切面近的部分depth值更精确，当然我们也很容易推出，当把near值适当增大时，可以更好的保证距离较远处的精度值，所以可以通过增大near值来解决，但是这种方法对于解决单模型渲染还是效果不错的，但是对于多模型渲染也会有问题。

3.提升硬件精度，主要是提升z缓冲区的位数来达到增加精度的目的，这对于要求较高的场合确实是很好的选择。

**万向节锁**

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

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

3D数学基础：图形与游戏开发

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

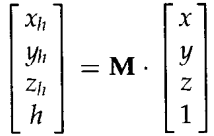
这两个变换都是通过透视矩阵一次完成的，顶点进入齐次裁剪空间并经过CVV裁剪，最终进行透视除法从4D齐次形式变回3D形式

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：



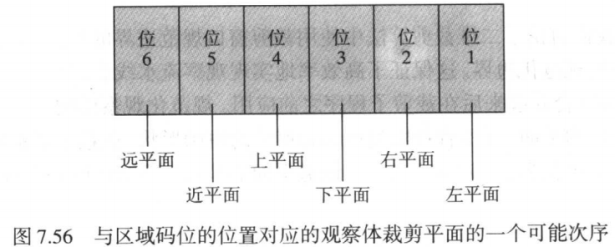
这里的M表示从世界坐标到规范化、齐次投影坐标的所有各种变换的组合，而齐次参数h可能不再是1。事实上，h可以有任意实数值，依赖于我们如何在场景中表达对象及使用的投影类型。

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。

三维裁剪：

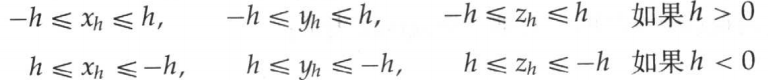
1）点和线段

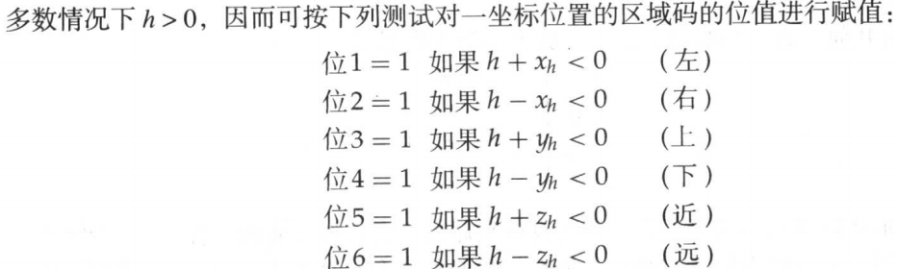


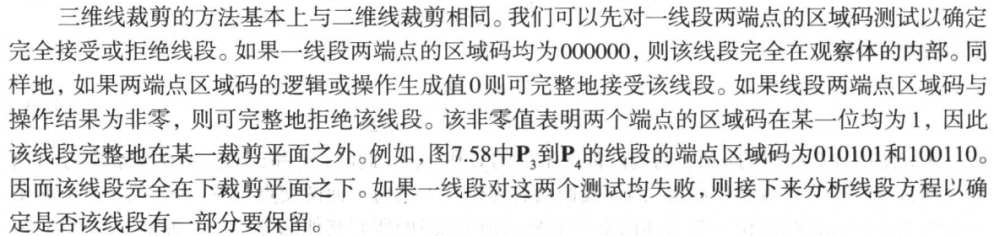
上述方法与之前的二维裁剪一样，只是加上了用于近和远裁剪平面的两个附加条件。然而，对于一个三维场景，我们需对变换到规范化空间的投影坐标使用裁剪子程序。在投影变换后，场景每一个四元素表示。假设用规范化对称立方体的边界进行裁剪，则当一点的投影坐标满足如下六个不等式时该店位于规范化观察体的内部



该齐次参数h不会为0，除非出现错误，但是我们可以再执行区域码过程之前先检查齐次参数为0或一个太小的数的可能性。齐次参数也可以为正或负，因此，假设h!=0，可以将前面的不等式写成下列形式



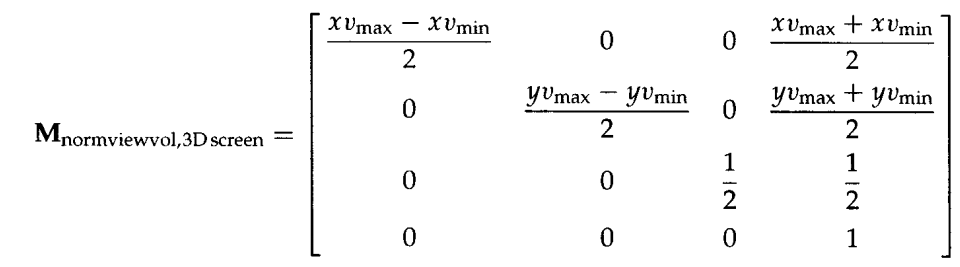




2）多边形

视口变换

完成向规范化投影坐标的变换后，可高效地对对称立方体（或单位立方体）进行裁剪。裁剪以后，规范化观察提的内容转变到屏幕坐标系。对于规范化的裁剪窗口中的x和y位置，



(x+1)/2 = (x'-xvmin)/(xvmax-xvmin)

关键在于z值的转换，由-1-1的区间变为0-1的区间

(z+1)/2=z'

视口的每一个xy位置与刷新缓存中的一个位置相对应，该位置包含了屏幕上这一点的颜色信息。而每一屏幕点的深度值存放在称为深度缓存的另一个缓存区。

二维裁剪算法

1）点的裁剪

2）线段的裁剪（直线段）

3）区域的裁剪（多边形）

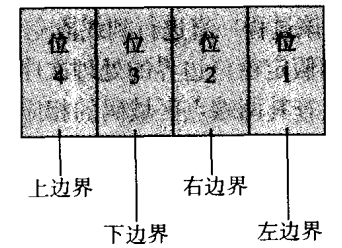
4）曲线的裁剪

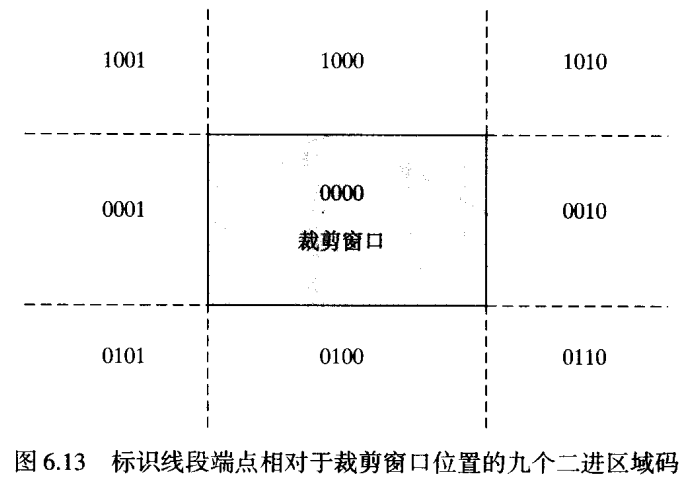
5）文字的裁剪

除非特别声明，我们都假设裁剪区域是一个正则矩形，这些边界与从0到1或从-1到1的规范化正方形的边界对应

线段裁剪方法：

（1）Cohen-Sutherland线段方法





一旦给所有的线段端点建立了区域码，就可以快速判断哪条线段完全在裁剪窗口之内，哪条线段完全在窗口之外。完全在窗口边界内的线段，其两个端点的区域码均为0000，因此保留了这些线段。两个端点的区域码中，有一对相同位置都是1的线段则完全落在裁剪矩形之外，因此丢弃这些线段

如果两个端点的区域码进行逻辑或的结果为0000，则线段完全位于裁剪区域之内

如果两个端点的区域码进行逻辑与的结果不为0000，则线段完全位于裁剪区域之外

对于不能判断为完全在窗口外或窗口内的线段，则要测试其与窗口边界的交点。

**关于libgdx的mesh**

\* A Mesh holds vertices composed of attributes specified by a {@link VertexAttributes} instance. The vertices are held either in

\* VRAM in form of vertex buffer objects or in RAM in form of vertex arrays. The former variant is more performant and is prefered

\* over vertex arrays if hardware supports it.

VRAM显存 GPU读取 RAM内存 CPU读取

顶点数组的数据是存储在内存中的，顶点缓冲对象的数据是存在显存中的

顶点数组渲染时数据从内存送入渲染管线，而VBO则从显存送入渲染管线

（该链接很重要）

<http://blog.csdn.net/zhanghua1816/article/details/23121735>

从上面的渲染管线中可以看出，在投影变换中也分为两个部分，第一部分是将上个阶段得到的**坐标转换为平面坐标**，第二个部分是将转换后的**平面坐标在进行归一化并进行裁剪**。一般地，将三维坐标转换为平面坐标有正交投影和透视投影

1.正交投影

采用函数glOrtho，该函数可以设置正交投影的投影空间，在该空间以外的坐标点就不会被投影到投影平面上，函数中的六个参数分别是投影空间六个平面，如下图：



在上图中，大的投影空间是根据这六个参数设置的投影空间，opengl会自动将该空间归一化，也就是将该空间或立方体转化为边长为1的正六面体投影空间，并且该正六面体的中心在相机坐标系的原点，一旦设置使用glOrtho函数设置投影空间，opengl会生成投影矩阵，这个矩阵的作用就是将坐标进行正交投影并且将投影后的坐标归一化（转换到-1到1之间）

**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

**纹理参数**

**纹理环境**

**纹理过滤**

**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);

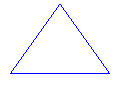
glColor3f(0.0f, 0.0f, 1.0f);

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);

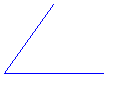
**glEdgeFlag(GL\_FALSE);** // 将以400为起点的线设置为非起点

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。

**Opengl的顶点数组**

按照正常的顶点依次绘制，需要调用的函数太多，同时另外一个问题是相邻多边形的共享顶点的冗余处理

Opengl提供了一些顶点数组函数，允许只用少数几个数组指定大量与顶点相关的数据，并用少量函数调用（与顶点数组的数量相仿）访问这些数据。使用顶点数组，一个拥有20条边的多边形的20个顶点可以放在一个数组中，并且只通过1个函数进行调用。如果每个顶点还有一条法线向量，所有20条法线向量可以放在另一个数组中，也可以只通过一个函数进行调用。

**把数据放在顶点数组中，可以提高应用程序的性能。使用顶点数组可以减少函数调用的次数，从而提高性能。另外，使用顶点数组还可以避免共享顶点的冗余处理。**

使用顶点数组对几何图形进行渲染需要3个步骤：

1 激活（启用）最多可达8个数组，每个数组用于存储不同的类型的数据：顶点坐标、表面法线、RGBA颜色、辅助颜色、颜色索引、雾坐标、纹理坐标以及多边形的边界标志

2 把数据放入数组中，这些数组是通过他们的内存位置的地址（即指针）进行访问的

3 用这些数据绘制几何图形，opengl通过指针从所有的被激活数组中获取数据

**步骤一：启动数组**

调用glEnableClientState函数（使用一个枚举参数），激活选择的数组，从理论上来说，最多可能调用这个函数8次，激活8个可用数组。但在实践中，可以激活的数组最多只有6个，这个因为有些数组不能同时激活，例如GL\_COLOR\_ARRAY和GL\_INDEX\_ARRAY。应用程序的显示模式可以支持RGBA模式，也可以支持颜色索引模式，但不能同时支持者两种模式

该函数的参数可以为下列选项：

GL\_VERTEX\_ARRAY、GL\_COLOR\_ARRAY、GL\_SECONDARY\_COLOR\_ARRAY、GL\_INDEX\_ARRAY、GL\_NORMAL\_ARRAY、GL\_FOG\_COORDINATE\_ARRAY、GL\_TEXTURE\_COORD\_ARRAY和GL\_EDGE\_FLAG\_ARRAY

如果需要使用光照，可能需要为每个顶点定义一条法线向量，在这种情况下使用顶点数组时，需要同时激活表面法线数组和顶点坐标数组

glEnableClientState(GL\_NORMAL\_ARRAY);

glEnableClientState(GL\_VERTEX\_ARRAY);

为什么opengl的设计者要添加一个新的glEnableClientState函数，而不是沿用原来的glEnable函数？

Opengl被设计为客户端/服务器模式，服务器就是图形硬件，客户端就是cpu和内存，由于这对设置功能启用/禁用状态的函数专门用于图形的客户端，因此需要专门设计一堆用于客户端的函数

**步骤二：指定数组的数据**

glVertexPointer(Glint size, GLenum type, GLsizei stride, const GLvoid\* pointer)

指定了需要访问的空间坐标数据。pointer是数组包含的第一个顶点的第一个坐标的内存地址。type指定了数组中每个坐标的数据类型（GL\_SHORT、GL\_INT、GL\_FLOAT、GL\_DOUBLE）。size是每个顶点的坐标数量，它必须是2、3或4，stride是连续顶点之间的字节偏移量，如果stride是0，数组中的顶点便是紧密相邻的

为了访问其他的几个数组，可以使用下面这些类似的函数：

Void glColorPointer(Glint size, GLenum type, GLsizei stride, const GLvoid\* pointer)

Void glSecondaryColorPointer(Glint size, GLenum type, GLsizei stride, const GLvoid\* pointer)

在涉及纹理时，还有一个小小的问题，使用glBegin/glEnd模式时。我们知道有一个称为glMultiTexCoord的新函数负责为每个纹理单位发送纹理坐标，使用顶点数组时，我们可以用下面的这个函数更改glTexCoordPointer的目标纹理

glClientActiveTexture(GLenum texture)

其中texture参数是GL\_TEXTURE0、GL\_TEXTURE1等

启动多个顶点数组例子：

void testVertexArray()

{

GLshort vertices[] = {25, 25, 100, 100, 120, 120, 200, 200};

GLfloat colors[] = {1.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f};

glEnableClientState(GL\_VERTEX\_ARRAY); // 启动顶点数组

glEnableClientState(GL\_COLOR\_ARRAY);

glVertexPointer(2, GL\_SHORT, 0, vertices);

glColorPointer(3, GL\_FLOAT, 0, colors);

//glShadeModel(GL\_FLAT);

glBegin(GL\_LINES);

glArrayElement(0); // 一次只能操作一个顶点

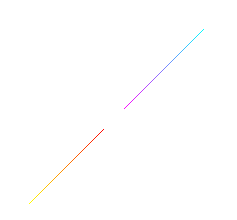
glArrayElement(1);

glArrayElement(2);

glArrayElement(3);

glEnd();

}



glArrayElement函数从任何已经glEnableClientState启用的数组中查找对应的数组数据，如果一个数组被启用，并且还没有指定一个对应的数组（glColorPointer、glVertexPointer等），就可能出现非法的内存访问，可能导致程序崩溃。使用glArrayElement的优点就是只要用一个函数调用就可以代替一个特定顶点指定所有数据所需要的几个函数调用（glNormal、glColor、glVertex等）

通过跨距将多个数组放置在一个数组

void testVertexArray()

{

GLfloat pointers[] = {25, 25, 1.0f, 1.0f, 0.0f,

100, 100, 1.0f, 0.0f, 0.0f,

120, 120, 1.0f, 0.0f, 1.0f,

200, 200, 0.0f, 1.0f, 1.0f};

glEnableClientState(GL\_VERTEX\_ARRAY); // 启动顶点数组

glEnableClientState(GL\_COLOR\_ARRAY);

glVertexPointer(2, GL\_FLOAT, 5 \* sizeof(GLfloat), &pointers[0]);

glColorPointer(3, GL\_FLOAT, 5 \* sizeof(GLfloat), &pointers[2]);

glBegin(GL\_LINES);

glArrayElement(0);

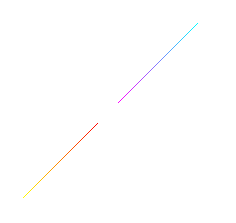
glArrayElement(1);

glArrayElement(2);

glArrayElement(3);

glEnd();

}



解引用顶点数组列表

Opengl提供了一个glDrawElements函数用于循环调用glArrayElement，但是需要调用一个索引数组。（与上面的等价）

void testVertexArray()

{

GLfloat pointers[] = {25, 25, 1.0f, 1.0f, 0.0f,

100, 100, 1.0f, 0.0f, 0.0f,

120, 120, 1.0f, 0.0f, 1.0f,

200, 200, 0.0f, 1.0f, 1.0f};

GLubyte index[] = {0, 1, 2, 3};

glEnableClientState(GL\_VERTEX\_ARRAY); // 启动顶点数组

glEnableClientState(GL\_COLOR\_ARRAY);

glVertexPointer(2, GL\_FLOAT, 5 \* sizeof(GLfloat), &pointers[0]);

glColorPointer(3, GL\_FLOAT, 5 \* sizeof(GLfloat), &pointers[2]);

glDrawElements(GL\_LINES, 4, GL\_UNSIGNED\_BYTE, index);

}

甚至可以直接用glDrawArrays(GL\_LINES, 0, 4)代替glDrawElements

混合数组：

glInterleavedArrays也可以同时指定几个顶点数组，同时该函数还能启用和禁用适当的数组（因此，它组合了步骤一的启动数组和步骤二的指定数据数据）

GLfloat pointers[] = {1.0f, 1.0f, 0.0f, 25, 25, 0,

1.0f, 0.0f, 0.0f, 100, 100, 0,

1.0f, 0.0f, 1.0f, 120, 120, 0,

0.0f, 1.0f, 1.0f, 200, 200, 0};

glInterleavedArrays(GL\_C3F\_V3F, 0, pointers);

glDrawArrays(GL\_LINES, 0, 4);

**opengl的索引顶点数组**

索引顶点数组就是并不按顺序从头遍历到尾的顶点数组，它的访问顺序是由一个单独的索引数组指定的。索引顶点数组可以节省内存，并减少变换开销，在理想状态下，它们可能比显示列表更快。

使用普通的渲染方式或顶点数组，不存在其他机制让两个相邻的三角形共享一组顶点，如下图显示了共享一条边的三角形带，尽管三角形带可以很好地处理各种三角形共享顶点，但是我们没有办法避免两个三角形带所共享的顶点的变换开销，因为每个三角形带都必须单独指定。



例子：一个简单的立方体

如果我们在一个顶点数组中对法线或顶点进行复用，而不是多次存储它，将可以节省大量的内存，除了节省内存外，优秀的opengl实现还可以对它进行优化，将这些顶点只变换一次，从而节省宝贵的变换时间。

static GLfloat corners[] = {

-25.0f, 25.0f, 25.0f, // 0 // 立方体的前面

25.0f, 25.0f, 25.0f, // 1

25.0f, -25.0f, 25.0f, // 2

-25.0f, -25.0f, 25.0f, // 3

-25.0f, 25.0f, -25.0f, // 4 // 立方体的后面

25.0f, 25.0f, -25.0f, // 5

25.0f, -25.0f, -25.0f, // 6

-25.0f, -25.0f, -25.0f // 7

};

static GLubyte indexes[] = {

0, 1, 2, 3, // 前面

4, 5, 1, 0, // 顶面

3, 2, 6, 7, // 底面

5, 4, 7, 6, // 后面

1, 5, 6, 2, // 右面

4, 0, 3, 7 // 左面

};

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

glPushMatrix();

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

glRotatef(20.0f, 1.0f, 0.0f, 0.0f);

glRotatef(20.0f, 0.0f, 0.0f, 1.0f);

glEnableClientState(GL\_VERTEX\_ARRAY);

glVertexPointer(3, GL\_FLOAT, 0, corners);

**glDrawElements(GL\_QUADS, 24, GL\_UNSIGNED\_BYTE, indexes);**

glDisableClientState(GL\_VERTEX\_ARRAY);

glPopMatrix();

注意：具有锐利边缘和叫的模型所拥有的可共享顶点往往较少，但是，那些具有较大平滑表面区域的模型在使用索引顶点数组时常常能节省更多的**内存和转换开销**，通过内存传输的集合图形越少，意味着对应的数学操作就越少。即使是在静态图形中，索引顶点数组有时候也可以取得比显式列表高得多的效果。对于许多实时渲染程序来说，索引顶点数组常常是进行几何图形渲染的理想选择，很快就可以看到，在顶点数组中更进一步，实现更快的性能，把显示列表远远抛开一边

**Opengl的VBO**

顶点数组的数据是存储在内存中的，顶点缓冲对象的数据是存在显存中的

顶点数组渲染时数据从内存送入渲染管线，而VBO则从显存送入渲染管线

顶点数组通过**glDrawEliments或glDrawArrays从cpu主存中拷贝到gpu**中进行运算与渲染

Opengl顶点缓冲区对象。GL\_ARB\_vertex\_buffer\_object扩展可以提升opengl的性能，它提供了顶点数组和显示列表，这避免了低效实现这些功能。**Vertex buffer object(VBO)允许顶点数组储存在高性能显卡上，即服务器端的内存中，改善数据传输效率，如果缓冲区对象保存了像素数据，它就被称为Pixel Buffer Object(PBO)**

使用顶点数组可以减少函数调用次数及复用共享顶点，然而，顶点数组的缺点是顶点函数以及顶点数据在客户端（对于opengl来说，显卡为服务器端，其他为客户端），每次引用顶点数组时，都必须将顶点数据从客户端（CPU）发送到服务器端（GPU），另一方面，显示列表是服务端函数，它不会再重头传送数据，但是，一旦显示列表被编译了，显示列表中的数据就不能修改了。

VBO为顶点创建了一个缓冲区对象。缓冲区对象在服务器端的高性能内存中，并提供了相同的函数，引用这些数组，如glVertexPointer，glNormalPointer，glTexCoorPointer等等。顶点缓冲区内存管理器将缓冲区对象放在存储器中的最佳位置，与显示列表不同的是，在顶点缓冲区对象中的数据可以读也可以将它映射到服务器端的内存空间中，然后更新它的数据。

VBO的另外一个重要优点是，可以在许多客户端中共享缓冲区对象，就像显示列表和纹理那样，由于vbo在服务器端，多个客户端可以通过对应的标识符访问同一个缓冲区。

**创建VBO的三个步骤**

1）使用glGenBuffersARB得到一个新的缓冲区对象

2）使用glBindBufferARB绑定一个缓冲区对象

3）使用glBufferDataARB复制顶点数据到缓冲区对象

glGenBuffersARB(GLsizei n, GLuint\* buffers)

glGenBuffersARB创建缓冲区对象并返回缓冲区对象的标识符。

参数n：表示需要创建顶点缓存对象的个数

参数buffers：用于存储创建好的顶点缓存对象句柄，句柄使用时大于0的正整数，0是opengl es保留

glBindBufferARB(GLenum target, GLunit buffer)

一旦缓冲区对象被创建，我们要再使用缓冲区对象前将缓冲区对象与ID绑定

参数target：指定绑定的对象，取值为GL\_ARRAY\_BUFFER（用于顶点数据）或GL\_ELEMENT\_ARRAY\_BUFFER（用于索引数据）

参数buffer：顶点缓存对象句柄

glBufferData(GLenum target, GLsizeiptr size, const GLvoid\* data, GLenum usage)

为顶点缓存对象分配空间，加载缓冲区对象

参数target：与glBindBufferARB的参数target相同

参数size：指定点点缓存区的大小，以字节为单位计数

参数data：用于初始化顶点缓存区的数据，可以为NULL，表示只分配空间，之后再由glBufferSubData进行初始化

参数usage：表示该缓存区域将会被如何使用，它的主要目的是用于提示opengl该对该缓存区域做何种程序的优化

其参数为以下三个之一：

GL\_STATIC\_DRAW：表示该缓冲区不会被修改

GL\_DYNAMIC\_DRAW：表示该缓存区的数据可能会经常变化

GL\_STREAM\_DRAW：表示该缓存区的数据可能不太会变化

如果顶点数据一经初始化就不会再被修改，那么就应该尽量使用GL\_STATIC\_DRAW，这样

能获得更好的性能

glBufferSubData(GLenum target, GLintptr offset, GLsizeiptr size, const GLvoid\* data)

参数offset：表示需要更新的数据的起始偏移量

参数size：表示要更新的数据的个数，也是以字节为计数单位

参数data：用于更新的数据

glDeleteBuffers(GLsizei , const GLunit\* buffers)

参数与glGenBuffers类似，用于删除顶点缓存对象，释放顶点缓存

也就是分为两个阶段（初始化阶段、使用阶段和收尾阶段）

1）初始化阶段

glGenBufferARB

glBindBufferARB

glBufferDataARB

2）使用阶段

glEnableClientState(GL\_VERTEX\_ARRAY) 开始使用VBO

glBindBufferARB(GL\_ARRAY\_BUFFER\_ARB,nVBOVertices) 选择当前VBO

glVertexPointer() 指定vbo顶点格式

glDrawArrays() 绘制

glDisableClientState(GL\_VERTEX\_ARRAY) 停止使用vbo

3）收尾阶段

glDeleteBufferARB(1, &nVBOVertices) 删除句柄，同时删除server端顶点缓冲

上面的过程与顶点数组的调用有一些变化：

1. 需要调用vbo初始化的三个函数
2. 需要调用glBindBuffer函数

上面的过程和纹理缓冲对象很相似：

1）初始化阶段

glGenTextures 创建句柄

glBindTexture(GL\_TEXTURE\_2D, texID) 选择当前使用的纹理缓冲

发送顶点和纹理坐标，绘制

glDisable(GL\_TEXTURE\_2D) 停止使用纹理

2）收尾阶段

glDeleteTextures 删除句柄，同时阐述server端缓冲

这个概念都很相似：

1. 创建句柄
2. 设置句柄类型
3. 上传数据
4. 开始使用缓冲
5. 选择句柄
6. 使用缓冲
7. 停止使用缓冲
8. 删除句柄和缓冲

glEnable glEnableClientState一个是针对服务器端的，一个是针对客户端的

**VBO与VA性能的比较：**

1）直接使用VA：

auto start = chrono::system\_clock::now();

for(int i = 0, size = 10000000; i < size; i++)

{

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

glEnableClientState(GL\_VERTEX\_ARRAY);

glVertexPointer(2, GL\_FLOAT, 0, vertices);

glDrawElements(GL\_LINES, 4, GL\_UNSIGNED\_INT, index);

}

auto end = chrono::system\_clock::now();

cout << "elapse time = " << chrono::duration\_cast<chrono::milliseconds>(end - start).count() << endl;



2）使用VBO

初始化：

glGenBuffersARB(1, &vboId);

glBindBufferARB(GL\_ARRAY\_BUFFER\_ARB, vboId);

glBufferDataARB(GL\_ARRAY\_BUFFER\_ARB, sizeof(vertices) \* sizeof(GL\_FLOAT), vertices, GL\_STATIC\_DRAW\_ARB);

delete[] vertices;

调用：

glEnableClientState(GL\_VERTEX\_ARRAY);

glBindBufferARB(GL\_ARRAY\_BUFFER\_ARB, vboId);

glVertexPointer(2, GL\_FLOAT, 0, NULL);

glDrawArrays(GL\_LINES, 0, NUMBER \* 4);

注意：在数据被复制到缓冲区对象之后，原指针就不再需要了，所有的工作空间缓冲区都被删除，这就产生了三个影响：首先，它会释放客户端内存，其次，它会消耗图形硬件的内存，最后，我们不再能够对数据进行修改，因为无法再访问它

下面的例子是开启多个缓冲区（其中顶点坐标、颜色与纹理为一个，边界标志为一个）：

static GLfloat vertices[] =

{

-100.0f, -100.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,

100.0f, -100.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f,

100.0f, 100.0f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f,

-100.0f, 100.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f

};

static GLboolean edgeFlag[] =

{

GL\_TRUE, GL\_FALSE, GL\_TRUE, GL\_TRUE

};

初始化：

glGenBuffersARB(1, &vboId);

glBindBufferARB(GL\_ARRAY\_BUFFER, vboId);

glBufferDataARB(GL\_ARRAY\_BUFFER, sizeof(GL\_FLOAT)\* sizeof(vertices), vertices, GL\_STATIC\_DRAW);

glGenBuffersARB(1, &edgeFlagId);

glBindBufferARB(GL\_ARRAY\_BUFFER, edgeFlagId);

glBufferDataARB(GL\_ARRAY\_BUFFER, sizeof(GL\_UNSIGNED\_BYTE)\* sizeof(edgeFlag), edgeFlag, GL\_STATIC\_DRAW);

// 纹理1

GLint width, height, component;

GLenum format;

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1);

glGenTextures(1, &textureId1);

glBindTexture(GL\_TEXTURE\_2D, textureId1);

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

if (bytes == NULL) cout << "load tga picture failed" << endl;

glTexImage2D(GL\_TEXTURE\_2D, 0, component, width, height, 0, format, GL\_UNSIGNED\_BYTE, bytes);

free(bytes);

// 设置纹理参数

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

// 设置纹理环绕

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

// 设置纹理环境

glTexEnvf(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_MODULATE);

// 纹理2

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1);

glGenTextures(1, &textureId2);

glBindTexture(GL\_TEXTURE\_2D, textureId2);

bytes = gltLoadTGA("..\\logo.tga", &width, &height, &component, &format);

if (bytes == NULL) cout << "load tga picture failed" << endl;

glTexImage2D(GL\_TEXTURE\_2D, 0, component, width, height, 0, format, GL\_UNSIGNED\_BYTE, bytes);

free(bytes);

// 设置纹理参数

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

// 设置纹理环绕

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

// 设置纹理环境

glTexEnvf(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_MODULATE);

// 开启纹理对象

glEnable(GL\_TEXTURE\_2D);

绘制：

// 顶点坐标与颜色

GLsizei stride = 9 \* sizeof(GL\_FLOAT);

GLuint \*vertexPointer = NULL;

GLvoid \*colorPointer = ((GLubyte\*)NULL) + 3 \* sizeof(GL\_FLOAT);

GLvoid \*texPointer = ((GLubyte\*)NULL) + 7 \* sizeof(GL\_FLOAT);

glBindBufferARB(GL\_ARRAY\_BUFFER, vboId);

if (index % 2 == 0)

{

glBindTexture(GL\_TEXTURE\_2D, textureId1);

}

else

{

glBindTexture(GL\_TEXTURE\_2D, textureId2);

}

index++;

glEnableClientState(GL\_VERTEX\_ARRAY);

glEnableClientState(GL\_COLOR\_ARRAY);

glEnableClientState(GL\_TEXTURE\_COORD\_ARRAY);

glVertexPointer(3, GL\_FLOAT, stride, NULL);

glColorPointer(4, GL\_FLOAT, stride, colorPointer);

glTexCoordPointer(2, GL\_FLOAT, stride, texPointer);

// 顶点边界标志

glBindBufferARB(GL\_ARRAY\_BUFFER, edgeFlagId);

glEnableClientState(GL\_EDGE\_FLAG\_ARRAY);

glEdgeFlagPointer(0, NULL);

glDrawElements(GL\_TRIANGLES, sizeof(indices), GL\_UNSIGNED\_BYTE, indices);

glBindBufferARB(GL\_ARRAY\_BUFFER, NULL);



关于glVertexAttribPointer

简单来说，它和glVertexPointer是同一个东西，只是如果用了glVertexAttribPointer，那么我们就不需要死板地只制定vertices为顶点位置，而是可以创建一个包含了所有顶点位置+顶点颜色+顶点法线+顶点XX的数组，通过一次将其送入显存，然后调用glVertexAttribPointer索引其属性就可以实现了，比如，有一个attrib[]数组，前3位位位置，后面一次为颜色、法线，这样就有3个index，依次为0，1,2，在调用glVertexAttribPointer(0,3GL\_FLOAT,GL\_FALSE,0,0)；就表示指定了buffer中第一个索引为pos，然后用颜色和法线以此类推

通常我们可以通过glBindAttribLocation绑定一个自定义的属性名字到一个index

**特别注意：该glBindAttribLocation的调用必须在link之前（已经验证）**

glProgram->init("sample.vert", "sample.frag");

glProgram->bindAttrib(1, "a\_position");

glProgram->link();

在link program时，opengl ed2.0对每个顶点属性执行如下操作：

首先检查属性变量是否被通过glBindAttribLocation绑定了属性，如果是，则使用此绑定的属性索引；否则，位置分配一个属性索引。在应用程序中，一般使用函数glBindAttribLocation来绑定每个attribute变量的位置，然后用函数glVertexAttribPointer为每个attribute变量赋值。

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

**如果对于动态数据的修改可以调用：**

glMapBufferARB与glUnmapBufferARB

glEnableClientState(GL\_VERTEX\_ARRAY);

glBindBufferARB(GL\_ARRAY\_BUFFER\_ARB, vboId);

glVertexPointer(2, GL\_FLOAT, 0, NULL);

glDrawArrays(GL\_LINES, 0, NUMBER \* 4);

GLfloat\* bufferData = (GLfloat\*)glMapBufferARB(GL\_ARRAY\_BUFFER, GL\_READ\_WRITE\_ARB);

if(bufferData != nullptr)

{

for(int i = 0, size = NUMBER \* 4; i < size; i++)

{

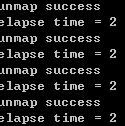
bufferData[i]++;

}

}

boolean b = glUnmapBufferARB(GL\_ARRAY\_BUFFER\_ARB);

cout << (b ? "unmap success" : "unmap failed") << endl;



实际的表现结果就是发现一条直线在向右上方移动（正确）

glMapBufferARB 用来将一个缓冲区对象中的数据映射为客户端中的地址空间

glUnmapBufferARB用来释放缓冲区对象与客户端地址空间的关系，如果一个映射关系在使用之后没有使用glUnmapBufferARB来释放，那么任何访问缓冲区对象的命令都会导致错误，当映射关系被解除后，使用映射得到的指针就会失效

注意：glMapBuffer()提供了对缓冲区对象中包含的整个数据集合的访问。如果需要修改

缓冲区中的大多数数据，这种方法很有用，但是，如果有一个很大的缓冲区并且只需要更新

很小的一部分值，这种方法效率很低。

C++代码

1. glBindBuffer(GL\_ARRAY\_BUFFER,m\_nPositionVBO);
2. glEnableClientState(GL\_VERTEX\_ARRAY);
3. glVertexPointer(2, GL\_FLOAT, 0, NULL);
4. glBindBuffer(GL\_ARRAY\_BUFFER,m\_nTexcoordVBO);
5. glEnableClientState(GL\_TEXTURE\_COORD\_ARRAY);
6. glTexCoordPointer(2, GL\_FLOAT, 0, NULL);
7. glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER,m\_nIndexVBO);
8. glDrawElements(GL\_TRIANGLES, 6, GL\_UNSIGNED\_SHORT,NULL);
9. glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, NULL);
10. glDisableClientState(GL\_TEXTURE\_COORD\_ARRAY);
11. glDisableClientState(GL\_VERTEX\_ARRAY);
12. glBindBuffer(GL\_ARRAY\_BUFFER, NULL);

C++代码2

1. glBindBuffer(GL\_ARRAY\_BUFFER,m\_nQuadPositionVBO);
2. glEnableVertexAttribArray(VAT\_POSITION);
3. glVertexAttribPointer(VAT\_POSITION, 2, GL\_INT, GL\_FALSE,0, NULL);
4. glBindBuffer(GL\_ARRAY\_BUFFER,m\_nQuadTexcoordVBO);
5. glEnableVertexAttribArray(VAT\_TEXCOORD);
6. glVertexAttribPointer(VAT\_TEXCOORD, 2, GL\_INT, GL\_FALSE,0, NULL);
7. glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER,m\_nQuadIndexVBO);
8. glDrawElements(GL\_TRIANGLES, 6, GL\_UNSIGNED\_SHORT,NULL);
9. glDisableVertexAttribArray(VAT\_POSITION);
10. glDisableVertexAttribArray(VAT\_TEXCOORD);
11. glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, NULL);
12. glBindBuffer(GL\_ARRAY\_BUFFER, NULL);

以上两端是效果一致的VBO渲染部分的代码，尽量使用第二种，使用第二种的前提是使用shader来进行顶点处理。VAT\_POSITION / VAT\_TEXCOORD需要与shader里代表顶点/纹理坐标的attribute变量建立联系。

**opengl的VAO**

VBO在渲染阶段才指定数据位置和顶点信息，然后根据此信息去解析缓冲区里的数据，联系着两者中间的桥梁是GL-Context。GL-Context整个程序一般只有一个，所以如果一个渲染流程里有两份不同的绘制代码，GL-Context就负责在他们之间进行状态切换、这也是为什么在渲染过程中，在每份绘制代码之中有glBindBuffer/glEnableVertexAttribArray/glVertexAttribPointer。那么优化方法就来了---把这些都放在初始化时候完成吧！--这样做的限制条件是“负责记录状态的GL-Context一般只有一个”，那么就不直接用GL-Context记录，用别的东西做记录吧--这个东西针对“每份绘制代码”有一个，记录该次绘制所需要的所有VBO所需信息，把它保存到GPU的特定位置，绘制的时候直接在这个位置取信息绘制。于是，VAO诞生了。

VAO的全名是Vertex Array Object，首先，它不是Buffer-Object，所以不用做存储数据；其次，它针对“顶点”而言，也就是说它根“顶点的绘制”息息相关。

**opengl的FBO**

**OpenGL默认的帧缓存包括：颜色缓存、深度缓存、模板缓存、累积缓存。**

**OpenGL扩展帧缓存（即FBO）包括：颜色缓存、深度缓存、模板缓存。**

在opengl渲染管线中，几何数据和纹理经过多次转化和测试，最后以二维像素的形式显示在屏幕上。opengl管线的最终渲染目的地被称为帧缓冲。帧缓冲是一些二维数组和opengl所使用的存储区的集合：颜色缓存、深度缓存、模板缓存和累积缓存。一般情况下，帧缓存完全由window系统生成和管理，由opengl使用，这个默认的帧缓存称为“window系统生成”（window-system-provided）的帧缓存。

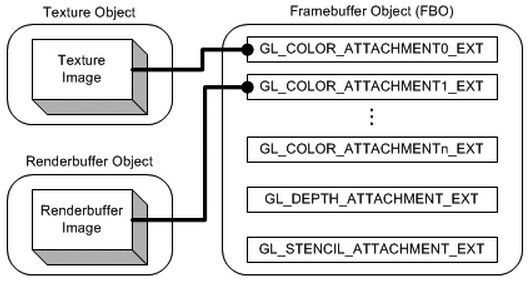
FBO最常见的应用就是：渲染到纹理（render to texture），通过这项技术可以实现发光效果、环境映射、阴影映射等很炫的效果，FBO被推荐用于把数据渲染到纹理对象，相对于其他同类技术，如数据拷贝或交换缓冲区等，使用FBO技术会更高效并且容易实现。

在opengl扩展中，GL\_EXT\_framebuffer\_object提供了一种创建额外的不能显示的帧缓存对象接口。为了和默认的“window系统生成”的帧缓存区别，这种缓存就称为应用程序帧缓存（application-createframebuffer）。通过使用帧缓存对象（FBO），opengl可以将显示输出到引用程序帧缓存对象，而不是传统的“window系统生成”帧缓存，而且，它完全受opengl控制。

类似于window系统提供的帧缓存，一个FBO也包含一些存储颜色、深度和模板数据的区域**（注意：没有累积缓存）**，我们把FBO中这些逻辑缓存称为“帧缓存关联图像”，它们是一些能够和一个帧缓存对象关联起来的二位数组像素。

有两种类型的“帧缓存关联图像“：纹理图像（texture images）和渲染缓存图像（renderbufer images）。如果纹理对象的图像数据关联到帧缓存，opengl执行的是”渲染到纹理“操作，如果渲染缓存的图像数据关联到帧缓存，opengl执行的是离线渲染（offscreen rendering）。

这里要注意的是，渲染缓存对象是在GL\_EXT\_framebuffer\_object扩展中定义的一种新的存储类型，在渲染过程中它被用作存储单幅二维图像。下面这幅图显示了帧缓存对象、纹理对象和渲染缓存对象之间的联系。多个纹理对象或者渲染缓存对象能够通过关联点关联到一个帧缓存对象上。



在一个帧缓存对象中有多个颜色关联点（GL\_COLOR\_ATTACHMENT0\_EXT... GL\_COLOR\_ATTACHMENTn\_EXT），一个深度关联点（GL\_DEPTH\_ATTACHMENT\_EXT）和一个模板关联点（GL\_STENCIL\_ATTACKMENT\_EXT）。每个FBO中至少有一个颜色关联点，其数目与实体显卡相关。可以公国GL\_MAX\_COLOR\_ATTACKMENTS\_EXT来查询颜色关联点的最大数目。FBO有多个颜色关联点的原因是这样可以同时将颜色而换成渲染到多个FBO关联去。这种“多渲染目标（multiple rendertargets，MRT）”可以通过GL\_ARC\_draw\_buffers扩展实现，需要注意的是：FBO本身并没有任何图像存储区，只有多个关联点。

FBO提供了一种高效的切换机制，将前面的帧缓存关联图像从FBO分离，然后把新的帧缓存关联图像关联到FBO。在帧缓存关联图像之间切换比在FBO之间切换要快得多，FBO提供了glFramebufferTexture2DEXT来切换2D纹理对象和glFramebufferRenderbufferEXT来切换渲染缓存对象。

1）glGenFramebuffersEXT(GLsizei n, GLuint \*ids)

该函数需要两个参数：第一个是要创建的帧缓存的数目，第二个是指向村塾一个或者多个ID的变量或数组的指针。它返回未使用的FBO的ID。ID为0表示默认帧缓存，即window系统提供的帧缓存

2）glBindFramebufferEXT(GLenum target, GLuint id)

一旦一个FBO被创建，在使用它之前必须绑定

第一个参数target应该是GL\_FRAMEBUFFER\_EXT，

**opengl的PBO**

**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

**opengl的显示列表**

几何图形或其他opengl数据在帧的变化过程中常常保持不变。例如，一个旋转的圆环面总是由同一组三角形带组成，具有相同的顶点数据，每一帧都要通过昂贵的三角函数重新进行计算，而帧与帧之间唯一发生的变换就是模型视图矩阵。解决这种不必要的重复渲染问题的一种方法是从命令缓冲区取出一块预先计算好的数据，他们负责执行一些重复的任务，例如绘制圆环面，这块数据随时可以复制回命令缓冲区，从而节省创建这块数据所需要的大量函数调用和编译开销。这就是显示列表，可以通过glNewList和glEndList定义一个显示列表

glNewList(<unsigned integer name>, GL\_COMPILE);

// 一些opengl代码

glEndList()

GL\_COMPILE告诉opengl编译这个列表但不要执行它，我们也可能执行GL\_COMPILE\_AND\_EXECUTE，创建这个显示列表并执行其中的渲染命令，但是一般而言，显示列表是在程序的初始化阶段创建的（只使用GL\_COMPILE），以后在渲染期间再执行

**显示列表的名称可以是任何无符号整数，但是，如果两次使用了相同的值，第二个显示列表将会覆盖前一个显示列表，opengl对分配唯一的显示列表提供了内置的支持，**

Gluint glGenLists(GLsizei range)

对应的用于释放显示列表名称以及这些显示列表所分配的内存的函数是：

glDeleteLists(GLuint list, GLsizei range)

然后，我们可以用glCallList(GLuint list)命令执行一个包含了任意数量的预编译OpenGL命令的显示列表，我们也可以使用glCallLists(GLsizei n, GLenum type, const GLvoid \*lists)

第一个参数指定了lists数组列表所包含的显示列表的数量，第二个参数表示这个数组的类型，在一般情况下，它是GL\_UNSIGNED\_BYTE。它可以非常方便地用做字体显示列表的地址偏移量

注意：一旦建立了显示列表就不能修改它，因为如果显示列表可以被修改，则显示列表的搜索、内存管理的执行等开销会降低性能

采用显示列表方式绘图一般要比立即模式快，尤其是显示列表方式可以大量地提高网络性鞥，即当通过王阔发出绘图命令时，由于显示列表驻留在服务器中，因而使网络的负担减轻到最小。另外，在单用户的机器上，显示列表同样可以提高效率，因为一旦显示列表被处理成适合于图形硬件的格式，则不同的opengl实现对命令的优化程序也不同。比如glRotate，若将它置于显示列表中，则可大大提高性能，在显示列表中，它值被存储为最终的渲染矩阵。一般来说，显示列表能将许多相邻的矩阵变换结合成单个的矩阵乘法，从而加快速度。

可以在glNewList与glEndList中调用glCallList，但是不允许在两者中间再次嗲用glNewList和glEndList

**普通方式与显示列表性能比较：**

groundList = glGenLists(3);

sphereList = groundList + 1;

torusList = groundList + 2;

glNewList(sphereList, GL\_COMPILE);

gltDrawSphere(10.0f, 40, 20);

glEndList();

glNewList(torusList, GL\_COMPILE);

gltDrawTorus(35, 0.15, 61, 37);

glEndList();

调用方法：

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

int method = 0;

auto start = chrono::system\_clock::now();

for(int i = 0, size = 10000; i < size; i++)

{

if(method == 0)

{

glCallList(torusList);

}

else

{

gltDrawTorus(35, 0.15, 61, 37);

}

}

auto end = chrono::system\_clock::now();

cout << "elapse time = " << chrono::duration\_cast<chrono::milliseconds>(end - start).count() << endl;

method==0时：

method==1时：

**关于几何变换：**

在二维情况下，所有的坐标变换（平移、放缩和旋转）都可以表示为P1 = M1\*P + M2，其中M1是一个2\*2的表达式

想象一下，如果我们想实现一个首先将坐标位置缩放，然后将缩放后的坐标旋转，最后将旋转后的坐标平移，我们必须一步一步地计算变换的坐标。更有效的方法是将变换组合，从而直接从初始坐标得到最后的坐标位置，这样就消除了中间坐标值的计算，计算量将大大的减少，因此，变换操作的有效实现是先形成变换矩阵、合并所有变换序列，最后计算变换坐标。

如果将2\*2矩阵表达式扩充为3\*3矩阵，就可以把二维几何变换的惩罚和平移项组合成单一矩阵表示。这时将变换矩阵的第三列用于平移项，而左右的变换公式可表达为矩阵乘法，这样就用到了齐次坐标。为了这样的操作，必须解释二维坐标位置到三元列向量的矩阵表示。标准的实现技术是将二维坐标位置表示（x，y）扩充到三维表示（xh，yh，h），其中

x=xh/h y=yh/h

这样，普通的二维齐次坐标表示可写为(h\*x, h\*y, h)，对于二维几何变换，可以把齐次参数h取为任何非零值，因此，对于每个坐标点(x,y)可以有无数个等价的齐次表达式

有些图形库仅仅只包含了绕坐标系原点旋转或放缩

对于旋转：可以通过平移-旋转-平移操作顺序来实现绕任意选定的基准点进行旋转

1 平移对象使其基准点位置移动到坐标原点

2 绕坐标原点旋转

3 平移对象使基准点回到其原始位置

对于放缩也一样：

1 平移对象使固定点与坐标原点重合

2 对于坐标原点进行缩放

3 使用步骤1的反向平移将对象返回到原始位置

void testGeometric()

{

GLfloat vertices[] = {100.0f, 100.0f,

120.0f, 100.0f,

120.0f, 120.0f,

100.0f, 120.0f};

GLfloat colors[] = {1.0f, 0.0f, 0.0f,

0.0f, 1.0f, 0.0f,

0.0f, 0.0f, 1.0f,

1.0f, 1.0f, 0.0f};

glEnableClientState(GL\_VERTEX\_ARRAY);

glEnableClientState(GL\_COLOR\_ARRAY);

glVertexPointer(2, GL\_FLOAT, 0, vertices);

glColorPointer(3, GL\_FLOAT, 0, colors);

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

//glScalef(1.5f, 1.0f, 1.5f);

glDrawArrays(GL\_QUADS, 0, 4);

}



void testGeometric()

{

GLfloat vertices[] = {100.0f, 100.0f,

120.0f, 100.0f,

120.0f, 120.0f,

100.0f, 120.0f};

GLfloat colors[] = {1.0f, 0.0f, 0.0f,

0.0f, 1.0f, 0.0f,

0.0f, 0.0f, 1.0f,

1.0f, 1.0f, 0.0f};

glEnableClientState(GL\_VERTEX\_ARRAY);

glEnableClientState(GL\_COLOR\_ARRAY);

glVertexPointer(2, GL\_FLOAT, 0, vertices);

glColorPointer(3, GL\_FLOAT, 0, colors);

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

**glScalef(1.5f, 1.0f, 1.5f);**

glDrawArrays(GL\_QUADS, 0, 4);

}



void testGeometric()

{

GLfloat vertices[] = {100.0f, 100.0f,

120.0f, 100.0f,

120.0f, 120.0f,

100.0f, 120.0f};

GLfloat colors[] = {1.0f, 0.0f, 0.0f,

0.0f, 1.0f, 0.0f,

0.0f, 0.0f, 1.0f,

1.0f, 1.0f, 0.0f};

glEnableClientState(GL\_VERTEX\_ARRAY);

glEnableClientState(GL\_COLOR\_ARRAY);

glVertexPointer(2, GL\_FLOAT, 0, vertices);

glColorPointer(3, GL\_FLOAT, 0, colors);

**glScalef(1.5f, 1.0f, 1.5f);**

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

glDrawArrays(GL\_QUADS, 0, 4);

}



上面的例子注意点：Sx与Sy缩放是沿着x和y方向缩放的，可以通过在应用缩放变换之前，将对象所希望的缩放方向旋转到与坐标轴一致而在其他方向上缩放对象

对可用glMatrixMode选择的4种模式（建模观察、投影、纹理和颜色）中的每一种，opengl都维护一个矩阵栈。开始，每一个栈仅包含单位矩阵，在处理场景的任何时刻，栈顶的矩阵称为该模式的“当前矩阵”。在指定观察和几何变换后，建模观察栈顶是一个应用于场景的观察变换和各种几何变换的4\*4复合矩阵。有时，需要创建多个视图和变换序列，然后分别保存复合矩阵。因此，opengl提供至少深度为32的建模观察栈，而有的实现允许在建模观察栈中保存超多32个矩阵，另外三个矩阵模式的栈深度至少为2

GLint modelViewStackDepth = 0, projectStackDepth = 0, textureStackDepth = 0;

glGetIntegerv(GL\_MAX\_MODELVIEW\_STACK\_DEPTH, &modelViewStackDepth);

glGetIntegerv(GL\_MAX\_PROJECTION\_STACK\_DEPTH, &projectStackDepth);

glGetIntegerv(GL\_MAX\_TEXTURE\_STACK\_DEPTH, &textureStackDepth);

cout << "modelViewStackDepth = " << modelViewStackDepth << endl;

cout << "projectStackDepth = " << projectStackDepth << endl;

cout << "textureStackDepth = " << textureStackDepth << endl;



如何获取当前栈中有多少个矩阵：

showModelViewStackMatrixCount();

glPushMatrix();

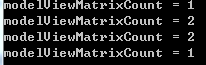
showModelViewStackMatrixCount();

glLoadIdentity();

showModelViewStackMatrixCount();

glPopMatrix();

showModelViewStackMatrixCount();



GLfloat matrix[] = {1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f};

showModelViewStackMatrixCount();

glPushMatrix();

glMultMatrixf(matrix);

glPushMatrix();

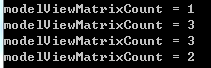
showModelViewStackMatrixCount();

glLoadIdentity();

showModelViewStackMatrixCount();

glPopMatrix();

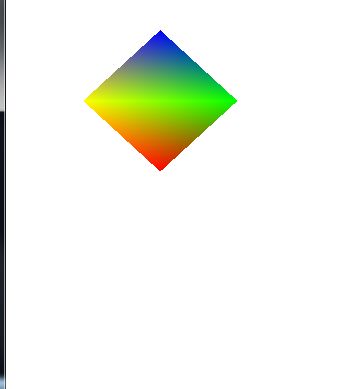
showModelViewStackMatrixCount();



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

glTranslatef(200.0f, 0.0f, 0.0f);

glDrawArrays(GL\_QUADS, 0, 4);



glTranslatef(200.0f, 0.0f, 0.0f);

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

glDrawArrays(GL\_QUADS, 0, 4);



上面的两种情况（理解）：

正向看，是改变局部坐标

反向看：是基于世界坐标

在选在opengl裁剪窗口和视口之前，必须建立合适的模式以便构建从世界坐标系到屏幕坐标系变换的矩阵。Opengl裁剪函数使用-1到1的规范化坐标系。gluOrtho2D函数设定变换矩阵的三维版本，将裁剪窗口中的对象映射到规范化坐标系。规范化正方形外的对象（及在裁剪窗口外的对象）不在显示的场景中出现。

Opengl可以为各种应用建立多个视口。获取当前活动视口参数的查询函数如下：

GLint\* vpArray = new GLint[4];

glGetIntegerv(GL\_VIEWPORT, vpArray);

cout << "x = " << vpArray[0] << ", y = " << vpArray[1] << endl;

cout << "width = " << vpArray[2] << ", height = " << vpArray[3] << endl;



**裁剪算法**

裁剪最多应用于观察流水线，目的是为了从场景（二维或三维）中指定部分显示在输出设备上。裁剪也用于对象边界的反走样、实体建模法构造对象，管理多窗口环境及在绘画程序中将图的一部分移动、复制或擦除

对裁剪窗口的规范化边界应用裁剪算法是实现观察流水线裁剪的高效方法，由于可以在裁剪前合并所有的几何和观察变换矩阵并应用于场景描述，因此大大减少了计算量。裁剪后的场景送到屏幕坐标系进行最后的处理。假设裁剪区域为XWmin,XWmax,YWmin,YWmax

图元的裁剪包括：点的裁剪、线段的裁剪、区域裁剪、曲线裁剪、文字裁剪

**（1）点的裁剪**

满足条件：XWmin <= x <= XWmax && YWmin <= y <= YWmax

**（2）线段的裁剪**

线段的裁剪算法通过一系列的测试和求交计算来判断是否整条线段或其中的某部分可以保存下来。线段与窗口边界的交点计算是线段裁剪函数的耗时部分，因此，减少交点计算是任一线段裁剪算法的主要目标。为此，我们可以先进行测试，确定线段是否完整地在裁剪窗口内部或完整地位于外部。确定一线段是否完整地在裁剪窗口内部是很容易的，但要确认整条线段都在窗口外部就比较困难。如果不能确定一条线段是否完整地在裁剪窗口的内部或者外部，则必须通过进行焦点计算来确定是否该线段有一部分落在窗口内部。目前已经有一些快速的裁剪算法，有些算法是针对二维图形的，有些算法可以很容易地移植到三维应用中。

1）Cohen-Sutherland裁剪算法

这是一个最早开发的快速线段裁剪算法，已经得到了广泛的使用