

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);

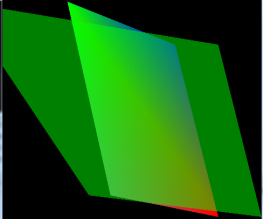
glDrawArrays(GL\_POLYGON, 0, 4);

**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

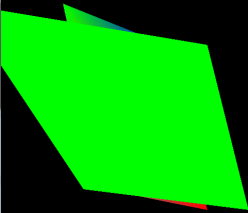
GLfloat colors2[] = {0.0f, 1.0f, 0.0f, 1.0f,

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);

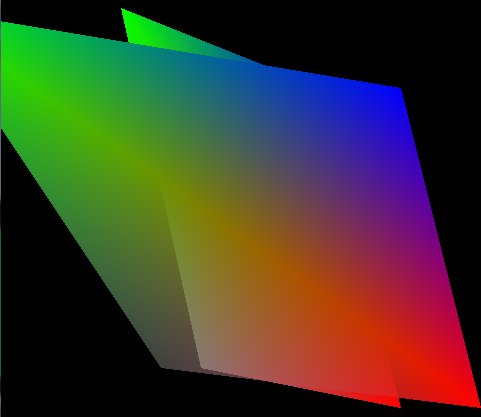
GLfloat colors2[] = {0.0f, 1.0f, 0.0f, 1.0f,

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

};

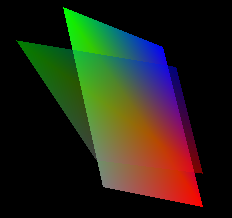
GLfloat colors2[] = {0.0f, 1.0f, 0.0f, 0.5f,

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

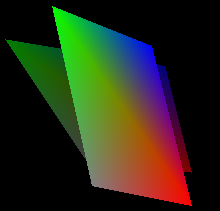
1.0f, 0.0f, 0.0f, 0.5f,

0.0f, 0.0f, 1.0f, 0.5f

};

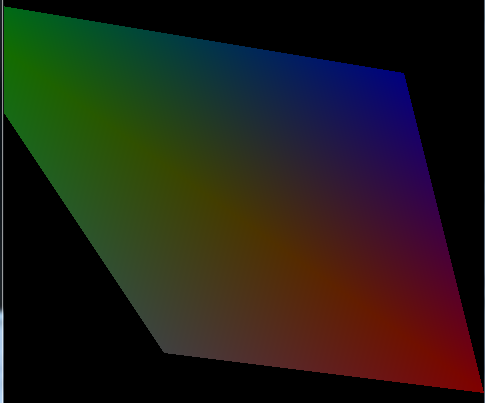


如果添加了：glEnable(GL\_DEPTH\_TEST);

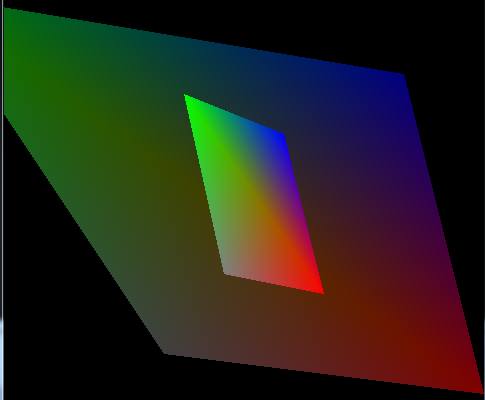


就会发现如果我们先绘制一个比较近的不透明物体，然后绘制一个比较远的透明物体，在调用深度测试的情况下，发现被挡住的不透明物体没有出现混合了（实际上是被遮挡的那一部分通过深度测试被舍弃了）

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。