OpenGL 编程

混合

混合

- 场景：透过红色的玻璃去看绿色的物体。
- 混合：把某一像素原来的颜色和将要画上去的颜色，通过某种方式混在一起，实现半透明效果：
  - 可先绘制绿色的物体，再绘制红色玻璃，在绘制红色玻璃时，利用“混合”功能，把将要绘制上去的红色和原来的绿色进行混合，从而得到一种新的颜色
  - 使用混合功能：glEnable(GL_BLEND)
  - 关闭混合功能： glDisable(GL_BLEND)

混合

- 使用混合功能前提条件：
  - 颜色模型为RGB模型
  - 颜色索引模型下无法使用
- 混合需要将原来的颜色和将要画上去的颜色找出来，经过某种方式处理后得到一种新颜色：
  - 将要画上去的颜色：源颜色
  - 原来的颜色：目标颜色

目标因子和源因子

- OpenGL将源颜色和目标颜色各自取出，并乘各自的系数（因子），然后相加，得到新颜色：
  - 运算种类较多：加，减，大和小
  - 数学表达式：(Rs*Sr+Rd*Dr,…)
  - 如果颜色某分量超过1.0，自动取值1.0
  - 源因子和目标因子的设定函数：glBlendFuc(s,o)

目标因子和源因子

- glBlendFuc(s,o)函数：
  - GL_ZERO：不使用这种颜色
  - GL_ONE：完全使用这种颜色
  - GL_SRC(DST)_ALPHA：源颜色的alpha值
  - GL_DST_ALPHA：目标颜色的alpha值
  - GL_ONE_MINUS_DST_ALPHA：(1-alpha)
  - GL_SRC_COLOR：源颜色的四个分量分别作为因子的四个分量
  - 其它有些设置因子的方法

目标因子和源因子

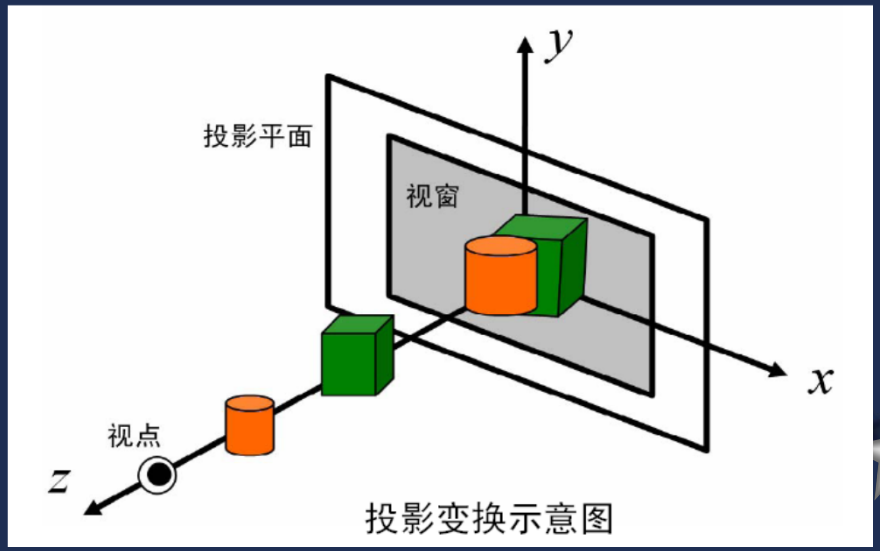
- 举例：
  - glBlendFuc(GL_ONE, GL_ZERO)：使用源颜色，不使用目标颜色(默认)
  - glBlendFuc(GL_ZERO, GL_ONE)
  - glBlendFuc(GL_ONE, GL_ONE)
  - glBlendFuc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)，alpha值可理解为“透明度”，混合中的常用方式
  - 源颜色和目标颜色跟绘制有关：先绘制目标颜色，再在其上绘制源颜色，绘制时要注意顺序，保证颜色和因子对应

二维图形混合举例

- 例子：实现将两种不同颜色的混合
  - 绘制有重叠区域的两个矩形
  - glBlendFunc(GL_ONE, GL_ZERO) 结果与不使用混合时相同
  - glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)，矩形都是半透明
  - glBlendFunc(GL_ONE, GL_ONE)，红色和绿色相加得到黄色

实现三维混合

- 深度缓存：记录了每一个像素距离观察者有多近，如起用深度缓存
  - 要绘制像素比原来像素更近，则被绘制否则不进行绘制
  - 不易显示半透明效果：近物体遮挡远物体



投影变换示意图

实现三维混合

- 如何实现半透明效果：
  - 绘制半透明物体时将深度缓冲区设置为只读：虽然半透明物体被绘制上去了，但深度缓冲区还保持原来的状态
  - 如果再有个物体出现在半透明物体之后，不透明物体之前，可被绘制
  - 如何绘制一部分透明一部分不透明物体：分成两个部分，分别绘制

实现三维混合

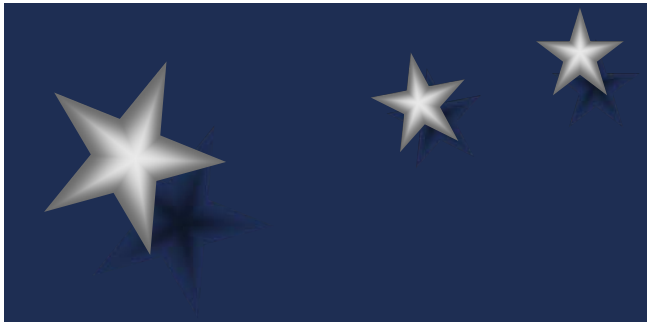
- 半透明效果绘制顺序：
  - 假设背景为蓝色，近处一块红色玻璃，中间一个绿色物体，如先绘制玻璃，则红色与蓝色混合，再绘制绿色物体时，无法与红色玻璃实现混合
  - 先绘制所有不透明物体，将深度缓冲区设为只读，绘制所有半透明物体，最后将深度缓冲区设置为可读可写形式
- 三维混合的实现：
  - glDepthMask(GL_FALSE)：将深度缓冲区设置为只读形式

实现三维混合

- 例子：假设有三个球体，红色不透明，绿色半透明，蓝色半透明。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。
  - 删去两处glDepthMask，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。

小结

- 介绍了OpenGL混合的相关知识
  - 混合不同于覆盖，新颜色为源颜色，旧颜色为目标颜色
  - 源因子和目标因子可分别设置，不同设置导致不同后果
  - 绘制顺序重要：先绘制成目标，后绘制为源
  - 三维混合：除考虑源和目标因子，还要考虑深度缓冲区，并注意绘制顺序



OpenGL 编程

像素操作

像素操作

- 计算机保存图像的类型：
  - 矢量图：保存图象中每个几何物体的位置、形状、大小等信息，在显示图象时，根据这些信息计算得到完整的图象
    - 进行放大、缩小时不会失真，数据量和运算量庞大
  - 像素图：图像由一系列像素组成，保存每一个像素的颜色就保存了整个图像
    - 放大、缩小时，数据量和运算量都不会增加
    - 会产生失真的情况

BMP文件格式

- BMP：像素文件，可以保存单色位图、16色或256色索引模式像素图、24位真彩色图象：
  - 常见的是256色BMP和24位色BMP
  - 定义像素保存方式：不压缩、RLE压缩等
- Windows所使用的BMP文件：
  - 54字节文件头：文件格式标识、颜色数、图象大小、压缩方式等信息（24位色、无压缩的BMP）；图像宽度和高度均为32位整数，在文件中的地址分别为0x0012和0x0016
  - 如果是16色或256色BMP，则还有一个颜色表
  - 实际像素数据:三个字节表示一个像素的颜色

BMP文件格式

- 读取图象的大小信息：
  - GLint width, height; // 使用OpenGL的GLint类型，32位。

```
FILE* pFile;            // 在这里进行“打开文件”的操作
fseek(pFile, 0x0012, SEEK_SET);  // 移动到0x0012位置
fread(&width, sizeof(width), 1, pFile); // 读取宽度
fseek(pFile, 0x0016, SEEK_SET);    // 移动到0x0016位置
                                   // 由于上一句执行后本就应该在
                                     0x0016位置
                                   // 所以这一句可省略
fread(&height, sizeof(height), 1, pFile); // 读取高度
```

BMP文件格式

- 注意事项：
  - OpenGL常用RGB来表示颜色，但BMP文件则采用BGR
  - 像素数据量不一定完全等于图象的高度乘以宽度乘以每一像素的字节数，而是可能略大于这个值
  - BMP文件采用了“对齐”机制:每行像素数据长度若不是4的倍数，则填充一些数据使它是4的倍数
  - 一幅17*15的24位BMP大小就应该是834字节
  - 分配内存时，一定要小心，有可能导致分配的内存空间长度不足，造成越界访问

BMP文件格式

- 简单的计算数据长度的方法如下：
  - int LineLength, TotalLength;
    LineLength = ImageWidth * BytesPerPixel;
    // 每行数据长度大致为图象宽度乘以
    // 每像素的字节数
    while(LineLength % 4 != 0)
    // 修正LineLength使其为4的倍数
    ++LineLenth;
    TotalLength = LineLength * ImageHeight;
    // 数据总长 = 每行长度 * 图象高度

简单的OpenGL像素操作

- OpenGL中的像素操作函数：
  - glReadPixels：把已绘制好的像素（存储在显卡内存中）读取到内存中
  - glDrawPixels：把内存中的数据作为像素数据进行绘制
  - glCopyPixels：把已绘制好的像素从一位置复制到另一位置，不需要经过内存，速度快

glReadPixels的用法

- glReadPixels函数（七个）参数说明：
  - 前四个参数：确定一个矩形，该矩形所包括的像素都会被读取出来
    - 第一、二个参数表示矩形的左下角坐标
    - 第三、四个参数表示矩形的宽度和高度
  - 第五个参数：读取的内容
    - GL_RGB依次读取像素的红、绿、蓝三种数据
    - GL_RGBA、GL_RED
    - 如果采用颜色索引模式，则也可以使用GL_COLOR_INDEX来读取像素的颜色索引
    - 还可读取其它内容，如深度缓冲区的深度数据等

glReadPixels的用法

- 第六个参数：读取的内容保存到内存时所使用的格式
  - GL_UNSIGNED_BYTE会把各种数据保存为GLubyte
  - GL_FLOAT会把各种数据保存为GLfloat等
- 第七个参数：一个指针，像素数据被读取后，将被保存到这个指针所表示的地址
  - 需要保证该地址有足够的可以使用的空间，以容纳读取的像素数据
  - 大小为256*256的图象，如读取RGB数据，且每一数据被保存为GLubyte，总大小：256*256*3=192千字节

glReadPixels的用法

- 注意事项：
  - glReadPixels实际上是从缓冲区中读取数据，如果使用了双缓冲区，则默认是前缓冲中读取
  - 绘制工作是默认绘制到后缓冲区
  - 如果需要读取已经绘制好的像素，往往需要先交换前后缓冲

glReadPixels的用法

- 解决OpenGL常用的RGB像素数据与BMP文件的BGR像素数据顺序不一致问题：
  - 用代码交换每个像素的第一和第三字节
  - 新版本的OpenGL可使用GL_RGB读取像素的红、绿、蓝数据外，也可使用GL_BGR按照相反的顺序依次读取像素的蓝、绿、红数据，这样就与BMP文件格式相吻合了
  - Windows环境下各种OpenGL实现都对GL_BGR提供了支持

glReadPixels的用法

- 消除BMP文件中“对齐”带来的影响：
  - glPixelStore:修改“像素保存时对齐的方式”

    glPixelStorei(GL_UNPACK_ALIGNMENT, int alignment);
    - 第一个参数表示“设置像素的对齐值”，第二个参数表示实际设置为多少
    - 像素可以单字节、双字节、四字节和八字节对齐，alignment对应值分别为1,2,4,8
    - 默认值是4，正好与BMP文件的对齐方式相吻合

glReadPixels举例

- 举例：如何把整个窗口图象抓取出来并保存为BMP文件：
  - 利用代码生成一幅图形，调用一函数，把图象内容保存为BMP文件

glDrawPixels的用法

- glDrawPixels函数的参数：
  - 第1、2、3和4个参数分别对应于glReadPixels函数第3、4、5和6个参数，依次表示图象宽度、图象高度、像素数据内容、像素数据在内存中的格式
  - 第5个参数：glReadPixels表示像素读取后存放在内存中的位置，glDrawPixels则表示用于绘制的像素数据在内存中的位置

glDrawPixels的用法

- glDrawPixels Vs glReadPixels函数：
  - 前者比后者少了两个参数，这两个参数在后者中分别是表示图像的起始位置
  - 前者不必显式的指定绘制的位置，因为绘制的位置是由另一个函数来指定的
  - glRasterPos*函数的参数与glVertex*类似，通过指定一个二维/三维/四维坐标，OpenGL将自动计算出该坐标对应的屏幕位置，并把该位置作为绘制像素的起始位置

glDrawPixels举例

- 举例：从BMP文件中读取像素数据，并使用glDrawPixels绘制到屏幕上
  - 把选定的bmp图片放到正确的位置，在程序开始时读取该文件，从而获得图象的大小后，根据该大小来创建合适的OpenGL窗口，并绘制像素

glCopyPixels的用法

- **例子**：绘制一个三角形后，复制像素，并同时进行水平和垂直方向的翻转，然后缩小为原来的一半，并绘制。
- 通过glReadPixels和glDrawPixels组合来实现复制像素的功能：
  - 一幅1024*768的图象，如使用24位BGR方式表示，则需要至1024*768*3字节，即2.25兆字节
  - 对该图像进行一次读操作和写操作，由于glReadPixels和glDrawPixels中设置的数据格式不同，很可能涉及到数据格式的转换

glCopyPixels的用法

- glCopyPixels进行像素复制的操作：
  - 直接从像素数据复制出新的像素数据，避免了多余的数据的格式转换，效率较高
  - 通过glRasterPos*系列函数来设置绘制的位置，因不涉及到主内存，不需要指定数据在内存中的格式和使用任何指针

glCopyPixels的用法

- glCopyPixels函数的（5个）参数：
  - 第一、二个参数表示复制像素来源矩形的左下角坐标
  - 第三、四个参数表示复制像素来源的矩形的宽度和高度
  - 第五个参数通常使用GL_COLOR，表示复制像素的颜色，也可以是GL_DEPTH或GL_STENCIL，分别表示复制深度缓冲数据或模板缓冲数据
  - 前两个函数中的各种操作，如glPixelZoom等，在该函数中同样有效

glCopyPixels的用法

- 例子：绘制一个三角形后，复制像素，并同时进行水平和垂直方向的翻转，然后缩小为原来的一半，并绘制。绘制完毕后，调用前面的grab函数，将屏幕中所有内容保存为grab.bmp

小结

- 结合Windows系统常见的BMP图象格式，简单介绍了OpenGL的像素处理功能及一些简单应用 ：
  - glReadPixels读取像素
  - glDrawPixels绘制像素
  - glCopyPixels复制像素
  - “外围”函数：glPixelStore*，glRasterPos*，以及glPixelZoom