**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。

常见的错误是，为mipmap过滤选项设置放大过滤，这样没有任何效果，因为mipmap主要使用在纹理缩小的情况下：纹理方法不会使用mipmap，为mipmap设置放大过滤选项会产生一个GL\_INVALID\_ENUM错误码。

**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);

**glTexParameteri(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。

**注意：如果不设置纹理过滤参数，则不会渲染出纹理**

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。

**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。