http://blog.csdn.net/damenhanter/article/details/22481563

http://blog.csdn.net/u013833399/article/details/47726067

http://blog.csdn.net/pizi0475/article/details/48547329



**切线空间**

直观的讲，模型顶点中的纹理坐标，就定义于切线空间，普通二维纹理坐标包含U、V两项，其中U坐标增长的方向，即切线空间中的tangent轴，V坐标增加的方向，为切线空间中的bitangent轴。模型中不同的三角形，都有对应的切线空间，其tangent轴和bitangent轴分别位于三角形坐在平面上，结合三角形面对应的法线，我们称为tangent轴（T/）、bitangent轴（B）及法线轴（N）所组成的坐标系，即切线空间（TBN）

**为什么需要切线空间？**

如果法线处于世界坐标系中（world space），那称为world space normal，如果是处于物体本身局部坐标中的，那称为object space normal

**world space normal**

一旦从贴图中解压出来，就可以直接用了，效率很高，但是有个缺点，这个world space normal是固定了，如果物体没有保持原来的方向和位置，那原来生成的normal map就作废了

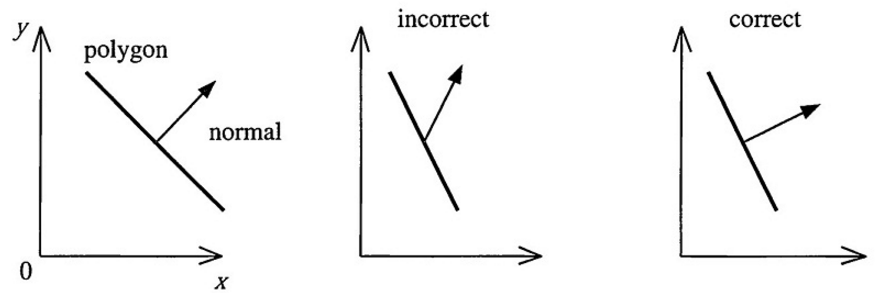
**object space normal**

它从贴图中解压出来，还需要乘以model-view矩阵转换到世界坐标，或者转化到其他坐标，object space normal生成的贴图，物体可以被旋转和位移，基本让人满意，但仍有一个缺点，就是一张贴图只能对应一个特定的模型，模型不能有变形（deform）

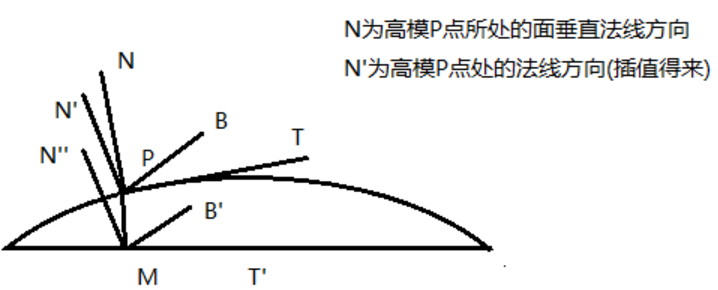
注意：这个的法线已经是一个点了，它需要经过mv的转换，其实也就不是直接指定模型顶点的法线，而是将法线保存在一张纹理图中，在这里就会法线非等比放缩会导致法线不垂直的问题。

变形时，顶点关系改变了，即面的形状，方向改变了，如果面上存在一个固定的坐标系，那当物体变形、移动、旋转时，这个坐标必定跟着面一起运动，那么在这个坐标系里的某个点或向量，不需要变动，当整个面发生变化时，我们只需要计算面上的坐标系到世界坐标系的转换矩阵，那么定义在这个面上的点或坐标（固定的），乘以这个矩阵即可，得到在世界中的坐标，这个坐标系就是tangent space

按照新方法每个面都有一个局部坐标系，当低模变形时，即三角形变化时，它的tangent space也会跟着变化，保存在贴图里的法线乘以低模这个面的tangent space到外部坐标系的转换矩阵即可得到外部坐标。



object space normal map中，低模的object space坐标系与高模中的object space坐标系是重合的，所以不需要构建，所以低模上某点才能直接用高模的法线替换自己的法线，坐标系重合这个概念很重要。新方法中，低模上的这个tangent space也必须与高模的坐标tangent space，因为低模上的一个面，可能对应高模上的几个面，按照新方法每个面都有一个局部坐标系，那对于低模上的每个面，高模上因为存在好几个面，就会出现好几个局部坐标系，这肯定是不行的，所以高模所用的tangent space就是低模上的，生成法线贴图，必定会确认高模上哪些面都对应低模上的哪个面，然后高模上的这几个面的法线，都会转换为低模这个面上所构建的tanget space的坐标。这样，当低模变形时候，即三角形变化时，它的tangent space也会跟着变化，保存在贴图里的法线乘以低模这个面的tangent space到外部坐标系的转换矩阵即可得到外部坐标。顺便再提一点，高模保存的这个法线是高模上object space的法线，看到这里你该明白这是自然而然的.你搜索文章时可能会看到什么把光转换到tangent space里,确保处于同一个坐标系下的话.确实是这样.但初次接触却还是模糊.我以为确保tangent sapce重合及做法,才是让人顿悟tangent space的诀窍点.



**在低模的每个面上，要构造出这个坐标系，这个坐标系术语里称为tangent space，所以高模上所用的tangent space，就是低模上的，生成法线贴图，必定会确认高模上的哪些面都对应低模上的哪个面，然后高模上的这个面的法线，都会转换为低模这个面上锁构建的tangent space的坐标。**

切线空间相对于模型空间更多的优点：

1. 自由度很高。模型空间下的法线纹理记录的是绝对法线信息，仅可用于创建它的那个模型，而应用到其他模型上就完全错误了。而切线空间下的法线纹理记录的是相对法线信息，这意味着，即便我们把纹理应用到一个完全不同的网格上，也可以得到一个合理的结果
2. 可进行uv动画。比如，我们移动一个纹理的uv坐标来实现一个凹凸移动的效果，但使用模型空间下的法线纹理会得到完全错误的结果。这种uv动画在水或者火山熔岩这种类型的物体上会经常用到
3. 可压缩。由于切线空间下的法线纹理中法线的z方向总是正方向，因此我们可以仅存储xy方向，从而推到得到z方向，而模型空间下的法线纹理由于每个方向都是可能的，因此必须存储3个方向的值，不可压缩。

**纹理坐标与位置坐标的关系**

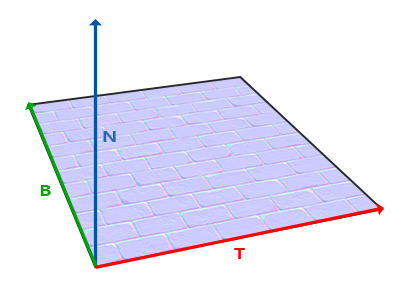
TBN矩阵将法线从切线空间坐标转向世界空间坐标

法线贴图中的法线向量在切线空间中，法线永远指着正z方向。切线空间是位于三角形表面之上的空间：法线相对于单个三角形的本地参考框架。它就像法线贴图向量的本地空间；它们都被定义为指向正z方向，无论最终变换到什么方向。使用一个特定的矩阵我们就能将本地/切线空寂中的法线向量转成世界或视图坐标，使它们转向到最终的贴图表面的方向。

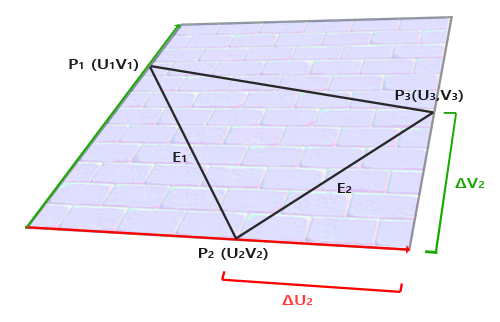
我们可以说，上个部分那个朝向正y的法线贴图错误的贴到了表面上。法线贴图被定义在切线空间中，所以一种解决问题的方式是计算出一种矩阵，把法线从切线空间变换到一个不同的空间，这样它们就能和表面法线方向对齐了：法线向量都会指向正y方向。切线空间的一大好处是我们可以为任何类型的表面计算出一个这样的矩阵，由此我们可以把切线空间的z方向和表面的法线方向对齐。

这种矩阵叫做TBN矩阵这三个字母分别代表tangent、bitangent和normal向量。这是建构这个矩阵所需的向量。要建构这样一个把切线空间转变为不同空间的变异矩阵，我们需要三个相互垂直的向量，它们沿一个表面的法线贴图对齐于：上、右、前；这和我们在摄像机教程中做的类似。

已知上向量是表面的法线向量。右和前向量是切线和副切线向量。下面的图片展示了一个表面的三个向量：



计算出切线和副切线并不像法线向量那么容易。从图中可以看到法线贴图的切线和副切线与纹理坐标的两个方向对齐。我们就是用到这个特性计算每个表面的切线和副切线的。需要用到一些数学才能得到它们；请看下图：



上图中我们可以看到边E2纹理坐标的不同，E2是一个三角形的边，这个三角形的另外两条边是U2和V2，它们与切线向量T和副切线向量B方向相同。这样我们可以把边E1和E2用切线向量T和副切线向量B的线性组合表示出来（译注：注意T和B都是单位长度，在TB平面中所有点的T、B坐标都在0到1之间，因此可以进行这样的组合）：

*E*2=Δ*U*2*T*+Δ*V*2*B*

我们也可以写成这样：

(*E*1*x*,*E*1*y*,*E*1*z*)=Δ*U*1(*Tx*,*Ty*,*Tz*)+Δ*V*1(*Bx*,*By*,*Bz*)

E是两个向量位置的差，U和V是纹理坐标的差。然后我们得到两个未知数（切线T和副切线B）和两个等式。你可能想起你的代数课了，这是让我们去接T和B。

上面的方程允许我们把它们写成另一种格式：矩阵乘法：

[*E*1*xE*2*xE*1*yE*2*yE*1*zE*2*z*]=[Δ*U*1Δ*U*2Δ*V*1Δ*V*2][*TxBxTyByTzBz*]

尝试会以一下矩阵乘法，它们确实是同一种等式。把等式写成矩阵形式的好处是，解T和B会因此变得很容易。两边都乘以UV的反数等于：

[Δ*U*1Δ*U*2Δ*V*1Δ*V*2]−1[*E*1*xE*2*xE*1*yE*2*yE*1*zE*2*z*]=[*TxBxTyByTzBz*]

这样我们就可以解出T和B了。这需要我们计算出delta纹理坐标矩阵的拟阵。我不打算讲解计算逆矩阵的细节，但大致是把它变化为，1除以矩阵的行列式，再乘以它的共轭矩阵。

[*TxBxTyByTzBz*]=1Δ*U*1Δ*V*2–Δ*U*2Δ*V*1[Δ*V*2−Δ*U*2−Δ*V*1Δ*U*1][*E*1*xE*2*xE*1*yE*2*yE*1*zE*2*z*]

有了最后这个等式，我们就可以用公式、三角形的两条边以及纹理坐标计算出切线向量T和副切线B。

如果你对这些数学内容不理解也不用担心。当你知道我们可以用一个三角形的顶点和纹理坐标（因为纹理坐标和切线向量在同一空间中）计算出切线和副切线你就已经部分地达到目的了（译注：上面的推导已经很清楚了，如果你不明白可以参考任意线性代数教材，就像作者所说的记住求得切线空间的公式也行，不过不管怎样都得理解切线空间的含义）。

**手工计算切线和副切线**

这个教程的demo场景中有一个简单的2D平面，它朝向正z方向。这次我们会使用切线空间来实现法线贴图，所以我们可以使平面朝向任意方向，法线贴图仍然能够工作。使用前面讨论的数学方法，我们来手工计算出表面的切线和副切线向量。

假设平面使用下面的向量建立起来（1、2、3和1、3、4，它们是两个三角形）：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | // positions  glm::vec3 pos1(-1.0,  1.0, 0.0);  glm::vec3 pos2(-1.0, -1.0, 0.0);  glm::vec3 pos3(1.0, -1.0, 0.0);  glm::vec3 pos4(1.0, 1.0, 0.0);  // texture coordinates  glm::vec2 uv1(0.0, 1.0);  glm::vec2 uv2(0.0, 0.0);  glm::vec2 uv3(1.0, 0.0);  glm::vec2 uv4(1.0, 1.0);  // normal vector  glm::vec3 nm(0.0, 0.0, 1.0); |

我们先计算第一个三角形的边和deltaUV坐标：



|  |  |
| --- | --- |
| 1  2  3  4 | glm::vec3 edge1 = pos2 - pos1;  glm::vec3 edge2 = pos3 - pos1;  glm::vec2 deltaUV1 = uv2 - uv1;  glm::vec2 deltaUV2 = uv3 - uv1; |

有了计算切线和副切线的必备数据，我们就可以开始写出来自于前面部分中的下列等式：

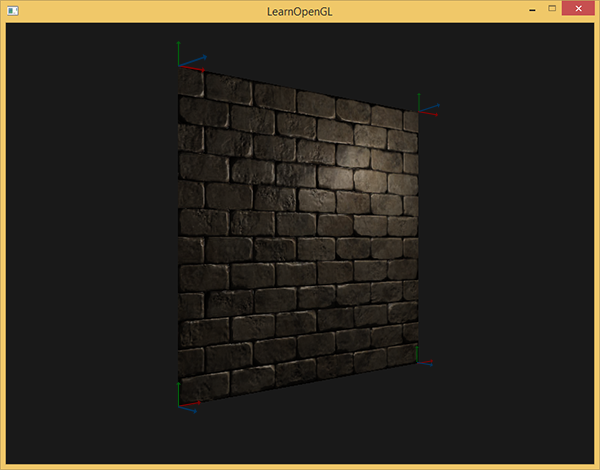


|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | GLfloat f = 1.0f / (deltaUV1.x \* deltaUV2.y - deltaUV2.x \* deltaUV1.y);    tangent1.x = f \* (deltaUV2.y \* edge1.x - deltaUV1.y \* edge2.x);  tangent1.y = f \* (deltaUV2.y \* edge1.y - deltaUV1.y \* edge2.y);  tangent1.z = f \* (deltaUV2.y \* edge1.z - deltaUV1.y \* edge2.z);  tangent1 = glm::normalize(tangent1);    bitangent1.x = f \* (-deltaUV2.x \* edge1.x + deltaUV1.x \* edge2.x);  bitangent1.y = f \* (-deltaUV2.x \* edge1.y + deltaUV1.x \* edge2.y);  bitangent1.z = f \* (-deltaUV2.x \* edge1.z + deltaUV1.x \* edge2.z);  bitangent1 = glm::normalize(bitangent1);    [...] // similar procedure for calculating tangent/bitangent for plane's second triangle |

我们预先计算出等式的分数部分f，然后把它和每个向量的元素进行相应矩阵乘法。如果你把代码和最终的等式对比你会发现，这就是直接套用。最后我们还要进行标准化，来确保切线/副切线向量最后是单位向量。

因为一个三角形永远是平坦的形状，我们只需为每个三角形计算一个切线/副切线，它们对于每个三角形上的顶点都是一样的。要注意的是大多数实现通常三角形和三角形之间都会共享顶点。这种情况下开发者通常将每个顶点的法线和切线/副切线等顶点属性平均化，以获得更加柔和的效果。我们的平面的三角形之间分享了一些顶点，但是因为两个三角形相互并行，因此并不需要将结果平均化，但无论何时只要你遇到这种情况记住它就是件好事。

最后的切线和副切线向量的值应该是(1, 0, 0)和(0, 1, 0)，它们和法线(0, 0, 1)组成相互垂直的TBN矩阵。在平面上显示出来TBN应该是这样的：



每个顶点定义了切线和副切线向量，我们就可以开始实现正确的法线贴图了。

**切线空间法线贴图**

为让法线贴图工作，我们先得在着色器中创建一个TBN矩阵。我们先将前面计算出来的切线和副切线向量传给顶点着色器，作为它的属性：



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | #version 330 core  layout (location = 0) in vec3 position;  layout (location = 1) in vec3 normal;  layout (location = 2) in vec2 texCoords;  layout (location = 3) in vec3 tangent;  layout (location = 4) in vec3 bitangent; |

在顶点着色器的main函数中我们创建TBN矩阵：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | void main()  {     [...]     vec3 T = normalize(vec3(model \* vec4(tangent,   0.0)));     vec3 B = normalize(vec3(model \* vec4(bitangent, 0.0)));     vec3 N = normalize(vec3(model \* vec4(normal,    0.0)));     mat3 TBN = mat3(T, B, N)  } |

我们先将所有TBN向量变换到我们所操作的坐标系中，现在是世界空间，我们可以乘以model矩阵。然后我们创建实际的TBN矩阵，直接把相应的向量应用到mat3构造器就行。注意，如果我们希望更精确的话就不要讲TBN向量乘以model矩阵，而是使用法线矩阵，但我们只关心向量的方向，不会平移也和缩放这个变换。

从技术上讲，顶点着色器中无需副切线。所有的这三个TBN向量都是相互垂直的所以我们可以在顶点着色器中庸T和N向量的叉乘，自己计算出副切线：vec3 B = cross(T, N);

现在我们有了TBN矩阵，如果来使用它呢？基本有两种方式可以使用，我们会把这两种方式都说明一下：

1. 我们可以用TBN矩阵把所有向量从切线空间转到世界空间，传给像素着色器，然后把采样得到的法线用TBN矩阵从切线空间变换到世界空间；法线就处于和其他光照变量一样的空间中了。
2. 我们用TBN的逆矩阵把所有世界空间的向量转换到切线空间，使用这个矩阵将除法线以外的所有相关光照变量转换到切线空间中；这样法线也能和其他光照变量处于同一空间之中。

我们来看看第一种情况。我们从法线贴图重采样得来的法线向量，是以切线空间表达的，尽管其他光照向量是以世界空间表达的。把TBN传给像素着色器，我们就能将采样得来的切线空间的法线乘以这个TBN矩阵，将法线向量变换到和其他光照向量一样的参考空间中。这种方式随后所有光照计算都可以简单的理解。

把TBN矩阵发给像素着色器很简单：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | out VS\_OUT {      vec3 FragPos;      vec2 TexCoords;      mat3 TBN;  } vs\_out;    void main()  {      [...]      vs\_out.TBN = mat3(T, B, N);  } |

在像素着色器中我们用mat3作为输入变量：



|  |  |
| --- | --- |
| 1  2  3  4  5 | in VS\_OUT {      vec3 FragPos;      vec2 TexCoords;      mat3 TBN;  } fs\_in; |

有了TBN矩阵我们现在就可以更新法线贴图代码，引入切线到世界空间变换：



|  |  |
| --- | --- |
| 1  2  3 | normal = texture(normalMap, fs\_in.TexCoords).rgb;  normal = normalize(normal \* 2.0 - 1.0);  normal = normalize(fs\_in.TBN \* normal); |

因为最后的normal现在在世界空间中了，就不用改变其他像素着色器的代码了，因为光照代码就是假设法线向量在世界空间中。

我们同样看看第二种情况，我们用TBN矩阵的逆矩阵将所有相关的世界空间向量转变到采样所得法线向量的空间：切线空间。TBN的建构还是一样，但我们在将其发送给像素着色器之前先要求逆矩阵：



|  |  |
| --- | --- |
| 1 | vs\_out.TBN = transpose(mat3(T, B, N)); |

注意，这里我们使用transpose函数，而不是inverse函数。正交矩阵（每个轴既是单位向量同时相互垂直）的一大属性是一个正交矩阵的置换矩阵与它的逆矩阵相等。这个属性和重要因为逆矩阵的求得比求置换开销大；结果却是一样的。

在像素着色器中我们不用对法线向量变换，但我们要把其他相关向量转换到切线空间，它们是lightDir和viewDir。这样每个向量还是在同一个空间（切线空间）中了。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | void main()  {      vec3 normal = texture(normalMap, fs\_in.TexCoords).rgb;      normal = normalize(normal \* 2.0 - 1.0);        vec3 lightDir = fs\_in.TBN \* normalize(lightPos - fs\_in.FragPos);      vec3 viewDir  = fs\_in.TBN \* normalize(viewPos - fs\_in.FragPos);      [...]  } |

第二种方法看似要做的更多，它还需要在像素着色器中进行更多的乘法操作，所以为何还用第二种方法呢？

将向量从世界空间转换到切线空间有个额外好处，我们可以把所有相关向量在顶点着色器中转换到切线空间，不用在像素着色器中做这件事。这是可行的，因为lightPos和viewPos不是每个fragment运行都要改变，对于fs\_in.FragPos，我们也可以在顶点着色器计算它的切线空间位置。基本上，不需要把任何向量在像素着色器中进行变换，而第一种方法中就是必须的，因为采样出来的法线向量对于每个像素着色器都不一样。

所以现在不是把TBN矩阵的逆矩阵发送给像素着色器，而是将切线空间的光源位置，观察位置以及顶点位置发送给像素着色器。这样我们就不用在像素着色器里进行矩阵乘法了。这是一个极佳的优化，因为顶点着色器通常比像素着色器运行的少。这也是为什么这种方法是一种更好的实现方式的原因。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | out VS\_OUT {      vec3 FragPos;      vec2 TexCoords;      vec3 TangentLightPos;      vec3 TangentViewPos;      vec3 TangentFragPos;  } vs\_out;    uniform vec3 lightPos;  uniform vec3 viewPos;    [...]    void main()  {      [...]      mat3 TBN = transpose(mat3(T, B, N));      vs\_out.TangentLightPos = TBN \* lightPos;      vs\_out.TangentViewPos  = TBN \* viewPos;      vs\_out.TangentFragPos  = TBN \* vec3(model \* vec4(position, 0.0));  } |

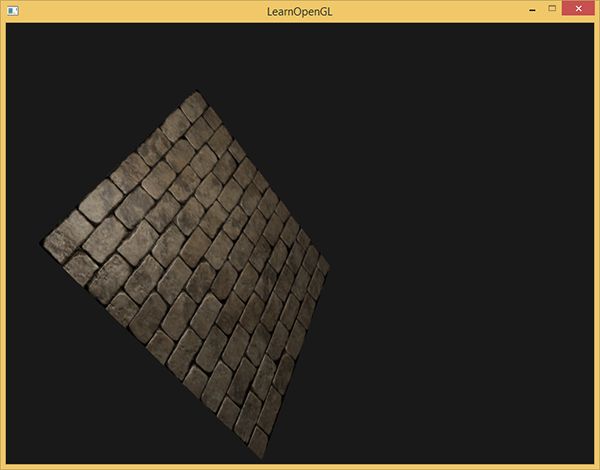
在像素着色器中我们使用这些新的输入变量来计算切线空间的光照。因为法线向量已经在切线空间中了，光照就有意义了。

将法线贴图应用到切线空间上，我们会得到混合教程一开始那个例子相似的结果，但这次我们可以将平面朝向各个方向，光照一直都会是正确的：



|  |  |
| --- | --- |
| 1  2  3  4 | glm::mat4 model;  model = glm::rotate(model, (GLfloat)glfwGetTime() \* -10, glm::normalize(glm::vec3(1.0, 0.0, 1.0)));  glUniformMatrix4fv(modelLoc 1, GL\_FALSE, glm::value\_ptr(model));  RenderQuad(); |

看起来是正确的法线贴图：



你可以在这里找到[源代码](http://www.learnopengl.com/code_viewer.php?code=advanced-lighting/normal_mapping)、[顶点](http://www.learnopengl.com/code_viewer.php?code=advanced-lighting/normal_mapping&type=vertex)和[像素](http://www.learnopengl.com/code_viewer.php?code=advanced-lighting/normal_mapping&type=fragment)着色器。

