# 读取文件

**顶点的法向，是包含该顶点的所有三角形的法向的均值。**

对于结构中的每个顶点，找到使用顶点的三角形。

计算这些三角形的法线向量：对于结构中的每个三角形，取三角形的两条边，通过叉乘获取垂直这个三角形法线向量。将这个向量添加到三角形的三个顶点的法线中。

求所有法线向量的平均值。

将这个平均法线存储到顶点中：求平均值的过程是必须的，因为你总是要归一化存储在顶点中的法线向量（在一个结构中，所有顶点被几个三角形共享。要获取平滑的效果，每个顶点需要存储周围三角形所有法线的平均值）

对所有三角形进行这个操作后，执行以下操作，归一化结构中的每个顶点的法线向量。

void ReadDataRabbit()

{

std::ifstream inFile(DataFileName);

if (inFile.fail())

{

std::cout << "Error opening file" << std::endl;

exit(1);

}

inFile >> PointNum >> TriNum; //读入顶点数和面片数

for (int i = 0; i < PointNum; i++) //i是第几个顶点

{

inFile >> vertices[6 \* i + 0] >> vertices[6 \* i + 1] >> vertices[6 \* i + 2];

}

for (int i = 0; i < TriNum; i++)//第几个面片

{

GLuint tmp\_num, in1, in2, in3;

inFile >> tmp\_num >> in1 >> in2 >> in3;

indices[i \* 3] = in1; //读到面片数据了

indices[i \* 3 + 1] = in2;

indices[i \* 3 + 2] = in3;

if (tmp\_num != 3)

{

std::cout << "Error file format" << std::endl;

exit(1);

}

glm::vec3 triangleOne = glm::vec3(vertices[in2 \* 6] - vertices[in1 \* 6], vertices[in2 \* 6 + 1] - vertices[in1 \* 6 + 1], vertices[in2 \* 6 + 2] - vertices[in1 \* 6 + 2]);

glm::vec3 triangleTwo = glm::vec3(vertices[in3 \* 6] - vertices[in2 \* 6], vertices[in3 \* 6 + 1] - vertices[in2 \* 6 + 1], vertices[in3 \* 6 + 2] - vertices[in2 \* 6 + 2]);

glm::vec3 triangleNormal = glm::normalize(glm::cross(triangleOne, triangleTwo));//两个向量cross叉乘的结果会同时垂直于两向量

Normalize()函数是让一个向量（任意向量，不一定必须是normal）除以其长度，从而使新长度为1。

法向(normal)则是某一类向量的名字。

//把刚求的三角面片法向量存储到对应顶点法线数组

vecNormal[in1] += triangleNormal;

vecNormal[in2] += triangleNormal;

vecNormal[in3] += triangleNormal;

}

-----------------------

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

{ vecNormal[i] = glm::normalize(vecNormal[i]); //归一化：P194 Gouraud着色

}

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

{

vertices[6 \* i + 3] = vecNormal[i].x; //各坐标分量

vertices[6 \* i + 4] = vecNormal[i].y;

vertices[6 \* i + 5] = vecNormal[i].z;

} inFile.close(); }

# 顶点数组对象VAO

/\*当我们打算绘制一个物体的时候，

\*我们只要在绘制物体前简单地把VAO绑定到希望使用的设定上就行了\*/

你需要创建一个顶点数组对象，并将它设为当前对象

GLuint VertexArrayID;

glGenVertexArrays(1, &VertexArrayID);

glBindVertexArray(VertexArrayID);

当窗口创建成功后（即OpenGL上下文创建后），马上做这一步工作；必须在任何其他OpenGL调用前完成。

glBindVertexArray binds the vertex array object with name array. array is the name of a vertex array object previously returned from a call to glGenVertexArrays, or zero to break the existing vertex array object binding.

# 顶点缓冲对象VBO (Vertex Buffer Objects)

**glBufferData**是一个专门用来把用户定义的数据复制到当前绑定缓冲的函数。

它的第一个参数是目标缓冲的类型：顶点缓冲对象当前绑定到GL\_ARRAY\_BUFFER目标上。

第二个参数指定传输数据的大小(以字节为单位)；用一个简单的sizeof计算出顶点数据大小就行。

第三个参数是我们希望发送的实际数据。

第四个参数指定了我们希望显卡如何管理给定的数据。它有三种形式：

• GL\_STATIC\_DRAW ：数据不会或几乎不会改变。

• GL\_DYNAMIC\_DRAW：数据会被改变很多。

• GL\_STREAM\_DRAW ：数据每次绘制时都会改变。

[**https://blog.csdn.net/baidu\_28949227/article/details/91865401**](https://blog.csdn.net/baidu_28949227/article/details/91865401)

# 索引缓冲对象EBO（Element Buffer Object）

索引缓冲对象EBO相当于OpenGL中的顶点数组的概念，是为了解决同一个顶点多次重复调用的问题，可以减少内存空间浪费，提高执行效率。当需要使用重复的顶点时，通过顶点的位置索引来调用顶点，而不是对重复的顶点信息重复记录，重复调用。

**EBO中存储的内容就是顶点位置的索引indices，EBO跟VBO类似，也是在显存中的一块内存缓冲器**，只不过EBO保存的是顶点的索引。

[**https://blog.csdn.net/dcrmg/article/details/53556664**](https://blog.csdn.net/dcrmg/article/details/53556664)

# 画三角形

**glVertexAttribPointer()**

第一个参数指定顶点属性位置，与顶点着色器中layout(location=0)对应。

第二个参数指定顶点属性大小。

第三个参数指定数据类型。

第四个参数定义是否希望数据被标准化。

第五个参数是步长（Stride），指定在连续的顶点属性之间的间隔。

第六个参数表示我们的位置数据在缓冲区起始位置的偏移量

# OpengGL的Draw函数

* **glDrawArrays**(GLenum mode, GLint first, GLsizei count)
* **glDrawElements**(GLenum mode, GLsizei count, GLenum type, GLvoid\* indices)

**2.1.1 glDrawArrays**

为最常用的Draw函数，按mode指定的图元类型，画出vertex buffer中第first个起的count个顶点。

**2.1.2 glDrawElements**

使用glDrawElements前，除了类似glDrawArrays一样准备好VAO和VBO外，还需要产生并绑定EBO(Element Buffer Object），注入的数据为一个类型为*type*的数组，其元素表示vertex buffer中顶点索引。glDrawElement以按照指定的图元类型*mode*,以其前*count*个元素作为VBO中顶点数据的索引。

DrawElements同样用6个顶点画出了4个三角形，用DrawArrays则需要12个顶点。当vertex buffer中的顶点会被多个图元频繁使用时，DrawElements比DrawArrays更为高效。

# DeltaTime

[OpenGL学习笔记（八）摄像机 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/108883986)

移动速度如果设置成常量的话，会因为使用者配置的不同，有些人可能移动很快，而有些人会移动很慢，很影响使用；

程序会跟踪一个时间差(Deltatime)变量，它储存了渲染上一帧所用的时间。我们把所有速度都去乘以deltaTime值。结果就是，如果我们的deltaTime很大，就意味着上一帧的渲染花费了更多时间，所以这一帧的速度需要变得更高来平衡渲染所花去的时间。使用这种方法时，无论你的电脑快还是慢，摄像机的速度都会相应平衡，这样每个用户的体验就都一样了。

**数值按照每秒多少进行变化时就把这个数值乘以 Time.deltaTime 就可以了**

在渲染循环中计算deltatime，后在物体运动时总是用速度乘以deltatime进行运动，可以实现在不同的帧速率下达到相同的运行速度。deltatime计算可通过以下方式得到：

float deltatime =0.0f;

float lastframe = 0.0f;

...

renderloop()

{

float currectframe = glfwGetTime();

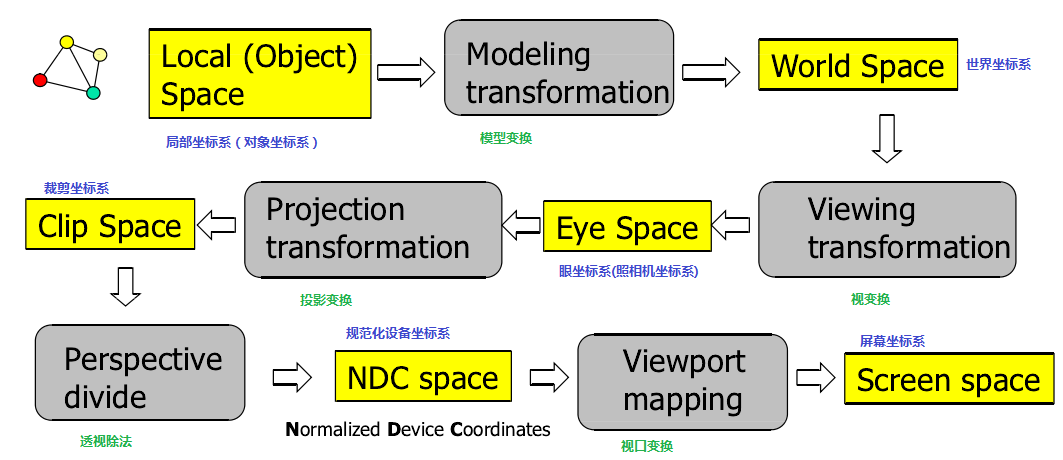
deltatime = currectframe -lastframe;

lastframe = currectframe;

...

}

# 鼠标屏幕坐标到三维世界坐标的转化



## 坐标空间

**Local Space**

local Space，也就是本地坐标空间。将物体本身作为一个坐标空间。一般选取物体的中心点作为坐标原点。在openGL中没有进行过任何变换的顶点信息可以看作是属于Local Space的。

**World Space**

在某个空间中多个物体都相对于同一个坐标系，那么这个空间也就可以被成为world space。表现在代码中就是，物体的顶点坐标乘以了一个**model**矩阵。

**View Space/Camera Space**

View Space则是基于World Space，将世界坐标中的某一个物体作为参考系，其余物体的空间位置，与该物体的空间位置相关。表现在代码的基础上就是，在World Space的基础上再乘以一个**view**矩阵。

**Clip Space**

Clip Space 则是再View Space的基础上根据投影关系，所截取的某个投影平面。表现在代码上就是，再View Space的基础上再乘以一个projection矩阵。

被裁剪掉的坐标就被忽略了，所以剩下的坐标就将变为屏幕上可见的片段。这也就是裁剪空间(Clip Space)名字的由来。

为了将顶点坐标从观察空间转换到裁剪空间，我们需要定义一个投影矩阵(Projection Matrix)，它指定了坐标的范围。

由投影矩阵创建的观察区域(Viewing Box)被称为平截头体(Frustum)，且每个出现在平截头体范围内的坐标都会最终出现在用户的屏幕上。将一定范围内的坐标转化到标准化设备坐标系的过程(而且它很容易被映射到2D观察空间坐标)被称之为投影(Projection)，因为使用投影矩阵能将3维坐标投影(Project)到很容易映射的2D标准化设备坐标系中。

一旦所有顶点被转换到裁剪空间，最终的操作——透视划分(Perspective Division)将会执行，在这个过程中我们将位置向量的x，y，z分量分别除以向量的齐次w分量；透视划分是将4维裁剪空间坐标转换为3维标准化设备坐标。

**Screen Space**

裁剪后需要进行真正的投影，需要把**视椎体**投影到**屏幕空间空(screen space)**，最后会得到像素位置。

将顶点从裁剪空间[投影](https://zh.wikipedia.org/wiki/%E6%8A%95%E5%BD%B1)到屏幕空间，来生成2D坐标。

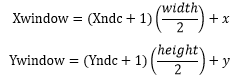
第一步，需要进行**齐次除法(homogeneous diision)**  
，就是使用齐次坐标系中的w分量去除以x,y,z。x,y,z分量，在OpenGL中这一步得到的坐标是**归一化的设备坐标(Normalized Device Coordinated, NDC)**.NDC坐标是三分量的坐标，只包含xyz信息，不再包含w信息。归一化后的NDC坐标中的x、y、z的取值范围都是[-1,1]。x、y的取值范围是[-1,1]，所表达的含义是将屏幕的中心点对应于[0,0],屏幕左下点对应[-1,-1]，屏幕右上点对应于[1,1]，x和y能够表示出该点相对于屏幕中心原点的位置。

把坐标从齐次裁剪坐标中转化到NDC中，最后的裁剪空间会变换到一个立方体中。分量范围在[-1, 1]。

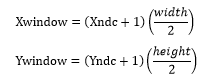
**Window (or screen) coordinates（屏幕坐标系）**

最后系统需要根据NDC坐标将其画到我们的屏幕上，此处不再通过矩阵来完成了，而是通过调用命令glViewPort（x, y, width, height）。  
width、height指定了视口的宽度和高度。当OpenGL上下文第一次绑定到window时，width和height被默认设置为window的宽度和高度。

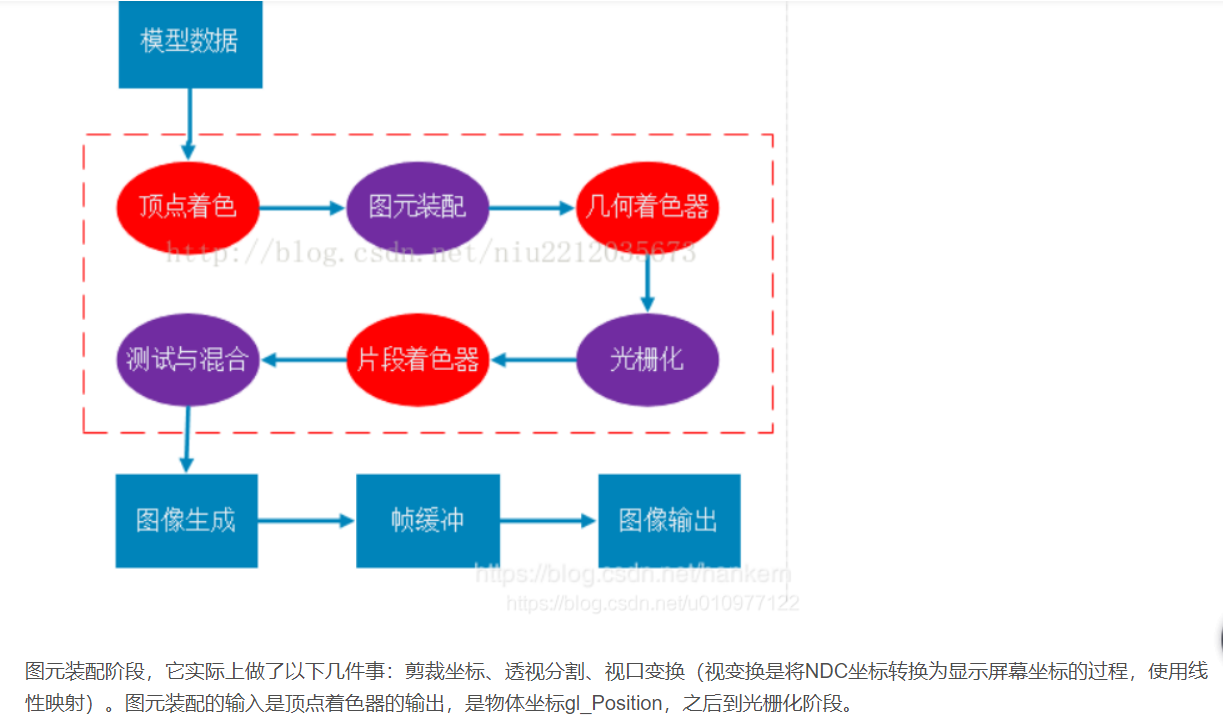
屏幕坐标是以像素来定义的, 屏幕左下角为原点(0,0), 右上角为(Screen.width,Screen.height)  
glViewport指定了x、y从NDC坐标系（归一化设备坐标系）到屏幕坐标系的仿射变换。如果(Xndc,Yndc)是归一化坐标，那么屏幕坐标(Xwindow,Ywindow)计算如下:



一般情况下，在调用glViewPort时，我们一般使用glViewPort(0, 0, width, height)，也就是前两个偏移坐标都为0，所以上式简化为：

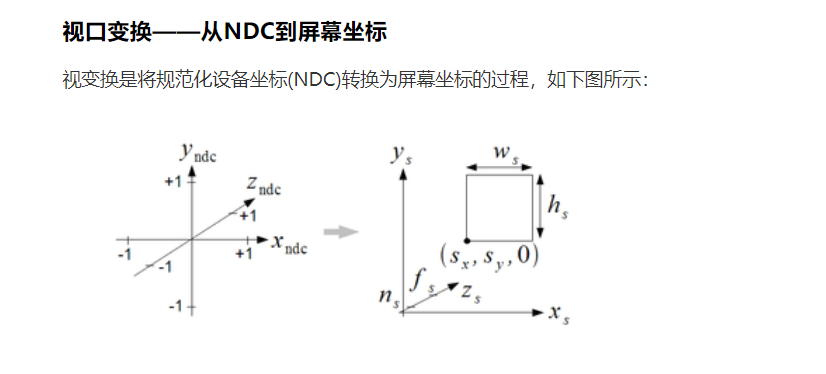


这些变换矩阵在程序中可以使用**glm:mat4**来进行表示，该类型的变量默认会生成一个4x4的单位矩阵。



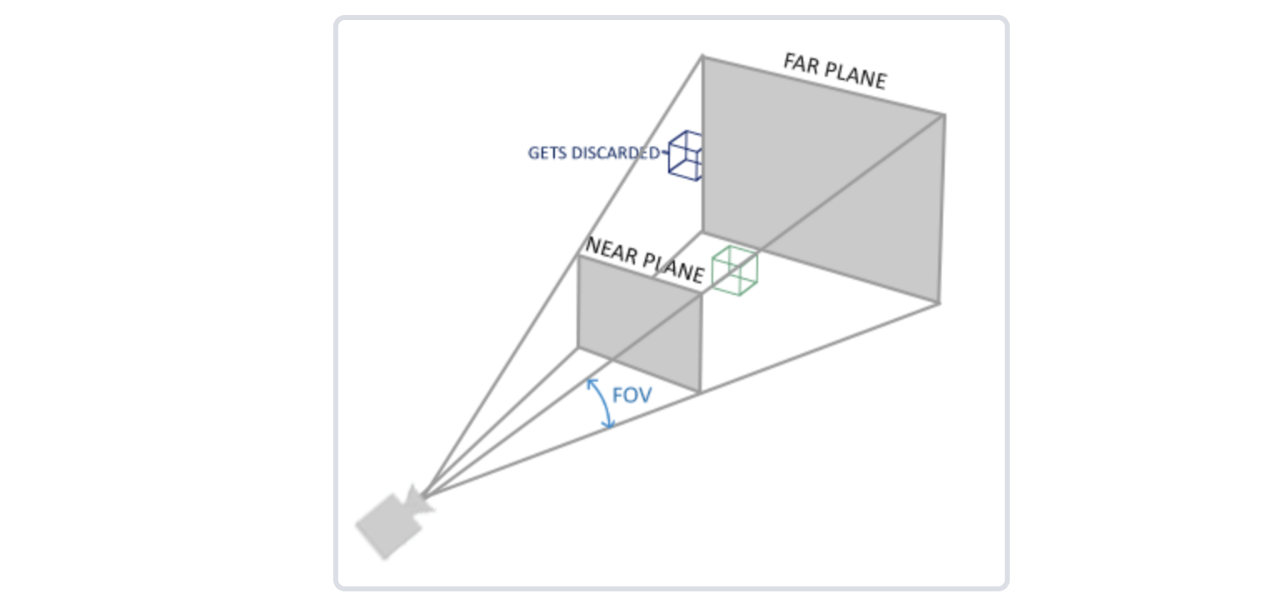
**透视除法**

对上面的剪裁坐标的点的x、y、z坐标除以它的w分量，除以w的坐标叫做归一化设备坐标。如果w分量大，除以w后的点就接近(0,0,0)，在三维空间中，距离我们较远的坐标如果它的w分量较大，进行透视除法后，就距离原点越近，原点作为远处物体的消失点，就有三维场景的效果。



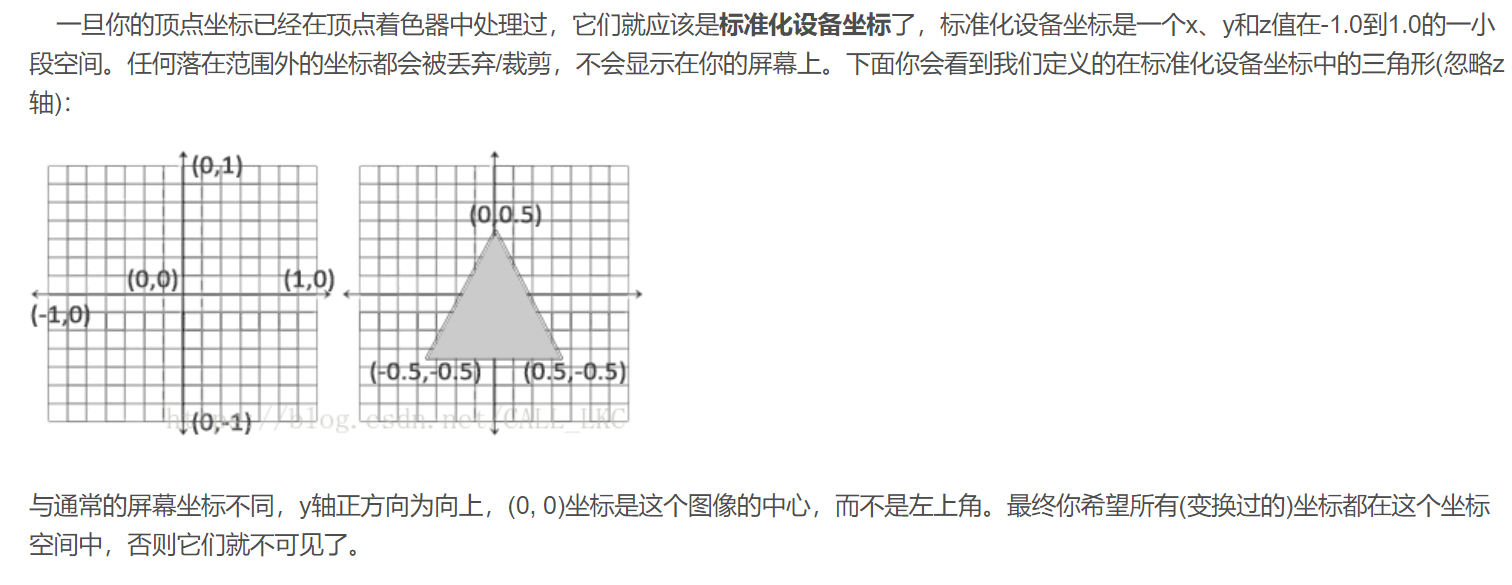
## 透视投影

离你越远的东西看起来更小，这个效果我们称之为透视(Perspective)。   
OpenGL要求所有可见的坐标都落在-1.0到1.0范围内从而作为最后的顶点着色器输出，因此一旦坐标在裁剪空间内，透视划分就会被应用到裁剪空间坐标：   
out=（x/w，y/w，z/w）   
在GLM中可以这样创建一个透视投影矩阵：   
glm::mat4 proj = glm::perspective(45.0f, (float)width / (float)height, 0.1f, 100.0f);   
glm::perspective所做的其实就是再次创建了一个定义了可视空间的大的平截头体，任何在这个平截头体的对象最后都不会出现在裁剪空间体积内，并且将会受到裁剪。   
它的第一个参数定义了fov的值，它表示的是视野(Field of View)，并且设置了观察空间的大小。对于一个真实的观察效果，它的值经常设置为45.0，但想要看到更多结果你可以设置一个更大的值。第二个参数设置了宽高比，由视口的高除以宽。第三和第四个参数设置了平截头体的近和远平面。我们经常设置近距离为0.1而远距离设为100.0。所有在近平面和远平面内且处于平截头体内的顶点都会被渲染。



# 渲染立方体

要渲染一个立方体，我们一共需要36个顶点(6个面 x 每个面有2个三角形组成 x 每个三角形有3个顶点)，这36个顶点的位置如下



# 回调函数

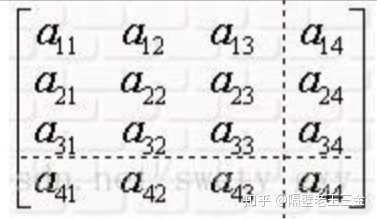
显示回调函数只有应用程序或操作系统确定显示窗口中的图形需要重新绘制时才会被调用。

# 光照

# 拾取

顶点代表的是一个点，在齐次坐标系下(x,y,z,1)，法线代表的是方向，在齐次坐标系下(x,y,z,0)。

* 我们知道一个4x4的变换矩阵，其平移的数据存放在第4列中即a14、a24、a34。而我们知道法线最后求得齐次形式是(x,y,z,0)，所以很明显变换矩阵的第四行和第四列都不会对法线产生影响。



*那我们只需要考虑变换矩阵的前3x3就行了咯。*

法向量的变换矩阵是顶点变换矩阵前3\*3的逆转置矩阵

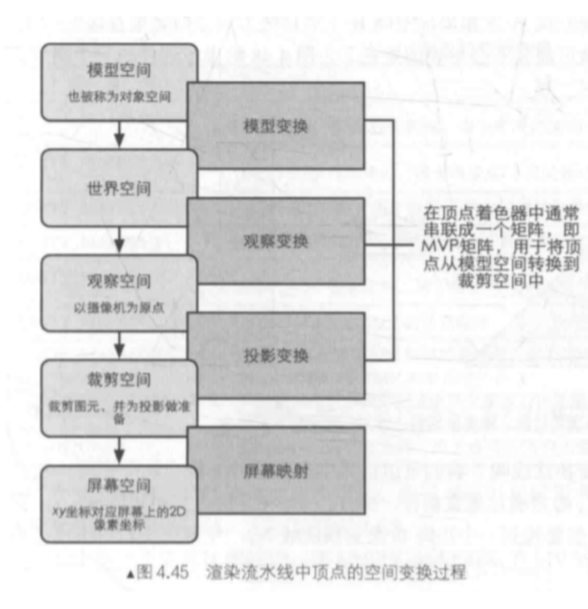
法线矩阵

**法线矩阵：**「模型矩阵左上角的逆矩阵的转置矩阵」。由于只在世界空间中进行操作（不是在观察空间），只使用模型矩阵。

在顶点着色器中，可以使用inverse和transpose函数自己生成法线矩阵，这两个函数对所有类型矩阵都有效。注意还要把被处理过的矩阵强制转换为3×3矩阵，来保证它失去了位移属性以及能够乘以vec3的法向量。

Normal = mat3(transpose(inverse(model))) \* aNormal;

# 着色器

顶点着色器的最基本任务就是把顶点坐标从模型空间转换到裁剪空间中。

在 OpenGL 中，在顶点 shader 作用了每一个顶点后，自动进行透视除法。这就是顶点 shader 中 main 方法输出的 gl\_position 变量，是4维向量，而不是3维向量。

## Bunny.vs

gl\_Position是一个内置的顶点着色器输出变量

#version 330 core

layout (location = 0) in vec3 aPos; //位置变量属性=0

layout (location = 1) in vec3 aNormal; //法线变量属性=1

out vec3 FragPos;

out vec3 Normal;

uniform mat4 model;

uniform mat4 view;

uniform mat4 projection;

void main(){

FragPos = vec3(model \* vec4(aPos, 1.0));

Normal = mat3(transpose(inverse(model))) \* aNormal;

gl\_Position = projection \* view \* vec4(FragPos, 1.0);

}

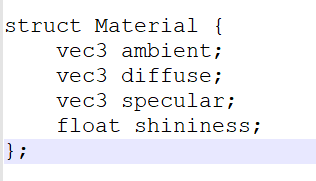
## Bunny.fs

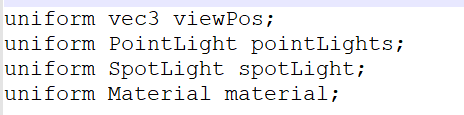
在片元着色器中实现光照计算

当描述一个物体的时候，我们可以用这三个分量来定义一个材质颜色(Material Color)：环境光照(Ambient Lighting)、漫反射光照(Diffuse Lighting)和镜面光照(Specular Lighting)。通过为每个分量指定一个颜色，我们就能够对物体的颜色输出有着精细的控制了。现在，我们再添加反光度(Shininess)这个分量到上述的三个颜色中，这就有我们需要的所有材质属性了：

要模拟不同的物体接受光照后的效果，就需要考虑物体的材质属性，利用材质属性模拟出不同的效果。

（总之，我们现在要定义一个材质属性，包括我们之前用到的ambient，diffuse，specular属性，在fragmentshader中用结构体来进行定义。）





物体的每个材质属性都乘上了它们对应的光照分量。

# 函数

## glViewport(0, 0, width, height);

// 确保视口与新的窗口尺寸匹配，请注意，窗口的宽度和高度将大于视口在显示屏上的指定的宽和高

//glViewport函数前两个参数控制窗口左下角的位置。第三个和第四个参数控制渲染窗口的宽度和高度（像素），这里是直接从GLFW中获取的。

坐标系是Y轴朝上，左下角是0，X轴朝右，左下角是0.都是整数。