向量

既有大小又有方向的量就叫做向量。大小就是向量的长度，也叫做模。**零向量长度是0，方向任意。**

单位向量是相对另一个向量的概念，向量相加，第一个的终点与第二个的起点连接。

```
const vector1 = new THREE.Vector3(1, 0, 0);
const vector2 = new THREE.Vector3(0, 1, 0);
const vector3 = new THREE.Vector3();
vector3.addVectors(vector1, vector2);
```

点乘dot 和叉乘 cross

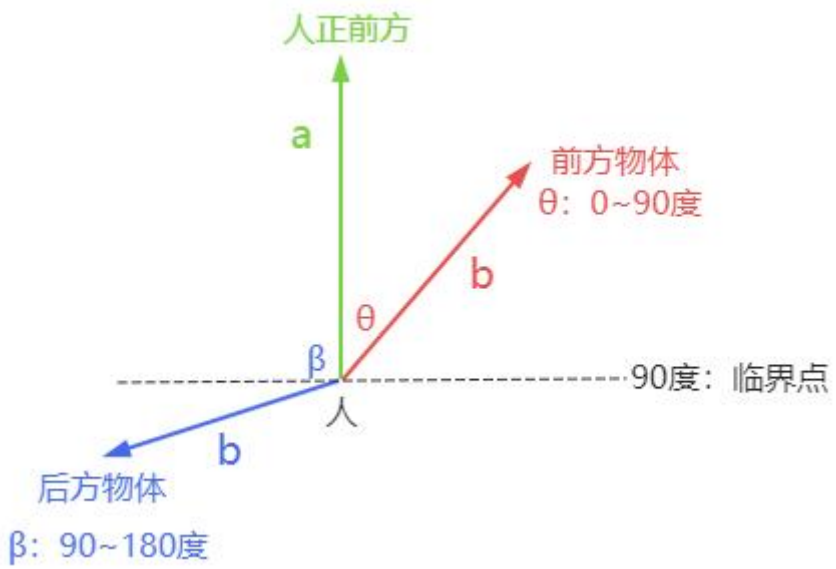
向量点乘、叉乘

```
const dot = dot(v1, v2) = v1 * v2 = |v1| * |v2| * Cos(θ);
cosθ = dot(v1,v2) / |v1| * |v2|;
θ = acos(dot(v1,v2) / |v1| * |v2|);
```

几何含义是向量a在向量b上投影长度与向量b相乘，或者说 `向量a长度 * 向量b长度 * cos(ab夹角)`

- A * B > 0 方向基本相同夹角在90°
- A * B = 0 正交垂直
- A * B < 0 方向相反夹角在90°到180

用处



- 判断物体在人前还是人后

  0 ~ 90度：物体在人的前方,人指向物体的向量b与人正前方向量a夹角处于0 ~ 90之间。

  90 ~ 180度：物体在人的后方，人指向物体的向量b与人正前方向量a夹角处于90 ~ 180之间。
- 判断物体是否在扇形内
- 光线向量跟物体的法线点乘可以计算出光照阴影。

叉乘 cross

右手螺旋定则。

食指指向第一个向量V1，中指指向第二个向量V2。大拇指的方向就是 叉乘结果向量的方向。 cross(V1 * V2);

```
const cross = cross(V1, V2) = |V1| * |V2| * sin(θ);
```

- 判断点是否在三角形内
- 判断两个点是否在线段的同一侧
- 计算三角形的面积
- 游戏中判断方向
- 求法线矢量 也需要叉乘
- 两个向量的关系。

四元数 Quaternion

四元数`Quaternion`和欧拉角`Euler`一样，可以用来计算或表示物体在3D空间中的旋转姿态角度

`.setFromAxisAngle()`是四元数的一个方法，可以用来辅助生成表示特定旋转的四元数。`.setFromAxisAngle(axis, angle)`生成的四元数表示绕axis旋转，旋转角度是angle。

```
const quaternion = new THREE.Quaternion();
// 旋转轴new THREE.Vector3(0,0,1)
// 旋转角度Math.PI/2
// 绕z轴旋转90度
quaternion.setFromAxisAngle(new THREE.Vector3(0,0,1),Math.PI/2);
```

四元数`Quaternion`的方法`.setFromUnitVectors(a, b)`可以通过两个向量参数a和b，创建一个四元数，表示从向量a表示的方向旋转到向量b表示的方向。(参数a, b是单位向量)

```
//飞机初始姿态飞行方向a
const a = new THREE.Vector3(0, 0, -1);
// 飞机姿态绕自身坐标原点旋转到b指向的方向
const b = new THREE.Vector3(-1, -1, -1).normalize();
// a旋转到b构成的四元数
const quaternion = new THREE.Quaternion();
//注意两个参数的顺序
quaternion.setFromUnitVectors(a, b);
// quaternion表示的是变化过程，在原来基础上乘以quaternion即可
fly.quaternion.multiply(quaternion);
```

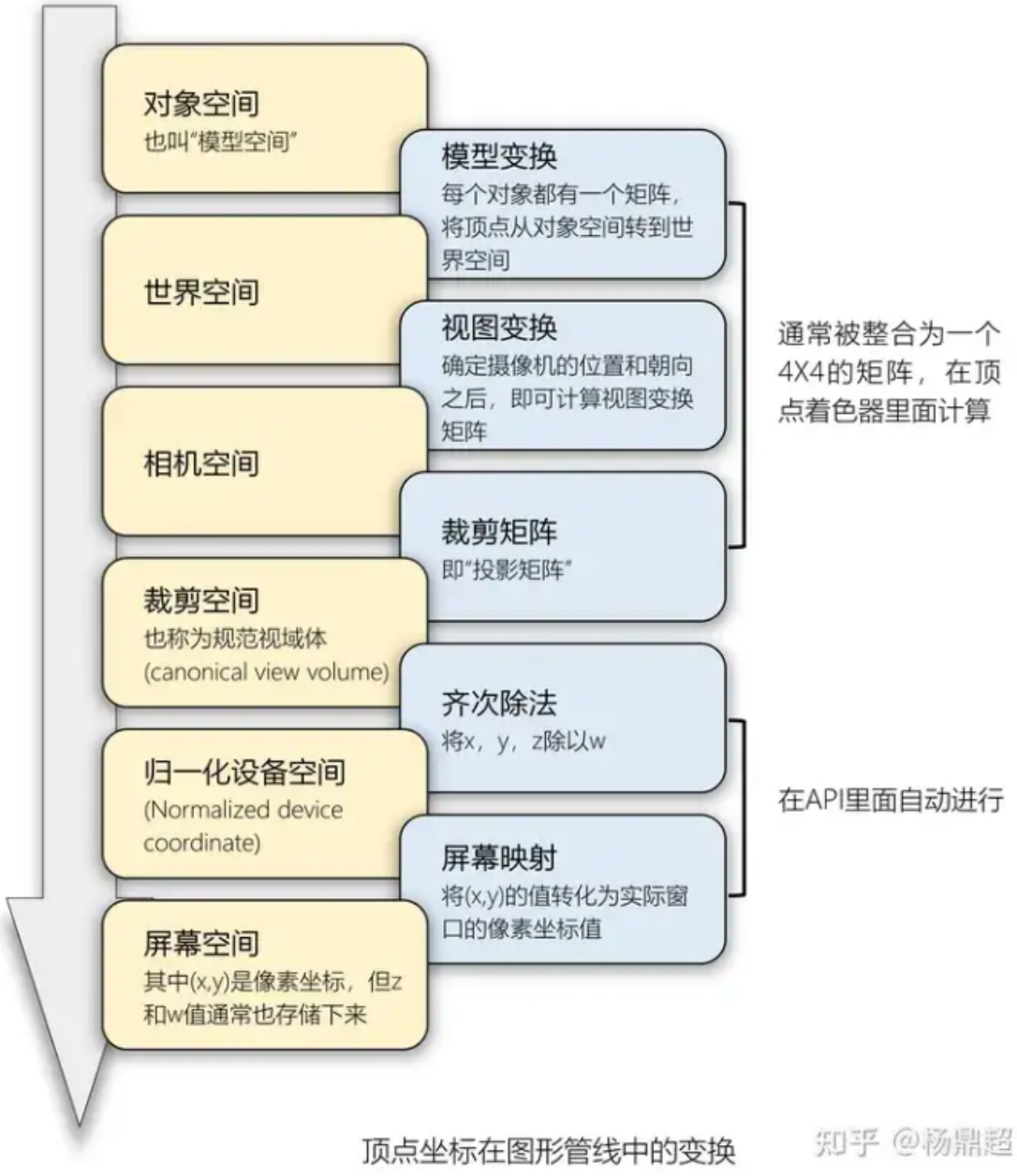
矩阵变换

平移、旋转、缩放、齐次坐标

- 模型变换： 将所有需要渲染到屏幕上的模型放在一个场景内。所有模型叠加的一个坐标空间，每个模型在世界空间都有一个位置放置他们。**物体的坐标会从局部变换到世界空间，由模型矩阵（Model Matrix）实现。**

- 视图变换：确定相机位置，然后将原本世界空间坐标下的物体转换到视图空间，这样方便我们下一步的投影操作以及后续的操作
- 投影变换：将所有的坐标从三维空间映射到屏幕上，这就是裁剪空间。**（由投影矩阵完成）**裁剪坐标会被处理至-1.0到1.0的范围内，并判断哪些顶点将会出现在屏幕上。

图形学的运行管线中，会经历**模型变换**



换（Model Transformation）到**视图变换（View Transformation）**最后再到**投影变换（Projection Transformation）**的过程。这里上节课讲述的仿射变换以及其他变换，都是针对世界坐标系下对于物体位置发生的改变，而在这一步结束之后，就会进行视图变换的操作。

模型变换是将所有需要渲染到屏幕上的模型放在一个场景内，那么视图变换就是确定相机位置，将原本世界空间坐标下的物体转换到视图空间，这样方便下一步的投影操作以及后续的操作。想象一下，一群人合照，大家在选择位置站好的过程就是**模型变换**，而摄像机选择角度和位置的过程就是**视图变换**，而最后按下快门选择拍照的过程，就称为**投影变换。**而这一整个流程在图形学管线中往往组合成为一个整体的变换矩阵，也就是**MVP矩阵。**

坐标

齐次坐标 原本是n维的向量表示如下；（X1, X2, X3...,Xn）的齐次坐标（hX1, hX2, hX3...,hXn，h）

h是一个实数，显然一个向量的齐次坐标表示是不唯一的。齐次坐标中的h取不同的值都表示同一个点，

[8, 4, 2], [4, 2, 1] 表示的都是二维点（4, 2）。齐次坐标"提供了用矩阵运算把二维、三维甚至更高维空间中的一个点集从一个坐标变换到另一个坐标系的有效方法；

光栅化

光栅化时一个三角形的上边界。如果三角形覆盖了像素的中心点，则该像素点被着色。如果三角形没有覆盖到该像素的中心点，该像素不会被着色。可以看出三角形在光栅化时，覆盖到的像素点是离散的、断断续续的，这样就形成了**锯齿/aliasing。**

深度测试

检测从某个方向看过去时，两个点A和B谁在谁的前面，以便知道谁挡住了谁，被挡住的点一般不会进行绘制，以达到和真实世界一样的遮挡效果。提供了深度测试的能力，开发者不用自己判断哪些被挡住然后不绘制，开启深度测试后，OpenGL会自动帮助我们完成。

原理就是：**在绘制一个像素时，将Z值（它到观察者的距离）分配给它，然后，当另外一个像素需要在屏幕上的同样位置进行绘制时，新像素的Z值将与已经存储的像素的Z值进行比较。如果新像素的Z值比较大，那么它距离观察者就更近，所以原来的像素就会被新的像素覆盖**

深度冲突

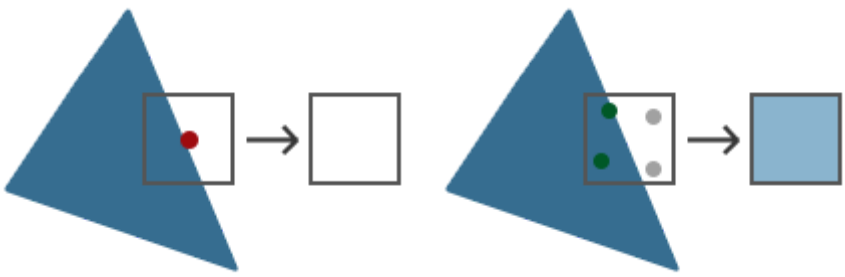
两个物体平行排列的时候，深度缓冲没有足够的精度来决定两个物体怎么绘制，就会不断切换两个形状的顺序。

- **不要把多个物体摆得太靠近，以至于它们的一些三角形会重叠**
- **尽可能将近平面设置远一些**
- **使用更高精度的深度缓冲**

抗锯齿

产生锯齿的原因是因为光栅化阶段，采样的方法是判断像素中心点是否在三角形内部，这就导致了某些边缘不在三角形内部的像素无法被片元着色器接收到。

可以多点去采样，分别在四个角增加采样点，但是这种会降低性能。



光照

风氏光照模型（Phong Lighting Model）由三个分量组成

- 环境（Ambient）即使在黑暗的情况下，世界上通常也仍然有一些光亮（月亮、远处的光），所以物体几乎永远不会是完全黑暗的。
- 漫反射（Diffuse）模拟光源对物体的方向性影响，视觉上最显著的分量
- 镜面（Specular）模拟有光泽物体上面出现的亮点。镜面光照的颜色相比于物体的颜色会更倾向于光的颜色。

```
void main() {
    float ambientStrength = 0.1;
    vec3 ambient = ambientStrength * lightColor;

    vec3 result = ambient * objectColor;
    FragColor = vec4(result, 1.0);
}
```

漫反射：通过计算光源与法向量的夹角。点乘dot可以很方便的计算出夹角。**两个单位向量的夹角越小，它们点乘的结果越倾向于1。当两个向量的夹角为90度的时候，点乘会变为0** 。点乘返回一个标量，我们可以用它计算光线对片段颜色的影响。不同片段朝向光源的方向的不同，这些片段被照亮的情况也不同。（法向量是一个垂直于顶点表面的（单位）向量。）

所以计算漫反射光照需要 1、法向量。2、光线的方向。

所有的光照都是在片元着色器里进行的。需要将法向量由顶点着色器传递到片元着色器。

```
// 顶点着色器
out vec3 Normal;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    Normal = aNormal;
}

// 片元
in vec3 Normal;
float diff = max(dot(norm, lightDir), 0.0);
vec3 diffuse = diff * lightColor;

// 环境分量和漫反射分量
vec3 result = (ambient + diffuse) * objectColor;
FragColor = vec4(result, 1.0);
```

纹理

纹理是一个2D图片，用来添加物体的细节，减少使用顶点来绘制模型，减少性能开销。

纹理坐标在x y 轴上。范围在0到1之间。如果超过了范围就会默认重复这个纹理图像。

环境光贴图、凹凸贴图、位移贴图

法线贴图就是用贴图扰动下反射光，给观察者感觉有凹凸感。当然必须有光照。

光线追踪 Ray-Tracing

光栅化的着色是一种局部的现象，在其着色的过程中只会考虑着色点自己的信息，而不会考虑其他物体，甚至不会考虑物资自身的其他部分对着色点的影响。事实上这些都是会有遮挡的关系的，是会产生阴影的，为了解决这个问题，就有了光线追踪

光栅化很快，但是质量不高。很难处理软阴影、反射、环境光照。光线追踪速度慢 但是渲染效果准确。

首先定义光线——沿直线传播，不会发生碰撞，从光源到人眼由光路的可逆性，在光线追踪的具体应用中，采用从人眼（认为是一个针孔摄像机）到光源的方法。

光线投射：人眼，成像平面，光源，物体

材质

不同的物体反射出来的光照强度是不同的，我们必须针对每种表面定义不同的材质属性。材质就像是物体的皮肤，它会决定物体看起来像什么样子，是木板、金属、是否透明、颜色等等。

```
// 定义材质
#version 330 core
struct Material {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};

uniform Material material;

void main()
{
    // 环境光
    vec3 ambient = lightColor * material.ambient;

    // 漫反射
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = lightColor * (diff * material.diffuse);

    // 镜面光
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
```

```
    vec3 specular = lightColor * (spec * material.specular);

    vec3 result = ambient + diffuse + specular;
    FragColor = vec4(result, 1.0);
}
```

材质 PBR

基于现实世界的物理原理的基本理论构成的渲染技术。是更加真实的。**基于物理参数为依据来编写表面材质**

是否PBR光照模型是否基于物理的判断条件：

- 基于微平面的表面模型
- 能量守恒
- 应用基于物理的BRDF