# Unity Shader入门精要——学习笔记

## 1.渲染流水线（概念，是一个渲染流程的基本划分）

1.1渲染流程分成3个概念性阶段，每个阶段本身通常也是一个流水线系统：

### a.应用阶段(Application Stage)。

通常由CPU负责实现，即开发者对应用阶段具有绝对的控制权。此阶段最重要的输出是渲染所需的几何信息，即渲染图元(rendering primitives)，通俗来讲渲染图元可以是点、线、三角面等。

此阶段中，开发者有3个主要任务：

首先准备好场景数据，例如摄像机的位置、视锥体、模型、光源等；其次，为了提高渲染性能，要做粗粒度剔除(culling)工作，以把那些不可见的物体剔除，不移交到几何阶段；最后，设置模型的渲染状态，这些渲染状态包括但不限于它使用的材质(漫反射颜色、高光反射颜色)、使用的纹理、使用的Shader等。

### b.几何阶段(Geometry Stage)。

通常由GPU负责实现，用于处理所有要绘制的几何相关的事情。几何阶段负责和每个渲染图元打交道，进行逐顶点、逐多边形的操作。几何阶段的一个重要任务就是把顶点坐标换到屏幕空间中，再交给光栅器进行处理。通过对渲染图元进行多步处理后，将会输出屏幕空间的二维顶点坐标、每个顶点对应的深度值、着色等相关信息，并传递给光栅化阶段。

### c.光栅化阶段(Rasterizer Stage)。

通常由GPU负责实现，光栅化的任务主要是决定每个渲染图元中的那些像素应该如何的被绘制到屏幕上。它需要对几何阶段中得到的逐顶点数据(例如纹理坐标、顶点颜色)进行插值，然后再进行逐像素处理。

## 2.CPU与GPU之间的通信

渲染流水线的起点是CPU，即应用阶段，由应用阶段通过DrawCall来命令GPU进行渲染，可分为3个阶段：

### a.把数据加载到显存中。

渲染所需的数据都需要从硬盘(HDD)中加载到系统内存(RAM)中。然后网格、顶点位置、法线方向、顶点颜色、纹理坐标等数据信息又被加载到显存中(VRAM)中。

### b.设置渲染状态。

定义如何渲染，例如，使用哪个顶点着色器(Vertex Shader)、片元着色器(Fragment Shader)、光源属性、材质等。

### c.调用DrawCall命令。

DrawCall是一个命令，它的发起方是CPU，接收方是GPU。这命令仅仅会指向一个需要被渲染的图元列表，而不会再包含任何材质信息

### 3.GPU流水线

3.1对于几何阶段、光栅化阶段，开发者无法拥有绝对的控制权，但每个阶段的GPU向开发者提供了不同的可配置性或可编程性。

### 

### 几何阶段——顶点着色器(Vertex Shader)：

是完全可编程的，它通常用于实现顶点的空间变换、顶点着色等功能。

### 几何阶段——曲面细分着色器(Tessellation Shader)：

是一个可选的着色器，它用于细分图元。

### 几何阶段——几何着色器(Geometry Shader)：

是一个可选的着色器，它可被用于执行逐图元的着色操作，或者被用于产生更多的图元。

### 几何阶段——裁剪(Clipping):

将不在摄像机视野内的顶点裁剪掉，并剔除某些三角图元的面片。此阶段是可配置的。例如可以使用自定义的裁剪平面来配置裁剪区域，也可以通过指令控制裁剪三角图元的正面还是背面。

### 几何阶段——屏幕映射(Screen Mapping)：

此阶段不可配置和编程，负责把每个图元的坐标转换到屏幕坐标系中。

### 光栅化阶段——三角形设置(Triangle Setup)：

固定函数(Fixed-Function)的阶段。

### 光栅化阶段——三角形遍历(Triangle Traversal)：

固定函数(Fixed-Function)的阶段。

### 光栅化阶段——片元着色器(Fragment Shader)：

是完全可编程的，它用于实现逐片元的着色操作。

### 光栅化阶段——逐片元操作(Per-Fragment Operations)：

此阶段不可编程，但具有很高的可配置性。负责执行修改颜色、深度缓冲、混合等。

### 3.2顶点着色器(Vertex Shader)

顶点着色器的处理单位是顶点，即输入进来的每个顶点都会调用一次顶点着色器。顶点着色器本身不会创建或销毁任何一个顶点，而且无法得知顶点与顶点之间的关系。因上GPU可以并行处理每个顶点的计算，这意味着此阶段的处理速度会很快。

顶点着色器的主要工作有：坐标变换、逐顶点光照(计算顶点颜色)。

坐标变换：顶点着色器可以在这一步中改变顶点的位置。例如，我们可以通过改变顶点位置来模拟水面、布料等。其最终必须完成的一个工作是，把顶点坐标从模型空间转换到齐次裁剪空间。

### 3.3裁剪(Clipping)

一个图元和摄像机视野的关系有3种：

完全在视野内：不裁剪，直接传递给下一个流水阶段。

部分在视野内：进行裁剪后，再传递给上一流水阶段。

完全在视野外：不裁剪，不传递给下一个流水阶段。

### 3.4屏幕映射(Screen Mapping)：三维转二维

### 

负责把每个图元的x和y坐标转换到屏幕坐标系(Screen Coordinates)下。

屏幕映射不会对输入的z坐标做任何处理。实际上，屏幕坐标系和z坐标一起构成了一个坐标系，叫做窗口坐标系(Window Coordinates)。

屏幕映射得到的屏幕坐标决定了这个顶点对应屏幕上哪个像素以及距离这个像素有多远。需要注意，OpenGL的原点在左下角，DirectX的原点在左上角。



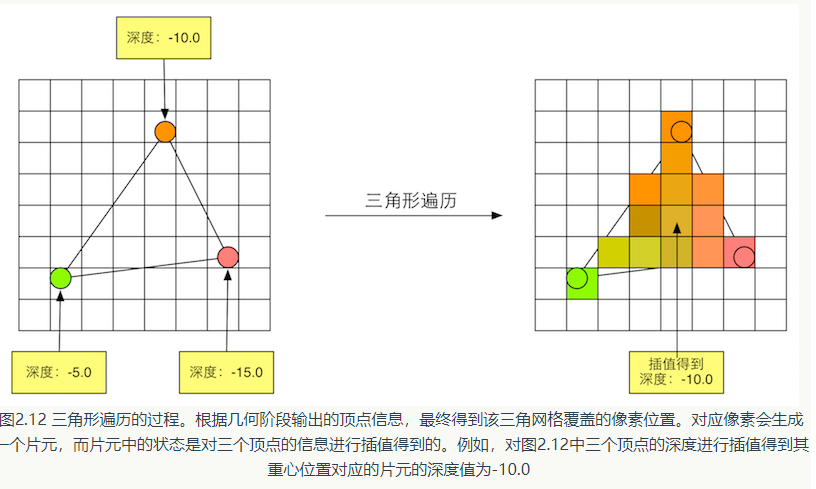
### 3.5三角形设置(Triangle Setup)

从这一步开始就进入了光栅化阶段。从上一个阶段输出的信息是屏幕坐标系下的顶点位置以及它们相关的额外信息，如深度值(z坐标)、法线方向、视角方向等。光栅化阶段有两个重要的目标：计算每个图元覆盖了哪些像素，以及为这些像素计算它们的颜色。

三角形设置：计算光栅化一个三角网格所需的信息，具体来说，上一个阶段输出的都是三角网格的顶点，即我们得到的是三角网格每条边的两个端点。但如果要得到整个三角网格对像素的覆盖情况，我们就必须计算每条边上的像素坐标。

### 3.6三角形遍历(Triangle Traversal)

三角形遍历阶段将会检查每个像素是否被一个三角网格所覆盖。如果被覆盖的话，就会生成一个片元(fragment)。这样一个找到哪些像素被三角网格覆盖的过程就叫三角形遍历，也叫扫描变换(Scan Conversion)。

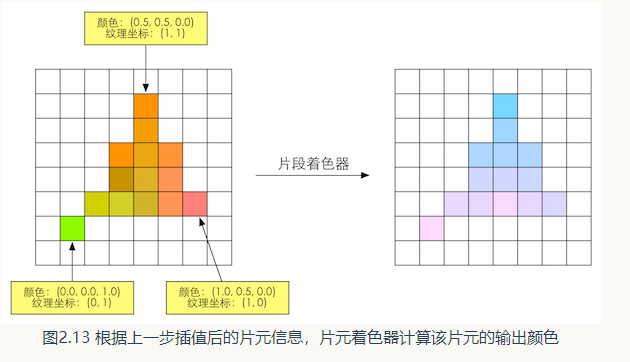
三角形遍历阶段根据上一个阶段的计算结果来判断一个三角网格覆盖了哪些像素，并使用三角网格3个顶点信息对整个覆盖区域的像素进行插值。最终输出的是一个片元序列。

### 3.7片元着色器(Fragment Shader)

片元着色器是可编程的，此阶段不会影响屏幕上每个像素的颜色值，只会生成片元数据。

片元：是用来表述一个三角网格是怎样覆盖每个像素的一系列数据。

片元着色器的局限在于，它仅可以影响单个片元，而不会将自己的任何结果直接发送给它的邻居们。



### 3.8逐片元操作

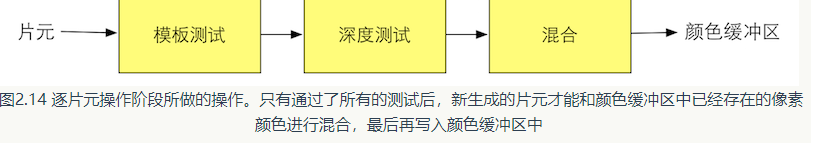
在OpenGL叫逐片元操作，在DirectX叫输出合并阶段。

此阶段是高度可配置性的。

此阶段的几个主要任务：

a.决定每个片元的可见性。这涉及了很多测试工作，如模板测试、深度测试等。

b.如果一个片元通过了所有测试，就需要把这个片元的颜色值和已经储存在颜色缓冲区中的颜色进行混合。



模板测试(Stencil Test)通常用于限制渲染的区域、渲染阴影、轮廓渲染等。

深度测试(Depth Test)

混合(Blend)

为什么要混合？因为渲染过程是一个物体接着一个物体画到屏幕上，而每个像素的颜色信息被存储在一个名为颜色缓冲的地方。因上，当我们执行这次渲染时，颜色缓冲中往往已经有了上次渲染之后的颜色结果，那么，我们是使用这次渲染得到的颜色完全覆盖之前的结果，还是进行其他处理，这就是混合需要解决的问题。

对于不透明物体，开发者可以关闭混合操作。

当模型的图元经过上述层层计算和测试后，就会显示到我们的屏幕上，我们的屏幕显示的就是颜色缓冲区中的颜色值。但是，为了避免我们看到那正在进行光栅化的图元，GPU会使用双重缓冲(Double Buffering)策略。这意味着，对场景的渲染是在幕后发生的，即在后置缓冲(Back Buffer)中，一旦场景已经被渲染到了后置缓冲中，GPU就会交换后置缓冲区和前置缓冲(Front Buffer)中的内容，而前置缓冲区是之前显示在屏幕上的图像。由此保证了我们看到的图像总是连续的。

### 什么是OpenGL/DirectX

OpenGL/DirectX就是这些图像应用编程接口，架起了上层应用程序和底层GPU的沟通桥梁。

### 

### 什么是HLSL、 GLSL、 CG

这些是高级着色器语言：

HLSL：DirectX的(High Level Shading Language)。

GLSL：OpenGL的(OpenGL Shading Language)。

CG：NVIDIA的(C for Graphic)

注意Unity可以选择使用以上三种高级着色器语言，但并不是真正意义上的对应的着色语言，尽管它们的语法几乎一样。

### 什么是Draw Call

DC就是CPU调用图像编程接口。

CPU与GPU是如何实现并行工作的？

要使两者并行工作，就需要一个命令缓冲区(Command Buffer)。命令缓冲区包含了一个命令队列，由CPU向队列末尾添加命令，而GPU从队列头中读取命令。

### 

### 为什么DC多了会影响帧率。

从感性的角度上你可以这样理解：

操作拷贝10000个，每个大小为1KB到另一文件夹，比拷贝一个大小为10MB的文件到另一文件夹，所需的时间要长得多，虽然他们的文件大小是一样的。

GPU的渲染能力很强，渲染200个和渲染2000个三角网格通常没什么区别。

### 如何减少DC？

在开发过程中，为减少DC有两点需要注意:

a.避免使用大量很小的网格，当不可避免时，考虑是否可以合并它们。

b.避免使用过多的材质，尽量在不同的网格之间共用同一个材质。

批处理(Batching):把很多小的DrawCall合并成一个大的DrawCall。

需要注意的是，由于我们需要在CPU的内存中合并网格，而合并的过程是需要消耗时间的。因此，批处理更加适合那些静态的物体，如不会移动的大地，石头等，对于这些静态物体我们只需要合并一次即可。当然我们也可以对动态物体进行批处理，但是，由于这些物体是不断运动的，因此每一帧都需要重新进行合并然后再发送给GPU，这对空间和时间都会造成一定的影响。

### 

### 4.什么是固定管线渲染

通常是指在较旧的GPU上实现的渲染流水线。固定管线只给开发者提供一些配置操作，但开发者没有对流水线阶段的完全控制权。一个形象的比喻是，我们在使用固定管线进行渲染时，就好像在控制电路上的多个开关，我们只能打开或关闭一个开，但永远无法控制整个电路的排布。

OpenGL3.2后不再支持。

### ----------------华丽的分隔线----------------------

### Unity Shader基础

UnityShader提供了一个地方给开发者更加轻松地管理着色器代码以及渲染设置(如开启/关闭混合、深度测试、设置渲染顺序等)。学习和编写着色器的过程一直是一个学习曲线很陡峭的过程。通常情况下，为了自定义渲染效果往往需要和很多文件和设置的交道，这些过程很容易消磨掉初学者的耐心。而且一些细节问题也往往需要开发者花费较多的时间去解决。Unity为了解决上述问题，为我们提供了一层抽象——Unity Shader。而我们和这层抽象打交道的途径就是使用Unity提供的一种专门为UnityShader服务的语言——ShaderLab。

在Unity5.2往后的版本中，提供了4种UnityShader模板供我们选择：

a.Standard Surface Shader:产生一个包含了标准光照模型的表面着色器模板。

b.UnlitShader:产生一个不包含光照(但包含雾效)的基本的顶点/片元着色器模板。

c.Image Effect Shader:为我们实现各种屏幕后处理效果的基本模板。

d.ComputeShader:产生一种特殊的Shader文件，这类Shader旨在利用GPU的并行性进行一些与常规渲染流水线无关的计算。

#### 什么是ShaderLab?

是编写Unity Shader的一种说明性语言。

ShaderLab的结构：

Shader:关健字，定义Shader的名称。加’/’可指定位置。

Properties:关健字，可以定义属性，并显示在材质面板中。

注意：即使我们不在Properties语义中声明这些属性，也可以直接在CG代码中定义变量，此时，可以通过脚本向Shader中传递这些属性。因此Properties语义块的作用仅仅是为了让这些属性可以出现在材质面板中。

8种属性类型：

Int：数字类型，默认值为单独的数字。

Float:数字类型，默认值为单独的数字。

Range(min, max):数字类型，默认值为单独的数字。

Color:默认值是用圆括号包围的一个四维向量。如:(1,1,1,1)。

Vector:默认值是用圆括号包围的一个四维向量。如:(1,1,1,1)。

2D、3D、Cube:默认值是一个字符串后跟一个花括号。如:””{}。其中字符串的值可以为空，或者是内置的纹理名称，如”white”, “black”,”gray”,”bump”。

SubShader:

每个Unity Shader中可以包含多个SubShader语义块，但最少要有一个。当Unity加载此Unity Shader时，会遍历选出第一个能够在目标平台上运行的SubShader。如果都不支持的话，就会使用Fallback语义指定的Unity Shader。

Unity提供多个Subshader的原因在于，不同的显卡具有不同的能力。

SubShader的成员：

[Tags]:标签设置

可选的。

是一个键值对。键和值都是字符串类型。

这些键值对是SubShader和渲染引擎之间的沟通桥梁。

支持的标签结构,注意这些标签公可以在Subshader中声明，如下：

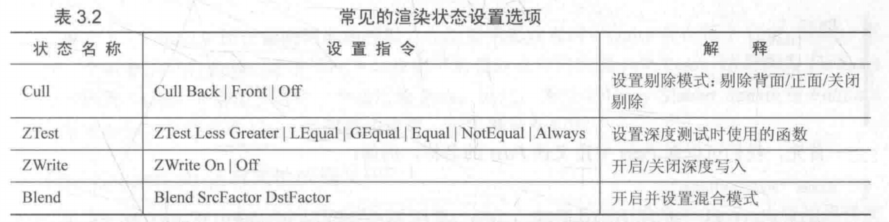


[RenderSetup]:渲染状态设置

可选的。

在SubShader块中设置的渲染状态，将会被应用到所有的Pass中。如果想单独对某个Pass起效，可在Pass语义块中单独进行渲染设置。

ShaderLab提供了一系列渲染状态的设置指令。如图：



Pass{ }：

可以有多个Pass，但过多的Pass往往会造成渲染性能的下降。

每个Pass定义了一次完整的渲染流程。

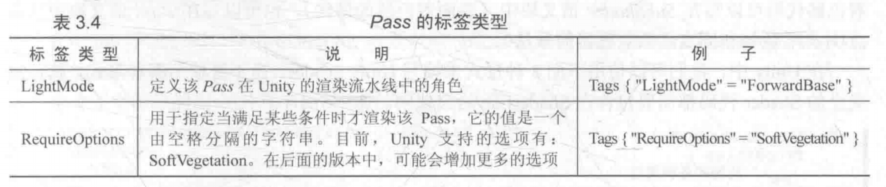
标签设置、渲染状态设置同样可以在Pass中声明。不同的是，Subshader中的一些标签设置是特定的，也就是说，这些标签设置和pass中使用的标签是不一样的。而对于渲染状态设置来说，其使用的语法是相同的。但是，如果我们在Subshader中进行了这些设置，那么将会用于所有的Pass中。

通过Name关健字来定义Pass的名称，通过名称，可以使用ShaderLab的UsePass命令来直接使用其他Unity Shader中的Pass。这样可以提高代码的复用性，需要注意，由于unity内部会把所有pass的名称转换成大写，因此，在使用UsePass命令时必须使用大写形式的名字。例如：

UsePass“MyShader/MYPASSNAME”。

可以对Pass进行渲染设置，SubShader的渲染状态设置同样适用于Pass，除了上面提到的渲染状态外，在Pass中还可以使用固定管线着色器的命令。

可以对Pass进行标签设置，但它的标签不同于SubShader的标签。如下：



特殊的Pass:

UsePass:可以使用该命令来复用其他Unity Shader中的的Pass。

Grabpass:负责抓取屏幕并将结果存储在一张纹理中，以用于后续的Pass处理。

Fallback:告诉Unity，如果以前的SubShader在这块显卡上都不能运行，那么你就用这个指定的吧。以下二选一：

Fallback “ShaderName“：指定某个内置的着色器。

Fallback Off：关掉，跑不了就算。

事实上，Fallback还会影响阴影的投射。在渲染阴影纹理时，Unity会在每个UnityShader中寻找一个阴影投射的Pass。通常情况下，我们不需要自己专门实现一个Pass，这是因为Fallback使用的内置Shader中包含了这样一个通用的Pass。因此，为每个Unity Shader正确设置Fallback是非常重要的。

Surface Shader：表面着色器。unity的宠儿。

Surface Shader是unity自己创造的一种着色器代码类型，是对顶点/片元着色器的更高一层的抽象。它存在的价值在于，unity为我们处理了很多光照细节。

SubfaceShader要求写在SubShader中的CGPROGRAM和ENDCG之间。不能写在Pass里面。

Vertex/Fragment Shader：顶点/片元着色器。最聪明的孩子。

在unity中可能使用CG/HLSL语言来编写顶点/片元着色器。

要求写在SubShader下的Pass中的CGPROGRAM和ENDCG之间。

Fixed Function Shader：固定管线着色器。被抛弃的角落。

要求写在SubShader下的Pass中。这些代码相当于Pass中的一些渲染设置。

使用ShaderLab的渲染设置来编写，而非使用CG/HLSL。

选择哪种UnityShader形式呢？

除非你的着色器要求运行在很老旧的GPU上，否则不要使用固定管线着色器。

如果你要和各种光源打交道，可以使用表面着色器，但需注意在移动平台上的性能表现。

如果你的光照数目较少，或者你有很多自定义的渲染效果，那么使用顶点/片元着色器是更好的选择。

## 总结与解惑

Unity Shader不是真正的Shader。实际上指的是一个ShaderLab文件。

他经真正的Shader所能做的事情多。

在Unity Shader中，可以在同一个文件里同时包含需要的顶点和片元着色器代码。

在Unity Shader中，可以通过一行特定指令就可以设置例如开启混合、深度测试等。

在Unity Shader中，只需要在特定语句块中声明一些属性，就可以依靠材质来方便地改变这些属性。而且对于模型自带的数据(如顶点位置、纹理坐标、法线等)，也提供了直接访问的方法，不需要开发者自行编码来传给着色器。

# 基础数学

## 二维笛卡尔坐标系：

包含了两个部份信息：

a.一个特殊的位置(原点)，它是整个坐标系的中心。

b.两条过原点并互相垂直的矢量即x,y轴。

二维坐标系中，x,y轴方向不是固定的，但总是可以通过旋转操作来使它们的坐标轴指向相同。所以说所有二维笛卡尔坐标系都是等价的。

三维笛卡尔坐标系中的坐标轴方向不是固定的。我们最多可以让其中两个坐标轴的指向重合，但第三个坐标轴的指向总是相反的。

在三维中就产生了两种不同种类的坐标系。

### a.左手坐标系：

手背向自己，三个手指所指方向为正向。

左手法则，旋转正方向是顺时针。

### b.右手坐标系:

手心向自己，三个手指所指方向为正向。

右手法则，旋转正方向是逆时针。

判断左手还是右手坐标系的方法：身体坐直，向右伸直右手，此时右手方向就是X轴的正向，头顶向上的方向就是y轴的正向，此时，如果你的正前方的方向是Z轴的正向，那么你本身所在的坐标系就是左手坐标系。如果你的正前方向是Z轴的负向，就是右手坐标系。

## 矢量(Vector)

矢量是指n维空间中一种包含了模(即长度)和方向的有向线段。

速度就是一种典型的矢量，距离就是一种标题。

一个矢量的长度(模)可以是任意的非负数。

矢量的方向则描述了这个矢量在空间中的指向。

### 点和矢量的区别。

点是一个没有大小之分的空间中的位置。

矢量是一个有模和方向但没有位置的量。

如果把矢量的尾固定在坐标系原点，那么这个矢量的表示就和点的表示重合了。

### 矢量和标量的运算。

矢量不能与标量进行加减运算。

矢量可以和标量进行乘法运算，结果是得到一个不同长度的新矢量，并且，矢量和标量的位置可以互换。

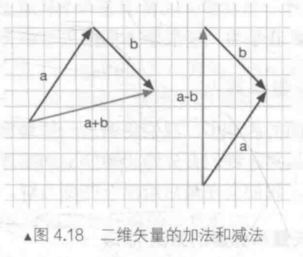
矢量可以和标量进行除法运算，结果是得到一个不同长度的新矢量，并且，只能是矢量被标量除，而不能是标量被矢量除。即((1，2，3) / 5)。

矢量和标量运算的几何意义是，意味着对矢量进行一个大小为标量的缩放。

### 矢量与矢量的运算。

矢量与矢量的加减法运算，其结果是一个相同维度的新矢量。

一个矢量不可以和一个标量相加减，或者是不同维度的矢量相加减运算。

矢量加法的几何意义，利用矢量的加减法来计算一个点相对于另一点的位移。****

### 矢量的模

矢量的模(空间中的长度)是一个标量,表示为|v|。

计算公式：矢量各元素的平方和，再开根。

### 。

### 矢量的归一化(单位矢量)

单位矢量指那些模为1的矢量。

对任何给定的非零矢量，把它转换成单位矢量的过程就被称为归一化(normalization)。

单位矢量表示为,矢量头上添加一个戴帽箱号。

****

### 矢量的点乘(内积，点积)

点积公式：a.b。

****。

点乘的结果是一个标量。

点乘的结果，具有如下意义：

a.在a，b非零的前提下，如果点乘后值为负，则a，b形成的角大于90度，如果为零，那么a，b垂直，如果为值，那么a，b形成的角为锐角。

b．两个单位向量的点乘得到两个向量的夹角的cos值，通过它可以知道两个向量的相似性，利用点乘可判断一个多边形是否面向摄像机还是背向摄像机。

C．向量的点乘与它们夹角的余弦成正比，因此在聚光灯的效果计算中，可以根据点乘来得到光照效果，如果点积越大，说明夹角越小，则物体离光照的轴线越近，光照越强。

矢量的点积满足交换律，即a.b=b.a。

点积的几何意义就是投影。

点积的符号可以让我们知道两个矢量的方向关系。

特性一：点积可结合标量乘法，对点积其中一个矢量进行缩放的结果，相当于对最后的点积结果进行缩放。

****

特性二：点积可以结合矢量加减法。

****

特性三：一个矢量和自身进行点积的结果，是该矢量的模的平方。

****。

点积的几何表达式：a.b=|a||b|cos⊙。

****。

****

### 矢量的叉积(外积，叉乘)

点积的公式：

### 。

叉积的结果是得到一个同时垂直到这两个矢量的新矢量。

叉积满足反交换律，即a×b等于-(b×a)。

叉积不满足交换律，即a×b不等于b×a。

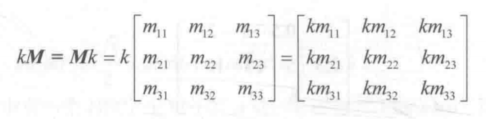
叉积不满足结合律，即(a×b)×c不等于a×(b×c)。

## 矩阵

矩阵有行矩阵和列矩阵之分。

### 矩阵可以和标量相乘。

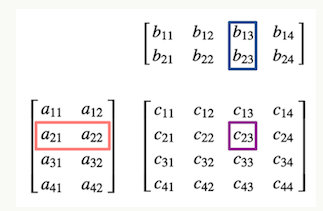
它的结果仍然是一个相同维度的矩阵。它们之间的乘法，就是矩阵的每个元素和该标量相乘。



### 矩阵和矩阵的乘法。

矩阵与矩阵相乘，必须满足:第一个矩阵的列数必须与第二个矩阵的行数相同。

它的结果是一个新的矩阵，前且新的矩阵的维度是由，第一个矩阵的行和第二个矩阵的列组成。

C23=a21\*b13 + a22\*b23。

矩阵乘法的性质：

a.矩阵乘法不满足交换律，即AB != BA。

b.矩阵乘法满足结合律，即ABCDE=(A(BC)D)E=(AB)(CD)E。

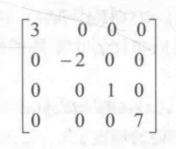
### 特殊矩阵

#### 方块矩阵：

简称方阵，是指那些行和列数目相等的矩阵。

方阵的特殊性质：**对角元素**，指的是行号和列号相同的元素，如m11,m22,m33,m44。

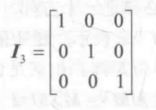
方阵的特殊性质：**对角矩阵**，指除了对角元素外的所有元素都为0，就叫对角矩阵。

对角矩阵，m11=3，m22=-2，m33=，,m44=7。

#### 单位矩阵：

如果对角矩阵的所有对角元素，都为1时，它是一个单位矩阵。

**单位矩阵的特性**：任何矩阵和单位矩阵相乘，结果还是原来的矩阵。

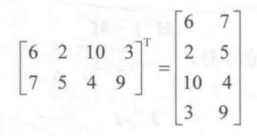
单位矩阵，对角元素，m11=1，m22=1，m33=1。

#### 转置矩阵：

实际是对原矩阵的一种运算，将原矩阵的第i行变成第i列，而第j列变成第j行。即r\*c转置后为c\*r。

**转置矩阵的特性**：矩阵转置的转置等于原矩阵。

**转置矩阵的特性**：矩阵串接的转置，等于反向串接各个矩阵的转置。

转置公式：转置例子：

转置的转置等于原矩阵：

串接转置：

#### 逆矩阵:

不是所有的矩阵都有逆矩阵，第一个前提就是，该矩阵必须是一个方阵。

如果一个矩阵有对应的逆矩阵，我们就说这个矩阵是可逆的，相反，就是不可逆的。

如何判断一个矩阵是否可逆？简单来说，如果一个矩阵的行列式不为0，它就是可逆的。

常使用逆矩阵来求解反向的变换。

**逆矩阵的性质一**：逆矩阵的逆矩阵是原矩阵本身。

**逆矩阵的性质二**：单位矩阵的逆矩阵是它本身。

**逆矩阵的性质三**：转置矩阵的逆矩阵是逆矩阵的转置。

**逆矩阵的性质四**：矩阵串接相乘后的逆矩阵等于反向串接各个矩阵的逆矩阵。

#### 正交矩阵:

正交矩阵：如果一个方阵M和它的转置矩阵的乘积是单位矩阵的话，那么它就是正交的，反过来也成立。

如果一个矩阵是正交的，那么它的转置矩阵和逆矩阵是一样的。

根据一个矩阵的结构过程判断是否为正交矩阵的三个条件：

a.矩阵的每一行，都是单位矢量，只有这样它们与自己的点积才能是1。

b.矩阵的每一行，它们之间互相垂直，只有这样它们之间的点积才能是0。

c.上述两条结论对矩阵的每一列同样适用，因为M是正交矩阵，那么M的转置矩阵也会是正交矩阵。

### 矩阵的几何意义

矩阵可视化的结果就是变换，变换一般包含了旋转、缩放、平移。

#### 线性变换：

线性变换是指那些可以保留矢量加和标题乘的变换。

线性变换公式:f(x) +f(y) = f(x+y) 和kf(x) = f(kx)。

缩放和旋转都是一种线性变换。

## 第六章 Unity中的基础光照

### 6.1 我们是如何看到这个世界的

当我们描述“这个物体是红色的”时，实际上是因为这个物体会反射更多的红光波长，而吸收了其它波长。

要模拟真实的光照环境来生成一张图像，需要考虑3种物理现象：

a.首先，光线从光源(light source)中被发射出来。

b.然后，光线与场景中的一些物体相交；一些光线被物体吸收，而另一些光线被散射到其他方向。

c.最后，摄像机吸收了一些光，产生了一张图像。

#### 光源

在实时渲染中，通常把光源当成一个没有体积的点，用小写L来表示它的方向。

在光学里，使用辐照度(irradiance)来量化光。

辐照度和cos(角度)成正比，因此使用点积来计算就可得到辐照度。

根据入射光线的数量和方向，可以计算出出射光线的数量和方向。通常用出射度(exitance)来描述它。辐照度和出射度之间满足线性关系，而它们之间的比值就是材质的漫反射和高光反射属性。

#### 吸收和散射

光线由光源发射出来后，就会与一些物体相交，通常相交有两种结果：

##### 散射(scattering)：

散射改变光线的方向，不改变光线的密度和颜色。

光线在物体表面经过散射后，有两种方向：

a.一种散射到物体内部，这种现象叫折射(refraction)或透射(transmission)。

b.另一种散射到外部，这种现象叫反射(reflection)。从物体表面重新发射(发射)出的光线将具有和入射光线不同的方向分布和颜色。

为了区分这两种不同的散射方向，在光照模型中使用了不同的部份来度算它们：

a.高光反射(specular)部分表示物体表面是如何反射光线的。

b.漫反射(diffuse)部分表示有多少光线会被折射、吸收和散射出表面。

##### 吸收(absorption)：

吸收改变光线的密度和颜色，不改变光线的方向。

#### 着色(shading):

着色指的是，根据材质属性(如漫反射属性等)、光源信息(如光源方向、辐照度等)，使用一个等式去计算沿某个观察方向的出射度过程。我们也把这个等式称为光照模型。

##### BRDF光照模型：

BRDF大多使用一个数学公式来表示，并且提供了一些参数来调整材质属性。

当给定入射光线的方向和辐照度后，BRDF可以给出某个出射方向上的光照能量分布。

### 6.2标准光照模型

标准光照模型只关心直接光照(direct light)，也就是那些直接从光源发射出来照射到物体表面后，经过物体表面的一次反射直接进入摄像机的光线。

它的基本方法是，把进入到摄像机内的光线分为4个部分，每个部分使用一种方法来计算它的贡献度：

a.**自发光(emissive)部分**，用于描述当给定一个方向时，一个表面本身会向该方向发射多少辐射量。需要注意的是，如果没有使用全局光照(global illumination)技术，这些自发光的表面并不会直的照亮周围的物体，而是它本身看起来更亮了而已。

b.**高光反射(specular)部分**，用于描述当光线从光源照射到模型表面时，该表面会在完全镜面反射方向散射多少辐射量。

计算高光反射需要知道：表面法线n、视角方向v、光源方向i、反射方向r。最少要知道三个。知道n, v, I 求r ：



即：表面法线单位向量n和光源单位向量i求得。CG函数：reflect(i入射方向, n法线方向);

利用Phong模型来计算高光反射：



即：(入射光线的颜色和强度 \* 材质的高光反射系数) \* max(0, 视角方向单位向量 \* 反射方向)。注意：视角向量与反射方向向量要保证在同一空间下，并且保证点积后不为负。

c.**漫反射(diffuse)部分**，用于描述当光线从光源射到模型表面时，该表面会向每个方向散射多少辐射量。

漫反射光照符合兰伯特定律：反射光线的强度与表面法线和光源方向之间夹角的余弦值成正比。漫反射的计算公式:



即：光源的颜色 \* 材质漫反射的颜色 \* 法线矢量与光源的单位矢量的点积。

注意法线矢量与光源矢量需要保证在同一空间下，并且要保证点积后的值不为负。

d.**环境光(ambient)部分**，用于,近似模拟其它所有的间接光照。

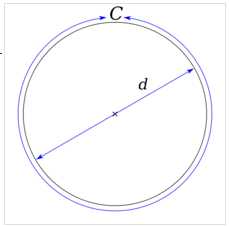
# 数学篇

## Mathf 数学运算

### Mathf.PI :

圆周率(∏),是圆的周长与直径的比值。约等于3.14159265358979323846....。

∏ = C / d = C / 2r 。



### Mathf.Deg2Rad:

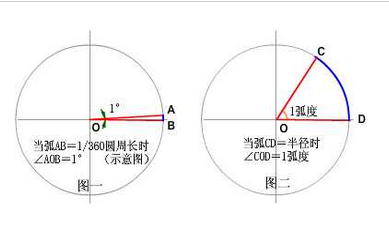
度(角度)转弧度的常量 即： (PI\*2)/360 或PI / 180 = 0.31830988618379067153776752674503。

用法：角度 \* Mathf.Deg2Rad = 弧度。

度(角度)与弧度是度量角大小的两种不同的单位。

角度：两条射线从圆心向圆周射出，形成一个夹角和夹角正对的一段弧，当这段弧长正如等到圆周长的360分之一时，这个夹角的大小为1度。图一。

弧度：两条射线从圆心向圆周射出,，形成一个夹角和夹角正对的一段弧，当这段弧长正好等于圆的半径时，这个夹角大小为1弧度。图二。



### Mathf.Rad2Deg:

弧度转角度的常量 即：360/(PI\*2) 或 180/PI = 57.295779513082320876798154814105。

一个圆等于2∏，即360(度) =2\*PI(弧度)，1弧度约等57.29577951。

2 \* PI \* 57.29577951 = 360度。

用法：弧度 \* Mathf.Rad2Deg = 角度。

### Mathf.Epsilon:

最小的浮点数，不等于0。

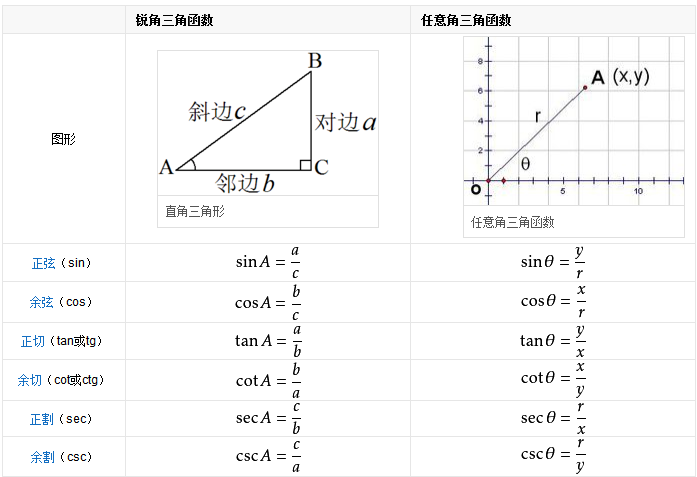
### Mathf.Infinity:

正无穷大的数。

### Mathf.NegativeInfinity:

负无穷小的数。

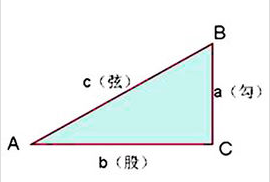
## 三角函数



**勾股定理定义：**

在一个直角三角形中，两个直角边a、b边长的平方之和等于斜边c边长的平方。

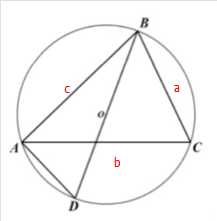
即：a \* a + b \* b = c \* c。如图：



**正弦定理定义：**

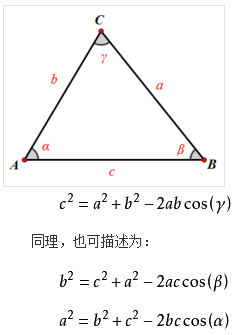
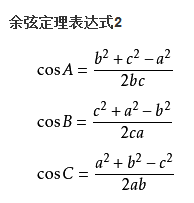
在任意三角形中，角A、B、C所对的边长分别为a、b、c，三角形外接圆的半径为R，直径为D则有：a/sinA = b/sinB = c/sinC = 2R = D。

即:一个任意三角形中，各边和所对角的正弦之比相等，且该比值等于该三角形外接圆的直径长度。如图：



**余弦定理定义：**

在任意三角形中，任何一边的平方等于其他两边平方的和 减去 这两边与它们夹角的余弦的两倍积。如图：

**表达式1:2:**

**正切定理定义：**

在平面三角形中，任意两条边的和 除以 第一条边 减 第二条边的差所得的商，等于 这两条边对角的和的一半的正切 除以 第一条边对角 减 第二条边对角的差的一半的正切，所得的商。

### Mathf.Sin(float f): float

计算并返回以弧度为单位f的正弦值。f = 弧度。结果是对边/斜边的比值。

正弦值：直角三角形中，对边/斜边的值。任意锐角的正弦值等于它的余角的余弦值，任意锐角的余弦值等于它的余角的正弦值。

### Mathf.Cos(float f): float

计算并返回以弧度为单位f的余弦值(介于-1~1之间的值)。f = 弧度。结果是邻边/斜边的比值。

余弦值：直角三角形中，邻边/斜边的值。

### Mathf.Tan(float f):float

计算并返回以弧度为单位f的正切值。f = 弧度。结果是对边/邻边的比值。

正切值：直角三角形中，对边/邻边的值。

### Mathf.Asin(float f) : float 反正弦

计算并返回正弦值f的弧度。f = 正弦值。结果是弧度角，可转为角度。

反正弦值：

用例：Mathf.Asin( Mathf.Sin(30 \* Mathf.Deg2Rad) ) \* Mathf.Rad2Deg = 30角度;

### Mathf.Acos(float f) : float 反余弦

计算并返回余弦值f的弧度。f = 余弦值。结果是弧度角，可转为角度。

用例：Mathf.Acos(Mathf.Cos(30 \* Mathf.Deg2Rad)) \* Mathf.Rad2Deg = 30角度。

### Mathf.Atan(float f) : float 反正切

计算并返回正切值f的弧度值(介于负二分之PI与正二分之PI之间)。f = 正切值。结果是弧度角。

用例：Mathf.Atan(Mathf.Tan(30 \* Mathf.Deg2Rad)) \* Mathf.Rad2Deg = 30角度。

### Mathf.Atan2(float y, float x) : float 反正切

同上,y是对边长，x是邻边长。返回值为x轴和一个零点超始在(x,y)结束的2D向量之间的夹角。

### Mathf.Sqrt(float f) : float 平方根(开平方)

计算并返回值f的平方根。即开平方。

用例：Mathf.Sqrt(25) = 5。

### Mathf.Pow(float f, float p) : float 次方

计算并返回f的p次方。

用例：Mathf.Pow(5, 2) = 25。

### Mathf.IsPowerOfTwo(int v):bool 是否是2的幂

如果该值是2的幂，返回true。

用例：Mathf.IsPowerOfTwo(7)=false；Mathf.IsPowerOfTwo(32)=true；

### Mathf.NextPowerOfTwo(int v):int 下个2的幂

返回大于v值的2的幂的值。

用例：Mathf.NextPowerOfTwo(7)=8；Mathf.NextPowerOfTwo(139)=256；

### Mathf.ClosesPowerOfTwo(int v):int 最近的二次方

返回最近v值的2的次方数。

用例：Mathf.ClosesPowerOfTwo(7)=8；Mathf.ClosesPowerOfTwo(19)=16；

### Mathf.Exp(float p) : 指数

返回自然对数e的次方的值。

指数是幂运算aⁿ(a≠0)中的一个参数，a为底数，n为指数，指数位于底数的右上角。

当指数n=0时，aⁿ = 1;

当n=2时，称为平方;

当n=3时，称为立方;

自然对数e：单位时间内，持续的翻倍增长所能达到的极限值。

e是一个无限不循环小数，其值约等于2.718281828459…..

用例：Mathf.Exp(2) = e \* e。

### Mathf.Log(float f, float p) : float 对数

返回以p为底f的对数。

用例：Mathf.Log(6，2) = 2.584963;

### Mathf.Log10(float f) : float 基于10的对数

返回基于10的f的对数。

用例：Mathf.Log10(100) = 2;

### Mathf.Abs(float f) : float 绝对值

计算并返回值f的绝对值。即结果如果为负，去除负符号。

用例：Mathf.Abs(-10) = 10。

### Mathf.Min(float a, float b …) : float 最小值

对比传入的，两个或更多的参数，返回最小的那个传入值。

用例：Mathf.Min(1， 0， -1， 99) = -1 。

### Mathf.Max(float a, float b …) : float 最大值

对比传入的，两个或更多的参数，返回最大的那个传入值。

对比传入的，两个或更多的参数，返回最大的那个传入值。

用例：Mathf.Max(1， 0， -1， 99) = 99。

### Mathf.Ceil(float f) : float 上限值

返回浮点数f，大于或等于最接近f的整数。

用例：Mathf.Ceil(10.0) = 10; Mathf.Ceil(10.1) = 11; Mathf.Ceil(-10.9) = -10；

### Mathf.Floor(float f) : float 下限值

返回浮点数f，小于或等于最接近f的整数。

用例：Mathf.Floor(10.0) = 10；Mathf.Floor(10.9) = 10；Mathf.Floor(-10.1) = -11；

### Mathf.Round(float f) : float 四舍五入

返回浮点数f的四舍五入后最接近的整数。

如果数字未尾是.5，将返回偶数。

用例：Mathf.Round(10.1) = 10；Mathf.Round(10.6) = 11；Mathf.Round(10.5) =10；

Mathf.Round(11.5) = 12;

### Mathf.Sign(float f)：float 符号

返回f的符号，即是正还是负。当f为正数或为0返回1，为负返回-1。

用例：Mathf.Sign(-10) = -1；Mathf.Sign(10) – 1;

### Mathf.Clamp(float v, float min, float max) : float 限制

限制值v在min和max之间。即如果v小于min，返回min的值;如果v大于max，返回max的值;否则返回v的值。

用例：Mathf.Clamp(10, 2, 20) = 10;

### Mathf.Approximately(float a, float b) : bool 近似

比较两个浮点数值，看它们是否非常接近。

### Mathf.InverseLerp(float f, float t, float v):float 反插值

计算两个值之间的Lerp参数。也就是v值在f值和t值之间的比例值。

使用例子:Mathf.InverseLerp(5,10,8) = 3/5 =0.6;

### Mathf.DeltaAngle(float c, float t):float 增量角

计算给定的两个角之间最短的差异。

用例：Mathf.DeltaAngle(1080, 90) = 90;

### Mathf.CorrelatedColorTemperatureToRGB(float k) : Color

计算返回色温k转换后的RGB颜色值。K取值在1000到40000之间。

K以开尔文为单位，是国际中温度单位。

色温是指绝对黑体从绝对零度(-273度)开始加温后呈现的颜色。黑体在受热后，逐渐由黑变红，转黄，发白，最后发出蓝色光。当加热到一定的温度，黑体发出的光所含的光谱成分，就称为这一温度的色温。

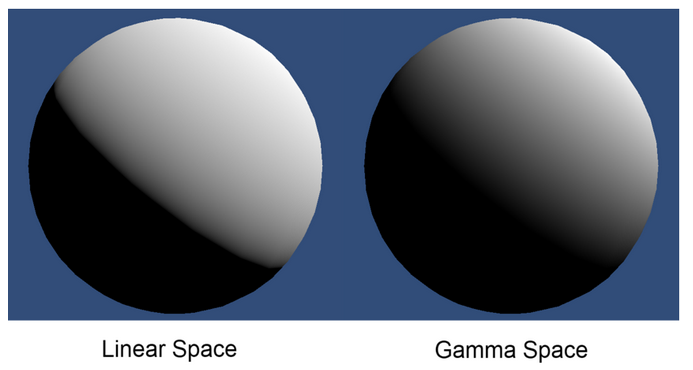
绝对黑体在2527k时的颜色相同。

### Mathf.GammaToLinearSpace(float v):flaot

将给定伽玛色彩空间(Gamma)的v值(sRGB)，转换到线性色彩空间。

线性渲染就是渲染场景所有输入都是线性的。一般来说存在的纹理都是经过Gamma矫正的，也就是说当纹理被采样到一个材质上是，颜色值已经不是线性的了。如果这此纹理用通常的计算方式去计算光照和图片效果，在非线性空间计算，这将导致轻微的偏差。

而线性渲染保证了在shader中输入输出的都在正确的颜色空间得出更正确的结果。



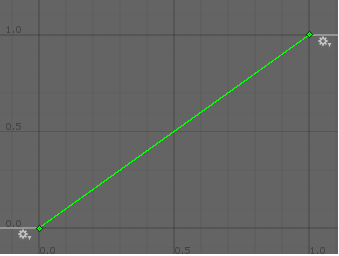
### Mathf.LinearToGammaSpace(float v):float

将给定的线性空间下的值v，转换到伽玛色彩空间(sRGB)。

**----以下函数需在帧函数或持续时间调用的函数中，才能体现效果---**

### Mathf.Lerp(float f, float to, float t) ： float 插值

基于浮点数t返回f到to之间的插值，t限制在0~1之间。

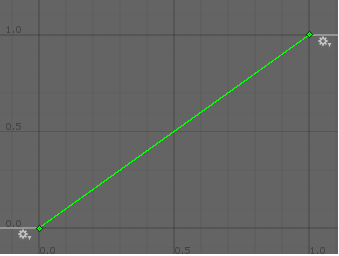


### Mathf.LerpUnClamped(float f, float to, float t)：float 插值

基于浮点数t返回f到to之间的插值，t不受限制。如果0>t>1,则返回f与to以外的值。

### Mathf.LerpAngle(float a, float b, flaot t)：float 插值角度

同上，a和b代表度数（0~360度）。



### Mathf.MoveTowards(float c, float t, float m):float 移向

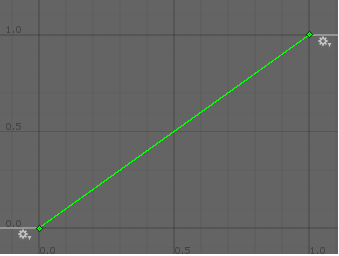
改变一个当前值几目标靠近。C=当前值，t=目标值，m=最大速度限制，如果为负值，将反方向作用。

实际上和Mathf.Lerp相同，但该函数确保我们的速度不会超过m值。

用例：void Update(){

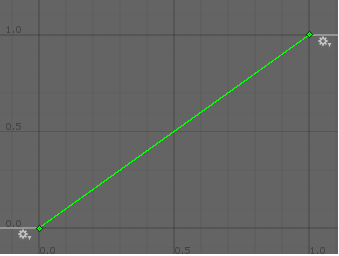
float vx = Mathf.MoveTowards(transform.position.x, target, speed \* Time.deltaTIme);

transform.position = vector3(vx, 0, 0); }



### Mathf.MoveTowardsAngle(float c, float t, float m):float移动角

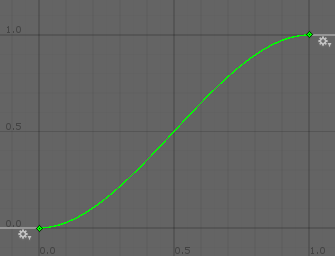
同上，变量c和t是作为度数。为优化原因，m的负值不被支持，要实现从t到c移动，添加180度角代替。



### Mathf.SmoothStep(float f, float to, float t):float 平滑插值

基于浮点数t返回f到to之间的插值，并在限制处渐入渐出。t限制在0~1之间。

同Mathf.Lerp相似，只是多了在 限制处渐入渐出。



### Mathf.SmoothDamp(float c, float ta, ref float cv, float st, float ms, float dt):float平滑阻尼

随着时间的推移逐渐改变一个值到期望值。这个函数可以用来平滑任何类型的值、位置、颜色、标量。

Current:当前位置。

Target:试图到达的位置。

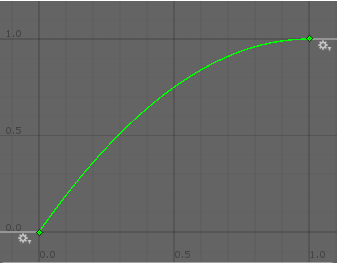
CurrentVelocity:当前速度。

smoothTime:要到达目标位置的近似时间，实际要快一些。

MaxSpeed:可选参数，允许限制的最大速度。

DeltaTIme:上次调用该函数到现在的时间 ，缺省值为Time.deltaTime。

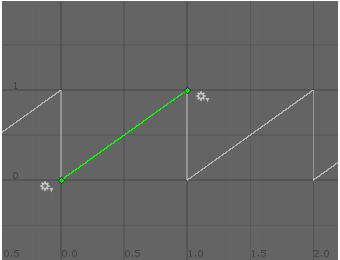
另Mathf.SmoothDampAngle()方法同上。



### Mathf.Repeat(float t, float len)：float 重复

循环数值t，从0到len之间。T值永远不会大于len值，也不会小于0。

每次结果是：0~len;0~len;



### Mathf.PingPong(float t, float len):float 乒乓

使数值t，在0到len之间往返。T值永远不会大于len值，也不会小于0。

每次结果是：0~leng,leng~0;

