Unity是顺时针绘制物体的

Unity使用unity\_SpecCube0，需要点上light里的auto generate。然后rendersetting的perObjectData要设置PerObjectData.ReflectionProbes。然后shader使用，上面这些少一步都不行哦，真麻烦。。。

**Msaa发生在什么时期，光栅化阶段还是片段着色器之后**

Msaa每个采样点为什么要有独立的深度与模板信息呢，他不是只针对每个像素做采样吗

自我回答：可分为两步，1是在光栅化阶段，每个像素都可以通过插值得到对应的深度与模板信息，但在每个像素新增四个sample点（称为2X），每个采样点判断与三角形的位置关系，那么由于每个像素多了4个sample，所以最终的结果图也大了4倍（可以看作生成了一张大四倍的纹理），最后在片段着色器之后，在进行加权平均计算，将结果还原为一张正常大小，覆盖到framebuffer上，最后输出。（不知道对不对，但应该差不多吧————）

**Batch和instance有什么区别：**

Batch是同材质切同纹理吧，会被自动批处理，用一个批次绘制。

Instance是主动用drawinstance接口渲染，同一个mesh进行多次绘制，材质要相同，但材质属性可以不同。

**使用重心坐标系做光栅化的优点是什么：**

**Mipmap中通过ddx与ddy求mipmap的层级：**

在映射在屏幕结果上取一个点，取其正上方的一个像素，这两个像素点分别再转换回ndc下，求这两个点的y的距离的平方，然后再求x距离的的平方，两个结果开根号。

再取其正右方一个像素，同样转换回ndc下，求这两个点的y距离平方与x距离平方，相加开根号。然后对这两个结果取max，结果就是L。

Level = log以2为底L的对数 L = 以上。

**Flat着色，gouraud着色和phone着色方式的区别：**

Flat：每个三角形面公用同一个发现（三个顶点共同贡献出面法线），然后公用这条发现去计算光照。所以每个三角形面颜色都是一样的

Gouraud：每个三角形的顶点都有自己的法线，先在每个顶点计算好颜色后，然后把结果进行像素差值。

Phone：每个三角形的顶点都有自己的法线，先将法线进行差值，之后每个像素都有了法线，在像素级进行光照颜色计算。

**色彩空间与gamma矫正：**

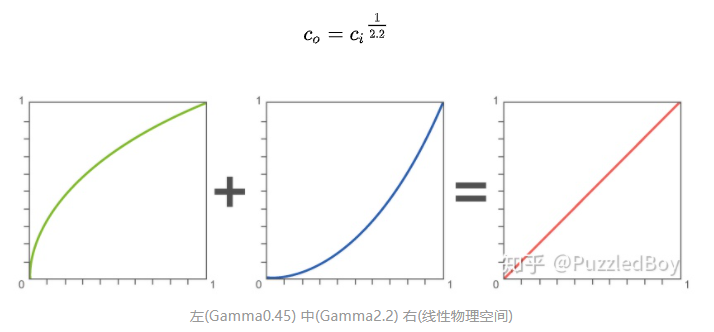
Gamma矫正是基于0-1颜色值之间的。

Gamma值越大越亮

矫正（linear->gamma）：pow(rgb,0.45) 反矫正(gamma->linear)：pow(rgb,2.2)

**Gamma矫正的由来：：：：**由于早期显示器是由阴极射线管来显示图像，亮度与电压不是线性关系，而是关于2.2次幂的非线性关系，这2.2叫做显示器的gamma值。但是真实的物理世界光的计算都是线性的，也就是线性空间，所以需要对gamma进行对应的处理。

**Gamma矫正的原理：：：**在数学上gamma矫正是一个约0.45的幂运算（和2.2互为逆运算）， C0 =C11/2.2。



所以一般先对图像做一次gamma矫正，当图输出时再经过显示器的2.2次幂运算就和真实物理世界一样了。

**什么是sRGB：：：**是一种标准色彩空间，对应是gamma0.45所在的空间。因为人眼对暗度比较铭感，所以尽量扩大暗度的存在范围。在sRGB下显示器自带2.2gamma，所以我们也不需要额外操作就能得到正确颜色。

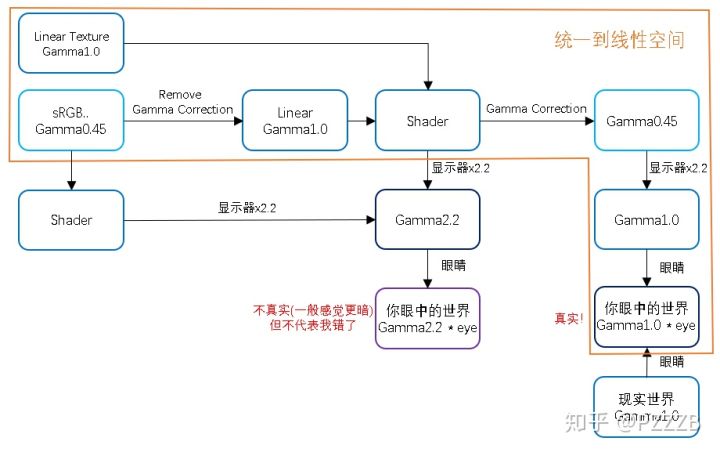


**统一到线性空间：：：**Gamma和线性空间的渲染结果是不同的，在表现上，gamma空间会偏暗些，linear更逼近真实世界。

在开发时，你要很清楚每个纹理在什么空间，项目用的什么空间等等才能得到好的渲染结果。

**下面是线性空间工作流，请详细阅读：**

统一到线性空间的过程是看起来是这样的，用图中橙色的框表示（现在看不懂图没关系，跟着后面的步骤来一步步看）：



我们从橙色框的左上角出发。

第一步，输入的纹理如果是sRGB（Gamma0.45），那我们要进行一个操作转换到线性空间。这个操作叫做**Remove Gamma Correction**，在数学上是一个2.2的幂运算  。如果输入不是sRGB，而是已经在线性空间的纹理了呢？那就可以跳过Remove Gamma Correction了。

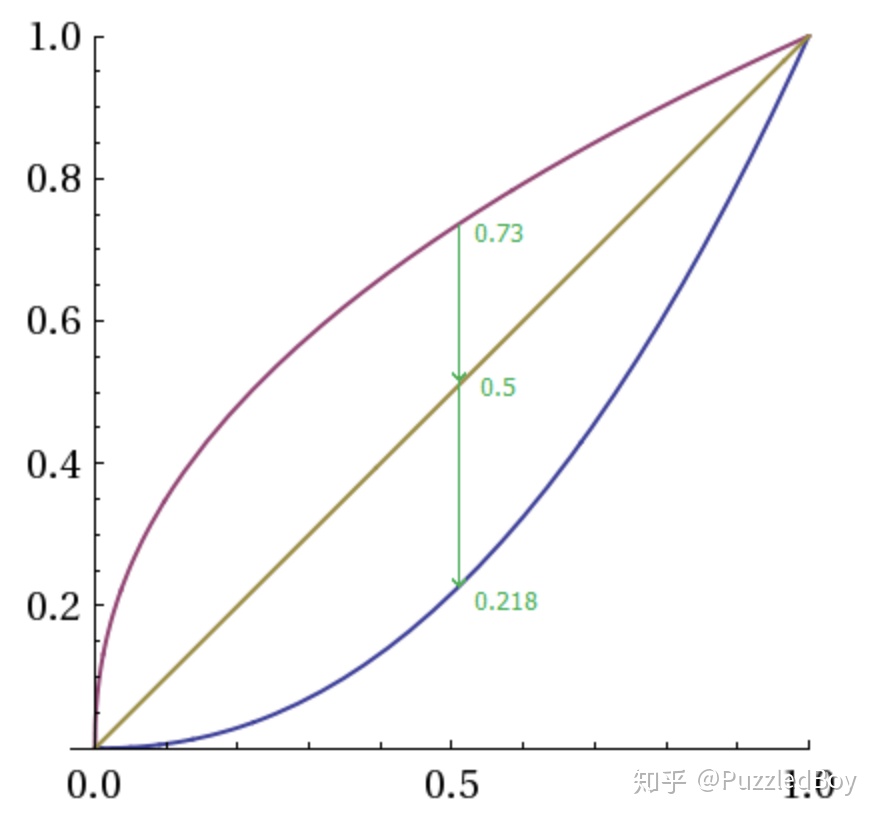
**注**：美术输出资源时都是在sRGB空间的，但Normal Map等其他电脑计算出来的纹理则一般在线性空间，即Linear Texture。详见后文！

第二步，现在输入已经在线性空间了，那么进行Shader中光照、插值等计算后就是比较真实的结果了（上文解释了哦~），如果不对sRGB进行Remove Gamma Correction直接就进入Shader计算，那算出来的就会不自然，就像前面那两张球的光照结果一样。

第三步，Shader计算完成后，需要进行**Gamma Correction**，从线性空间变换到Gamma0.45空间，在数学上是一个约为0.45的幂运算  。如果不进行Gamma Correction输出会怎么样？那显示器就会将颜色从线性空间转换到Gamma2.2空间，接着再被你看到，结果会更暗。

第四步，经过了前面的Gamma Correction，显示器输出在了线性空间，这就和人眼看物理世界的过程是一样的了！

我们再举个例子，我们取sRGB纹理里面的一个像素，假设其值为0.73。那么在统一线性空间的过程中，它的值是怎么变化的？



第一步，0.73(上曲线) \* [Remove Gamma Correction] = 0.5(直线)。（  ）

第二步，0.5(直线) \* [Shader] = 0.5(直线)（假设我们的Shader啥也不干保持颜色不变）

第三步，0.5(直线) \* [Gamma Correction] = 0.73(上曲线)。（  ）

第四步，0.73(上曲线) \* [显示器] = 0.5(直线)。（  ）

如果不进行Gamma Correction，就会变暗，因为第三步不存在了，第四步就会变成：

0.5(直线) \* [显示器] = 0.218(下曲线)。（  ）

再对照上面的图琢磨琢磨？

Unity中有gamma和linear两种，纹理上还有srgb的选项：

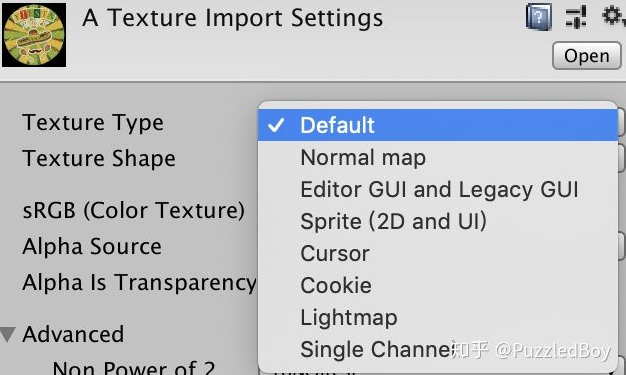
在gamma空间下一般无论时候勾选srgb值，一般纹理都不会有什么变化，因为一般美术作图的时候默认就是gamma空间下了，输出的图也是进行gamma矫正过的。

在linear空间下勾选srgb，纹理会保留gamma矫正。不勾选会移除gamma矫正（进行反gamma矫正）。

**Unity中的ColorSpace：：：**当你选择gamma色彩空间，Unity不会对输入输出做任何处理，也就是没有removeGammaCorrection和GammaCorrection，除非自己手动实现。

如果选了linear，就是上面的线性空间流程。对于sRGB纹理，Unity在进行纹理采样前会自动进行RemoveGammaCorrection。对于非sRGB的线性纹理则没有这一步。而在输出前，Unity会自动进行GammaCorrection

对于类型为NormalMap和LightMap的纹理都是Linear，Unity会自动处理



还有一些纹理不是上面的任何类型，但又已经在线性空间了例如：（mask纹理，噪声图等），就手动点击取消sRGB按钮，让他们跳过RemoveGammaCorrection过程

**到底什么纹理应该是sRGB，什么是Linear？**

关于这一点，我个人有一个理解：所有需要人眼参与被创作出来的纹理，都应是sRGB（如美术画出来的图）。所有通过计算机计算出来的纹理（如噪声，Mask，LightMap）都应是Linear。

这很好解释，人眼看东西才需要考虑显示特性和校正的问题。而对计算机来说不需要，在计算机看来只是普通数据，自然直接选择Linear是最好的。

除了纹理外，在Linear Space下，Shaderlab中的颜色输入也会被认为是sRGB颜色，会自动进行Gamma Correction Removed。

有时候你可能需要想让一个Float变量也进行Gamma Correction Removed，那么就需要在ShaderLab中使用[Gamma]前缀：

[Gamma]\_Metallic("Metallic",Range(0,1))=0

如上面的代码，来自官方的Standard Shader源代码，其中的\_Metallic这一项就带了[Gamma]前缀，表示在Lienar Space下Unity要将其认为在sRGB空间，进行Gamma Correction Removed。

**3D纹理采样实现以及天空盒实现细节：**

首先对3d纹理的规定一定是width是4的倍数，height是3的倍数。

加载3d纹理：然后分别读取一张整图，将其分割为六张小图，以此用于天空盒的六个面。将读取到的长宽分别除4和3，表示然后以此规定每张小图在大图中的起始位置（每张图以左下角代表为原点）然后将大图中对应位置的六张小图的数据拷贝到六张纹理中，记得规定各个面熬，这时3d纹理的加载完成了。

3d纹理采样：采样的依据就是立方体盒子的每个顶点坐标作为采样的3duv（normalize后的哦），得到之后xyz三个分量绝对值的最大值来决定大致在哪个方向上采样，然后在逐步分析xyz各自的正负性，然后把对应的采样点转换到对应采样面上去。此时，每个面的中心是（0,0），要变换为左下角为中心（0,0）点，因为这才是纹理坐标的规范。接着在对应的采样面上根据得到对应采样面上的采样点进行普通的纹理采样就可以了。

细节：

\*\*\*在vs中，要将顶点的z设置为与w一样，这样到了cdn下可以保证盒子一直在远裁剪面处。

\*\*\*不需要model矩阵，始终在世界中心，乘了也是白乘。

\*\*\*要注意，由于摄像机一直处于盒子内部，所以要把背面剔除改为正面剔除

\*\*\*要注意修改深度测试函数，如果测试函数是小于的话，那么要改为小于等于，否则盒子深度一直等于1，一直被剔除，不能显示出来。

**一次drawcall做了什么，等价在软渲染器上相当于什么过程：**

**HLSL中constantBuffer的概念：**

引入：-----------------------------

在dx10中被引入，dx9中发送到GPU命令缓冲区的80%的数据是着色器常亮数据（constant Data），很多引擎发送所有的绘制调用更新，例如vs阶段和fs阶段分别访问相同的常亮，你不得不设置两次常量，cbuffer出现之前，你没有办法来保存之前存在的常量。

Cbuffer可以绑定到不同的着色器阶段，没有必要多次存储，在需要改变是才进行更改。但不幸的是，将其绑定到不同的缓存区，修改的代价就变得很大，实际过程中我们要竟可能避免这样的情况。

Cbuffer结构的设计：----------------------------------------

根据不同的更新频率来将更新的值放到不同的缓冲区中。

布局也要合理，和c++的内存对齐是一样的，要注意定义时的顺序，cbuffer是16字节（float4）对齐的。对于大小小于64k的缓存而言，cbuffer要比struct的性能好，Unity的GPUinstance也是用cbuffer实现的，Unity有UNITY\_MAX\_INSTANCE\_COUNT是500，也就是一次绘制调用最多500个实例。

**深度值得学习：**

透视投影后传：

\*Opengl中的GL\_PROJECTION是透视投影矩阵和透视除法整合到一起的矩阵。

\*opengl的ndc范围是[-1,-1,-1]-[1,1,1]。Dx的ndc范围是[-1,-1,0]-[1,1,1]

**颜色乘法与加法的意义：**

加法：简单的说，不同光的运算结果就是相加

乘法：简单的说，光作用在材质上的结果就是相乘。光作用在材质/贴图上，贴图吸收后反射出来的光就可以用乘法模拟，所以用乘法。

C++疑问：

我想通过material中的shader调用对应shader中的一些方法，那么调用时候怎么能知道闯进来的是哪个子类的实例呢.除了dynamic\_cast有什么好方法吗。

class Material

{

Void SetShader(Shader& sd);

Shader\* shader

}

class Shader…

class PhongShader:public Shader…

class GourangSahder:public Shader…

Shader\* pSd = new PhongShader();

Shader\* gSd = new GourandShader();

Mat.setShader(\*gSd);

传进来的是父类指针/引用，如果想调用子类特有的方法，是不是只能new个子类出来或cast为子类。

为什么float\*和char\* 使用memcpy，float\*结果是错误的，使用copy才是对的。Memcpy一般用来拷贝串这没问题，因为他就是在<string.h>头文件中

Batch：调用一次渲染API接口例如（glDrawArrays/glDrawElements），来向GPU提交相同渲染状态的一定数量的三角形的行为为一个渲染批次。

优化总结：

\*想方设法减少贴图尺寸

\*如果qualitysetting里texturequality为half，那么在现有的效果上可以降低贴图尺寸(降低后同时不再设置为half)，因为mip0是不会采样的

\*检查变体数量，评估运行时shaderlab内存占用，减少变体。项目中不能有内置shader，会有很多变体，不能用内置的材质，会使用内置shader导致加载进来默认shader。

\*每秒钟可处理的Batch数和CPU好坏有关系，和GPU没啥关系，要在保证GPU没到性能瓶颈的情况下去增大Batch。

发现dx一次drawindexed超过65520（大约是unsigned short），会在来个批次。

\*动态合批未必真的可以降低消耗，ui上应该可以，但场景中的未必可以，因为合批为了减少DC，减少DC可以降低CPU消耗，但动态合批又要增加CPU的计算量，有可能数字变了，但未必真的快了。

\*三线性插值是比较耗的，但双线性采样基本不怎么消耗。因为三线性要比较大的带宽去在mipmap中插值

\*降低纹理尺寸，理论上除了比较精细（光照贴图，人物，大一些的场景模型贴图）的地方要上1024，一般都用不太1024，项目中1024的图比较多，之后可酌情调整优化一下。理论上几乎所有的模型贴图都可以压缩（例如一些不那么频的贴图可以压缩，ETC属于有损压缩，一般法线或颜色细节变化明显的贴图可以不压缩，但也要酌情去决定），发现有很大一部分贴图没压缩，ui压缩有会降低颜色位数时需要和美术确认效果。

\*带宽是指一定时间内传输的数据量，带宽占用高导致收集容易发热，发热严重会导致降频，影响性能，理想的带宽标准是1.5G/s以内。

\*设置textureQuality为half，是在GPU中去控制使用贴图的大小，可以减少带宽，但内存还是那么多

\*使用八面体映射压缩天空盒面数，降低包体大小。打出来的ab，其中纹理的ab大小由其自身大小和像素的复用率决定。

\*解ab包内容用AssetStudio。查看shader编译后运行效率可使用arm的离线shader编译器跑一下。<https://developer.arm.com/documentation/101863/7-4/Using-Mali-Offline-Compiler/Compiling-OpenGL-ES-shaders> 使用教程看文档。

\*MeshCompression压缩mesh降低包体。加载时会有解压的过程。启用了压缩mesh，会有两个坑，1：AUP会失效 2：TextureStream会失效

\*AUP ：Async Upload Pipeline 2018.3之后支持Texture和Mesh 2018.2支持Texture

开启后Texture和Mesh资源都是在加载线程加载好由渲染线程提交到GPU的。关闭时Texture和Mesh都是在主线程提交到GPU，降低主线程处理量，可以处理更多逻辑事物。如果Texture和Mesh量很大时，渲染线程也可能是吃不消的，要酌情考虑同步和异步加载。

踩坑记录：

\*\*\*垃圾es2.0不支持一些语义，使用时要查一下，例如vertex\_id。

\*\*\*有好多辣鸡机器还不支持Depth格式纹理，new rt时候可能会闪退崩溃，判断平台是否支持也没用。 使用Depth格式时赋值给相机的maintexture时还会警告RenderTexture.Create: Depth|ShadowMap RenderTexture requested without a depth buffer. Changing to a 16 bit depth buffer。这时候需要再搞个垃圾colorbuffer来解决：<https://github.com/cnlohr/shadertrixx>

\*\*\*手动创建材质球同时用的自己的shader时，里面有关键字会失效，需要手动enablekeywards

\*\*\*graphicsetting里，fog模式设置为automatic，在打ab时候可能会丢失无效，因为相关keyword的变体会打不到ab里（编译后的shader代码），Unity会自动剔除fog的一些宏。设置为custom不会剔除。全局keyword超了上限有可能有些变体也打包不进去。使用shader variant collection打包不会丢变体，还可以进行预加载等好处。

\*\*\*用materialBlock设置材质属性后，不支持srpBatch了。Submit后，每个drawrenderers会发起一个sort和buildBatch的task，然后按顺序挨个写入uniformBuffer，逐个调用Drawcall