GAMES202 Lecture14-Glimpse of Industry Solution

The last lecture of games202,but it's not the end of studying rather a new start for me.

- 终于来到了Newcastle，开始了10天的quarantine，因此决定在这十天内把202收尾并且把101的东西重新看一遍和cherno的c++系列为之后游戏工程的课做好准备。
- 在此特别感谢强哥，一起来的路上对我照顾有加，永远的好大哥。

- **也十分感谢康姐，我在英国吃的第一顿热饭是康姐煮的汤圆，救了孩子一命。XD**

好了回归正题。

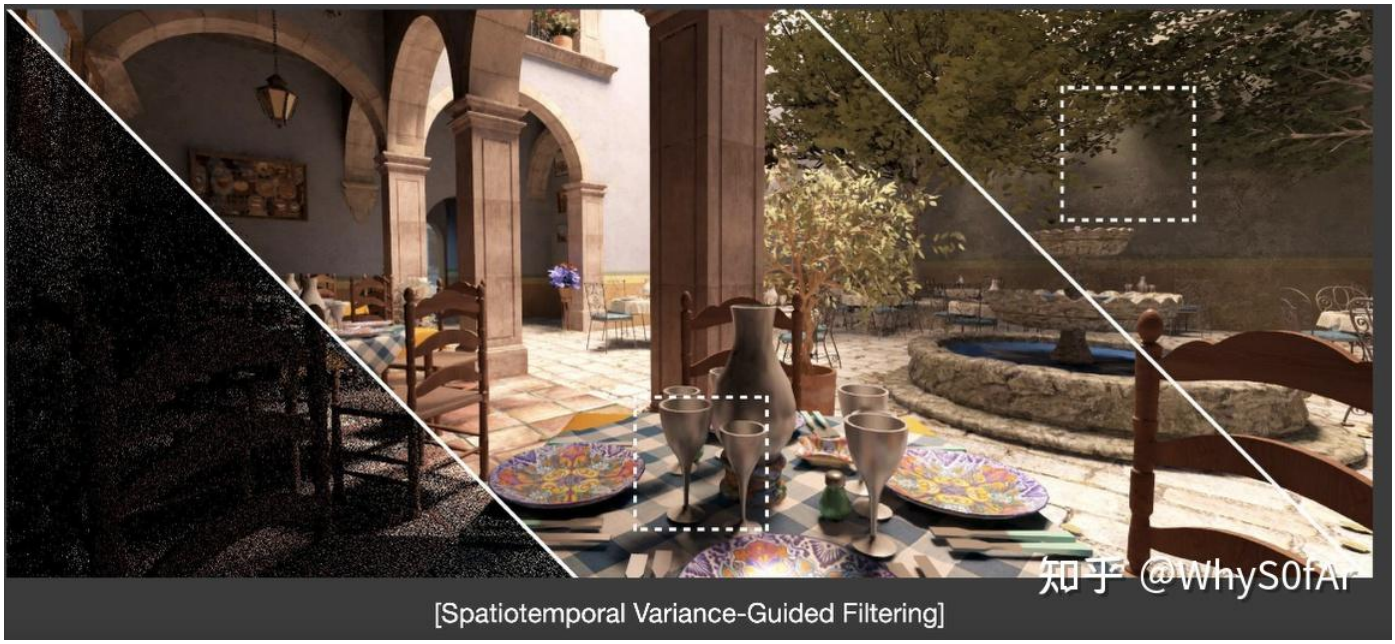
本节课主要讲一下工业界的一些技术.

前情回顾:

本节内容:

Specific Filtering Approaches for RTRT

SVGF-Basic Idea



1spp结果 SVGF结果 ground turth

我们可以看到ground truth和svgh结果非常接近,这说明svgf效果很不错.

- SVGF的方法与在时空上降噪的方法差不多
- 但是,加了一些variance analysis和tricks

SVGF-Joint Bilateral Filtering

三个指导filtering的重要因素:

1.Depth:

以图中的A,B互相贡献为例,还记得我们上节课说的

如果二者之间的深度差异小,则贡献权值大

反之二者深度差异大,则贡献权值小.

$$w_z = exp(-\frac{|z(p)-z(q)|}{\sigma_z|\Delta z(p)\cdot(p-q)|+\epsilon}) \qquad (1)$$

我们来分析一下公式的各个部分:

这是SVGF用来判断深度贡献权值的公式,从公式中可以看出:

由于exp(x)是返回e的x次方,由于公式返回的是-x次方,所以差异越大,贡献越小.

但是分母部分多了个 σ_z,多了个 $\left|\Delta z(p)\cdot(p-q)\right|$(深度的梯度 * 两点间的距离),还多了一个 ϵ.

ϵ的作用是因为我们filter时候考虑一个点Q周围所有的点P,也包括这个点Q,因此它是有可能为0的,避免分母为0的情况,而且如果两点足够接近会导致最后的数值太大,为了避免一些数值上的问题加上一个ϵ.

σ_z是一个用来控制指数衰减的快慢的参数,或者理解为控制深度的影响大还是小.

至于$\left|\Delta z(p)\cdot(p-q)\right|$我们以例子为例来理解:

1.A和B两点在同一个平面上,颜色也相近,所以理论上A和B会贡献不少给彼此

2.但是由于这一面是侧向面对我们的,因此A和B是有很大深度差异的.此时如果我们用深度来判断A和B对彼此的贡献时候,会发现它所给出的是A和B之间不应该有特别大的贡献,这明显是不合理的.

3.因此简单的用深度来判断贡献值是不行的,那么我们考虑A和B在平面法线方向上的深度变化,比如图中的A和B虽然在实际深度差异很大,但是在法线方向上的深度差异几乎没有,因为二者几乎在同一个平面上.

- 3 factors to guide filtering
  - **Depth**

$$w_z = \exp\left(-\frac{|z(p)-z(q)|}{\sigma_z|\nabla z(p)\cdot(p-q)|+\epsilon}\right)$$

  - Understanding:
    - A and B are on the same plane, of similar color, so they should contribute to each other
    - But the depth between A and B are very different!

首先我们要知道,深度的梯度就是往某一方向上的变化率,因此当我们知道A和B之间的距离,之后用 **深度梯度 X 距离 = 实际深度变化量.** ($|\Delta z(p)\cdot(p-q)|$)

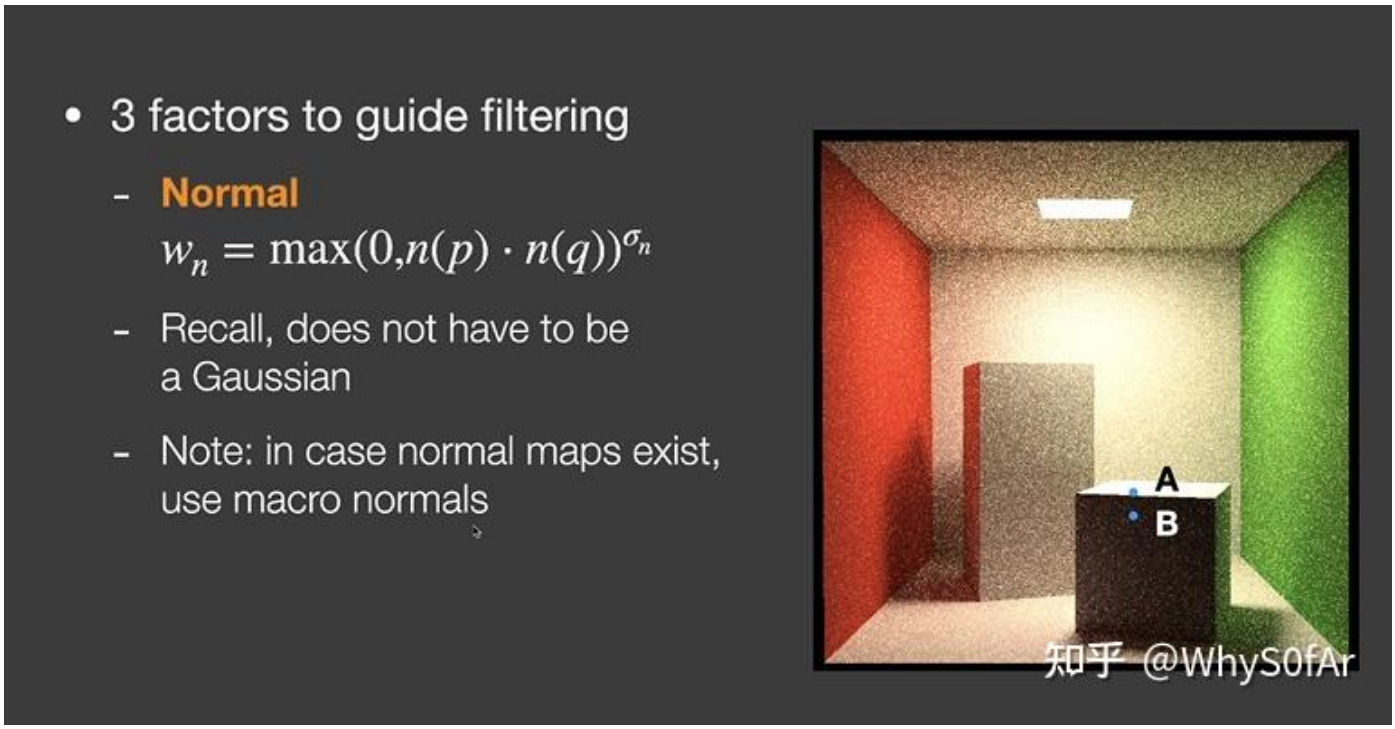
(垂直于平面法线上的变化 * 距离 = 实际深度变化量)

如果实际深度变化量也很大,从而告诉我们这个平面是侧向我们的,在这种情况下来看公式,虽然分子深度差异大,但是分母中的梯度也大,二者一除,值就没那么大了,因此EXP()得到的值就大了.

总结来说,通常情况下不会简单的用二者之间的深度差异来判断贡献值,而是在他们所在的面的法线方向上投影出来的深度差异来判断,或者说是在它们所在平面上的切平面上的深度差异.

TIPS:平面的切平面等于平面本身.

2.Normal



$$W_n = max(0, n(p) \cdot n(q))^{\sigma_n}$$

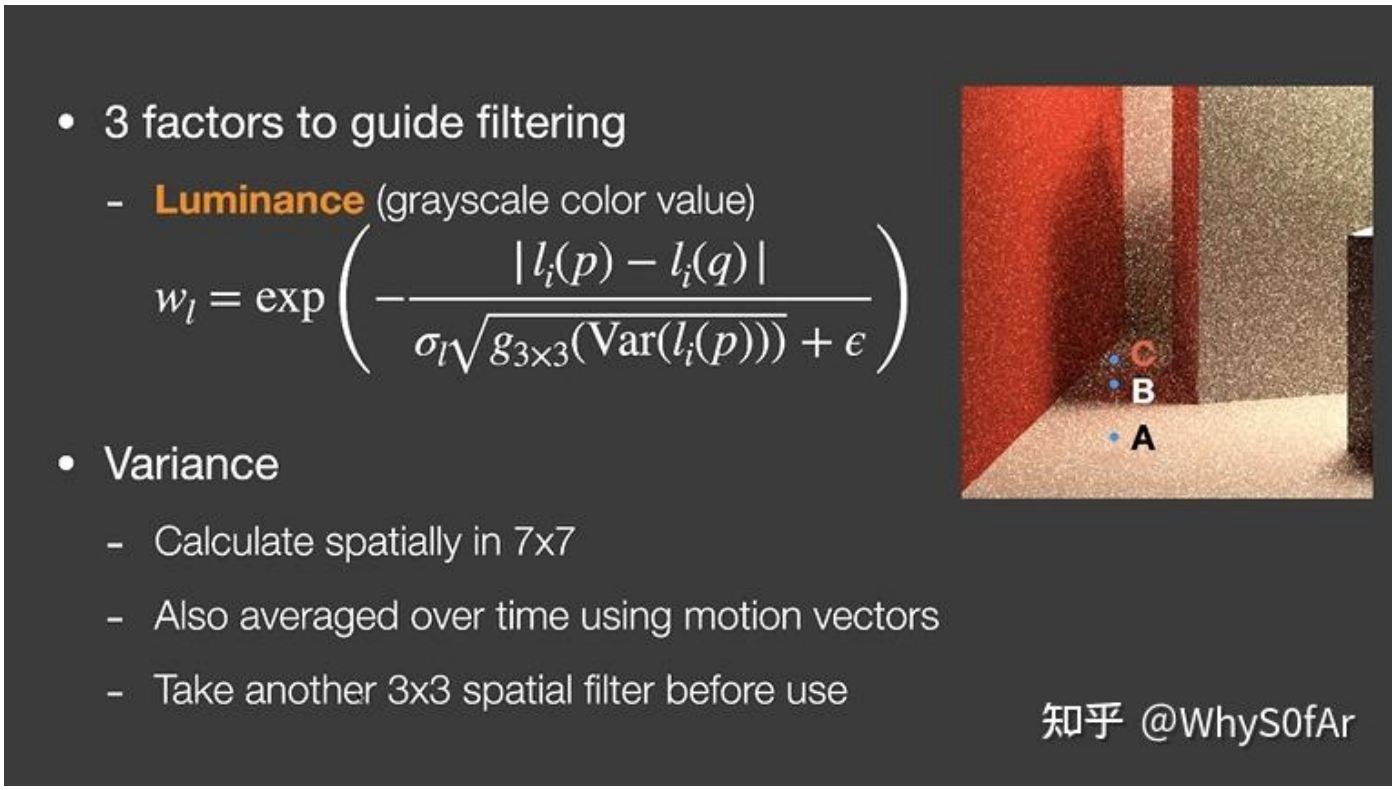
我们用两个点法线向量求一个点积,由于求出来的值有可能是负值,因此使用max()把负的值给clamp到0。

如果两个向量相同则点乘结果为1,点乘结果为负时则为0，点乘结果为正时则使用这个结果.

至于 σ_n是用来判断点乘的这个cos函数从1到0是快还是慢,σ_n越大,从1到0就越快,也就是判断法线之间的差异的严格程度.

另外,如果场景中运用了法线贴图来制造凹凸效果,我们再判断时运用平面原本的法线,而不是为了制造凹凸效果而改变过的法线.

3. Luminance



Luminance (grayscale color value)

$$w_l = exp(-\frac{|l_i(p)-l_i(q)|}{\sigma_l \sqrt{g_{3\times3}(Var(l_i(p)))}+\epsilon}$$

我们在考虑颜色差异时，最简单就是应用双边滤波里给的颜色差异来考虑，比如我们将RGB转换为grayscale（灰度），这种颜色我们称其为luminance，只是一个名字，我们就认为其是颜色。

任意两点间我们看他们的颜色差异，如果颜色差异过大，则认为靠近与边界，此时A和B的贡献不应该互相混合起来，比如A不应该贡献到B，B也不应该贡献到A。

由于噪声的存在会出现一些干扰，也就是B点虽然在阴影里，但是可能刚好选择的点是一个噪声，也就是其特别亮，此时A特别亮，B也特别亮，那么A和B就会互相贡献，但是这样是错误的现象。

此时就是SVGF中V的作用了------》variance。

variance（方差）：我们先去看A，B两点间的颜色差异，并且思考filter的中心点的variance，当variance比较大时，我们不应该去过多的相信两点间的颜色差异。这就是为什么会去除以一个标准差。

eg:但我们在考虑A点是否贡献到B点时，先看A和B点之间的颜色差异，并且除以B点周围的一个标准差（$\sqrt{方差}$）。

我们要知道B点的variance，由于variance是一个统计学的量，不可能只看他自己本身，假设当前帧渲染出来之后的结果是很noisy的：

$$\sqrt{g_{3\times3}(Var(l_i(p)))}$$

- 在点的周围取一个7*7的区域算出区域里的variance。

- 同样的操作在时间上累积下来，所谓累积也就是在时间上平均下来，它可以去找上一帧中对应点的variance，从而通过motion vector在temporal上把variance信息给filter下来，这样先spatial filter求出variance之后，再随时间平均从而得到了一个比较平滑的variance值。
- 最后在使用variance时如果不放心，我们再在周围取一个3*3的区域做一次spatial filter得到variance。

也就是进行 spatial filter --> temporal filter --> spatial filter 从而得到B点精准的variance，

可以看到SVGF的结果总体还是不错的，但还是有一点点的噪声。

但是SVGF也有他自己无法处理的情况，当一个场景固定，我们只移动光源时候，阴影会随着光源的移动而变化，当前帧会复用上一帧的阴影，但由于没有发生任何的几何变换，因此motion vector等于0，此时复用上一帧时会产生“残影“现象，这是SVGF无法解决的一个问题。

RAE（Recurrent AutoEncoder）

RAE是指Recurrent AutoEncoder,用RAE这么一种结构对Monte carlo路径追踪得到的结果进行reconstruction，也就是对RTRT做一个filter从而得到一个clean的结果。

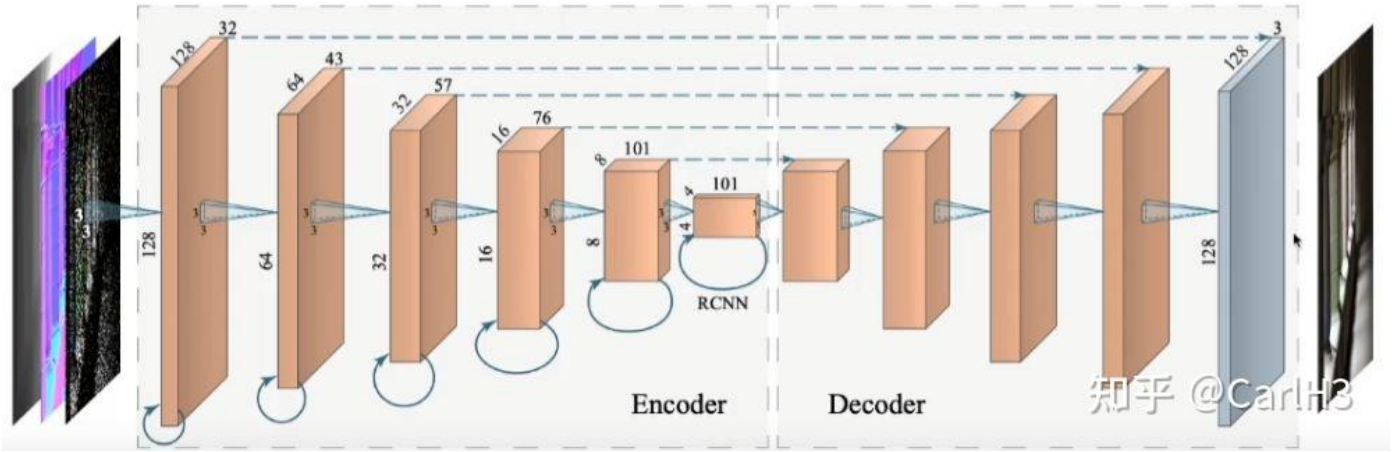
- 是一个后期处理降噪的方法，将一个noisy的图输入最后输出一张clean的图
- 会用到一些G-buffer上的信息，因此是与noisy的图一起作为神经网络的输入。
- 神经网络会自动将temporal的结果累积起来。

之所以不使用motion vector也可以将temporal的结果累计起来是因为两个设计原则”

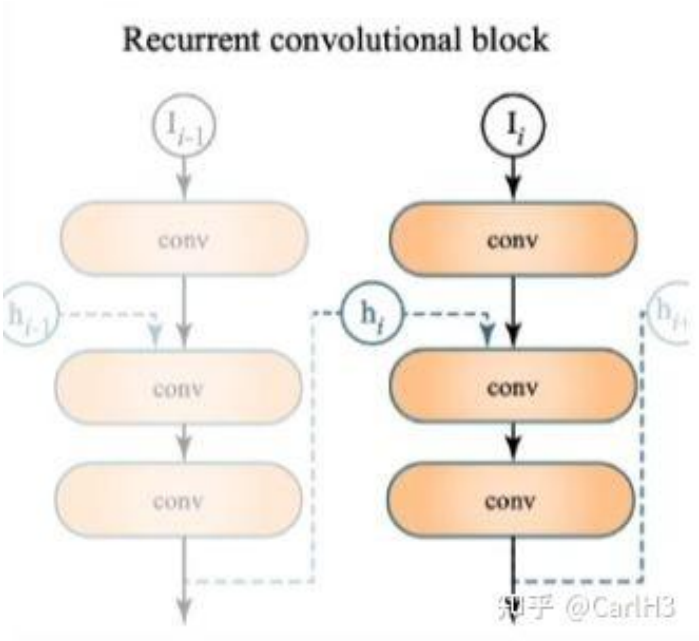
1. AutoEncoder,是一个漏斗形的结构，输入经过若干层神经网络后变成一个很小的东西，之后再将小的东西不断地展开，形状很像一个U字形，因此也可以称之为U型神经网络，其适合去做一些图像上的各种操作。

AutoEncoder

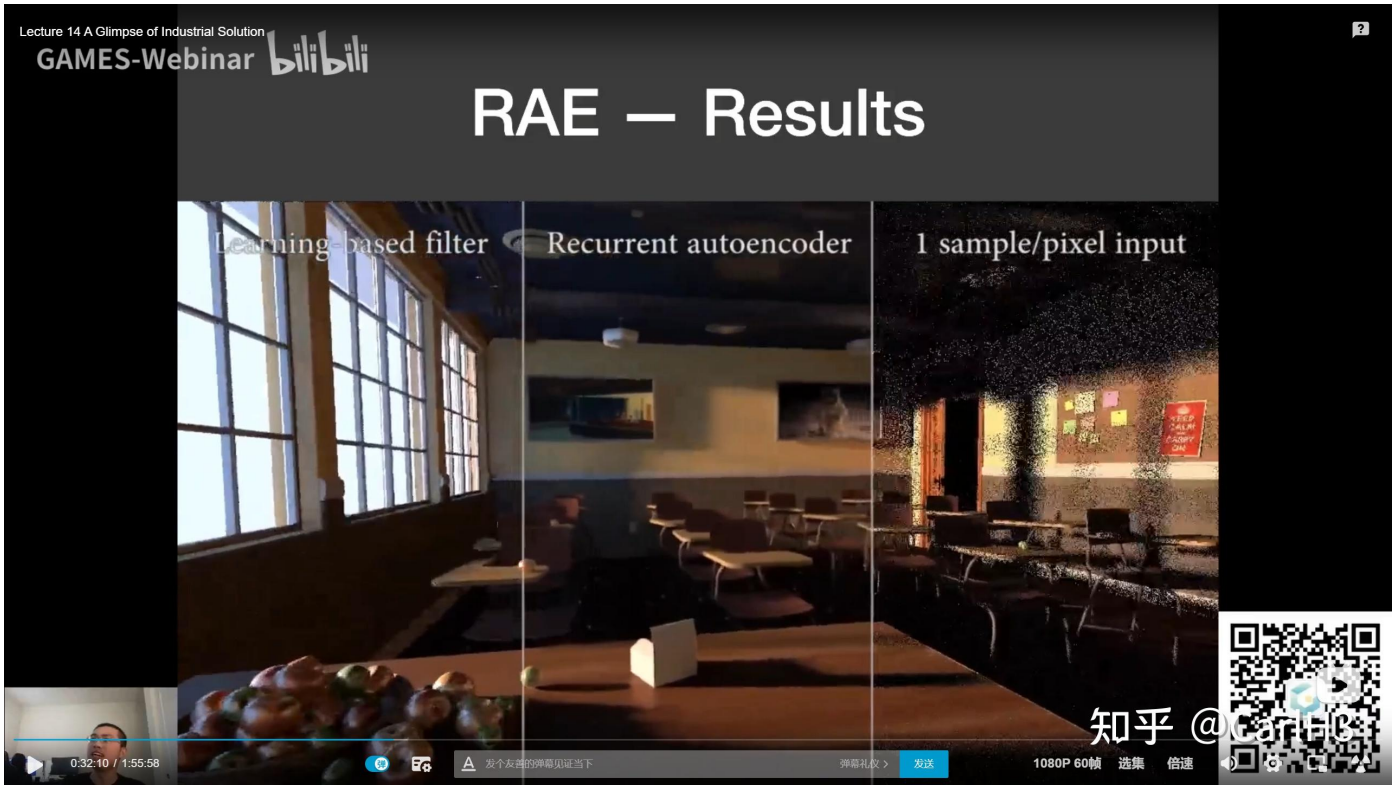
- Skip / residual connections for faster and better training



2.之所以可以利用历史的信息是由于每一层神经网络叫做convolution block但是有一个recurrent连接，也就是每一层神经网络不仅可以连向下一层，也可以连回自己这一层。因此假设神经网络一直在跑，跑完当前帧后，会有信息遗留在神经网络里面，信息每一层又可以连向它自己，因此在跑下一帧图时候，可以用神经网络自己学出来的方法去复用上一帧遗留下的信息而不是通过motion vector。



- Recurrent block
- Accumulates (and gradully forgets) information from previous frames



	Quality	Artifact	Performance	Explanability	Where did the paper go
SVGF	Clean	Ghosting	Fast	Yes	HPG
RAE (when first invented)	Overblur	Ghosting	Slow	No	SIGGRAPH

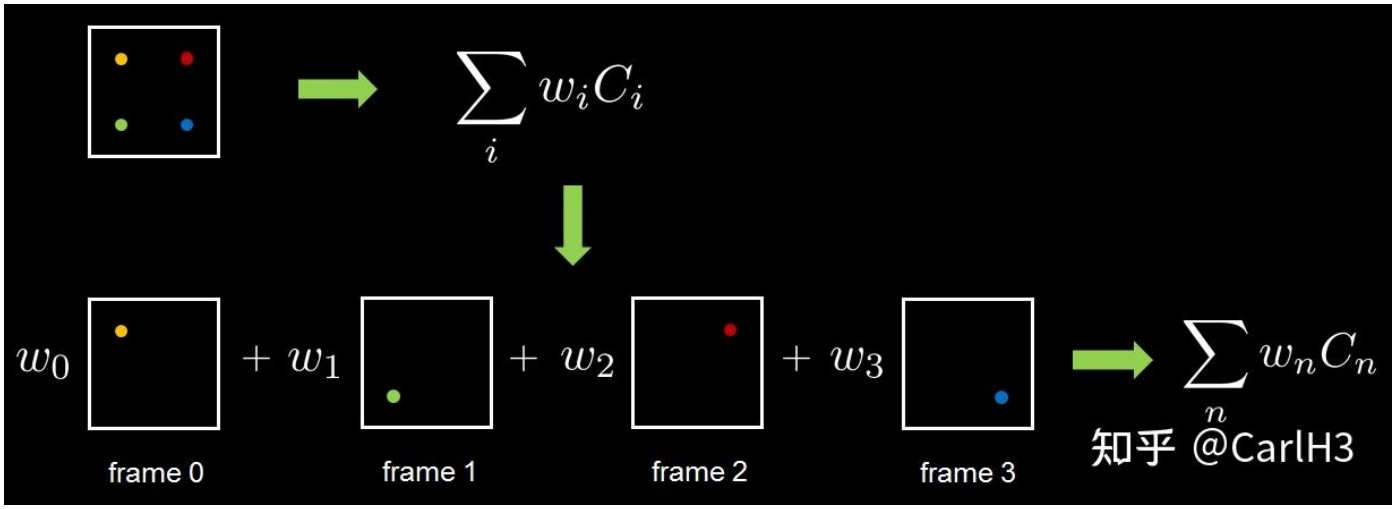
最早temporal的思路是用来解决Anti-Aliasing的，先有TAA的巨大成功才会有RTRT里的应用，因此我们接下来了解一下TAA。

Temporal Anti-Aliasing(TAA) 在时间上的抗锯齿或反走样

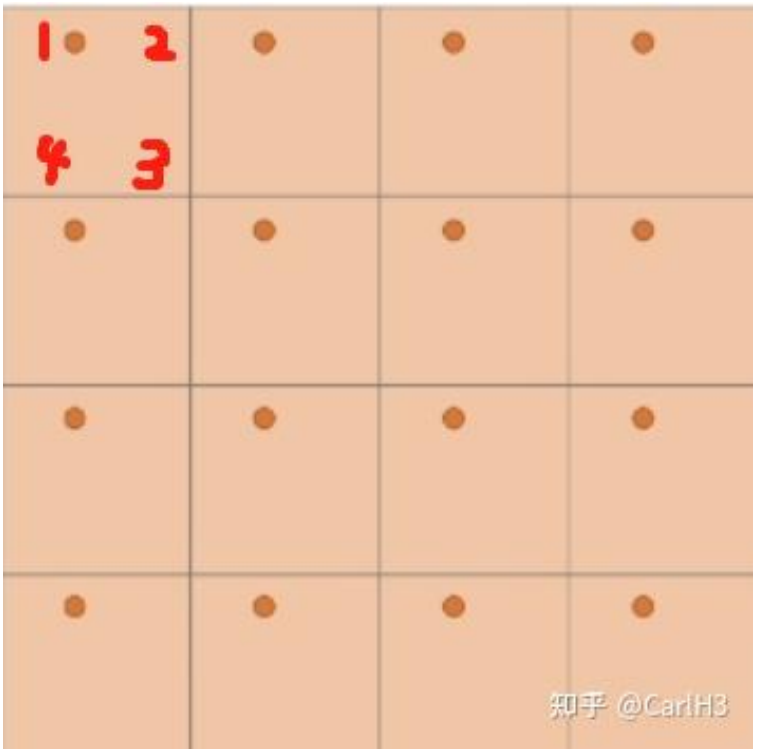
- Recall:why aliasing?
- Not enought samples per pixel during rasterization **一个像素中的样本数量不足**
- Therefore, the ultimate solution is to use more samples **终极解决方案就是用更多的样本，也就是101中的MSAA**

Tmeporal AA的思路也是需要用更多的sample，只不过是当前帧会复用上一帧的sample，使得这一帧仍然用1SPP，但是无形中通过temporal的reuse，增加了SPP。它的思路与RTRT中如何运用temporal的思路一模一样。

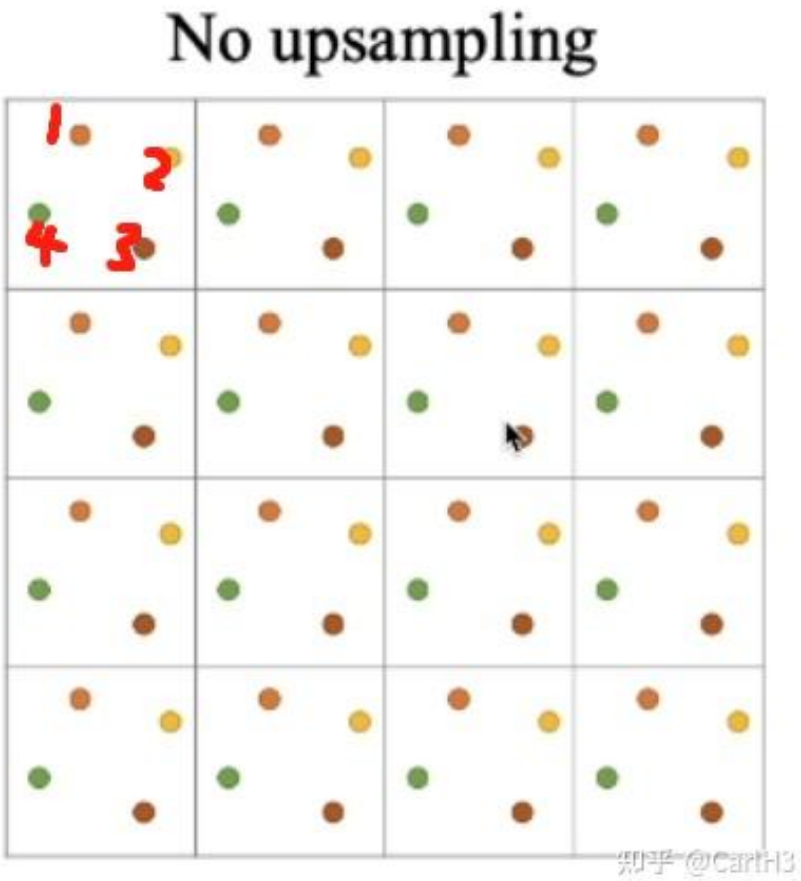
Temporal AA尝试用在避免性能损失的情况近似Super Sampling AA的结果。它的做法一句话总结就是，把样本分布到过去的N帧中去，然后每一帧从过去的N帧中取得样本信息然后Filter，达到N倍Super Sampling的效果，如下图。



我们考虑一个静止的场景，我们通过TAA提供的一个叫Jittered sampling的方法去复用上一帧的sample，假设当前帧1我们都像图中一样将样本分布在像素的左上角，上一帧的在右上角2，上上一帧在3，上上上一帧在4，我们可以认为连续的四帧之间有一个移动的pattern，他们在时间上各不相同，样本就这样从1，2，3，4这样的转.



假设我们在当前帧每个像素里只有1这个点，我们就可以复用上一帧每个像素中2这个点，由于是一个递归的过程，因此又可以复用再上一帧像素中3这个点，以此类推到4这个点，等于我们把temporal的这些样本都考虑进去了，由于我们的场景是静止的，就等于把之前各自样本的结果在一起做了一个平均。



这样就好象我们在当前帧中做了一个2*2的upsampling，两个结果是很接近的。

也就是通过将前N帧内的样本点的结果平均起来，其效果与在当前帧内增加样本点的效果一样。

之所以不随机采样点是因为随机的效果不一定好，因为在temporal中可能会引入额外的高频的信息，因此使用规定好的样本点，如上图每四帧一个，这样固定样本点位置避免了分布不均匀的情况。

当场景是运动的情况下，原本在静止情况下我们是在一个像素里找sample的结果并复用上一帧中的样本点的结果，在运动场景中，当前帧几何的位置通过motion vector找到上一帧中对应的位置，并复用其结果，如果temporal的信息不太可信时，使用clamping方法，也就是把上一帧的结果拉到接近当前帧的结果。

Notes on Anti-Aliasing 01

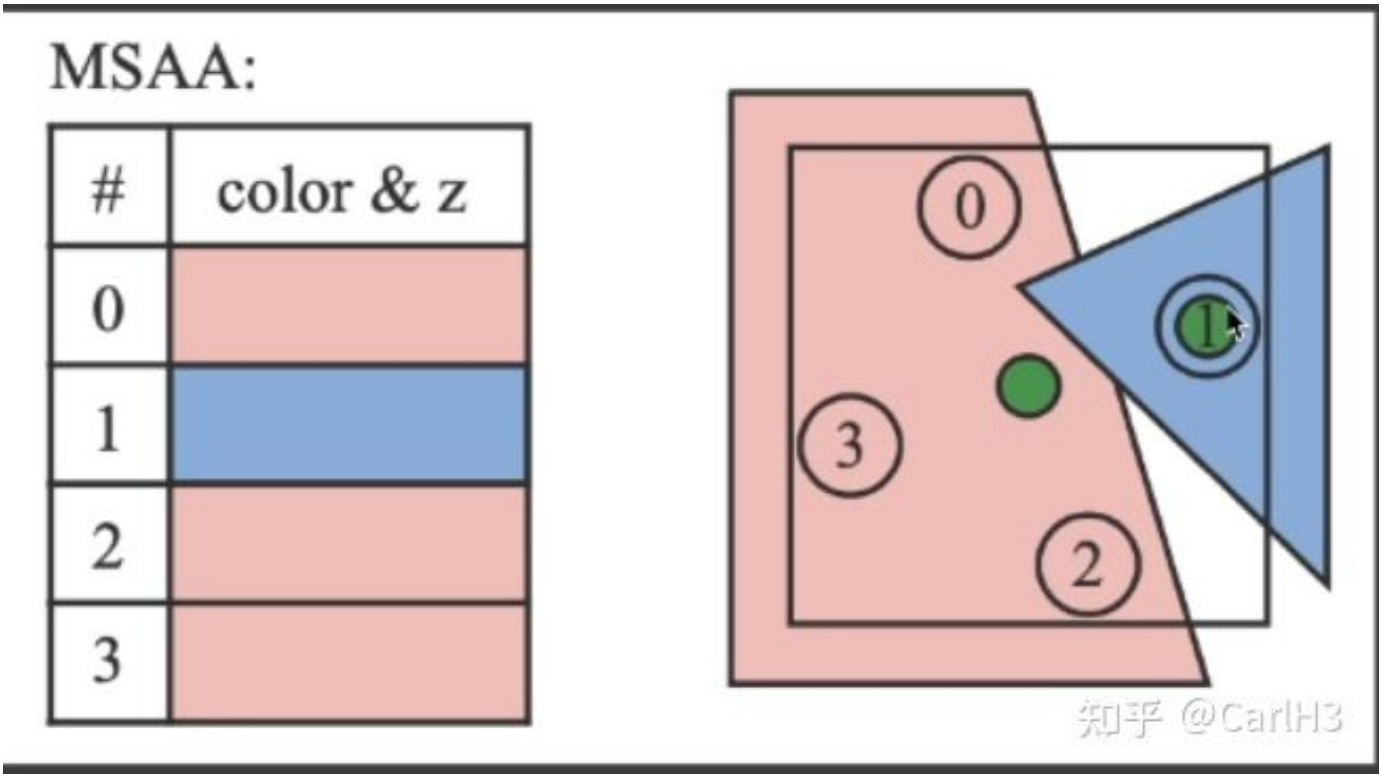
*MSAA*VS *SSAA*：

SSAA可以看作是将一个场景按照几倍的分辨率先渲染后再降采样，把几个像素的结果平均起来，思路是完全正确的，就是开销比较大。

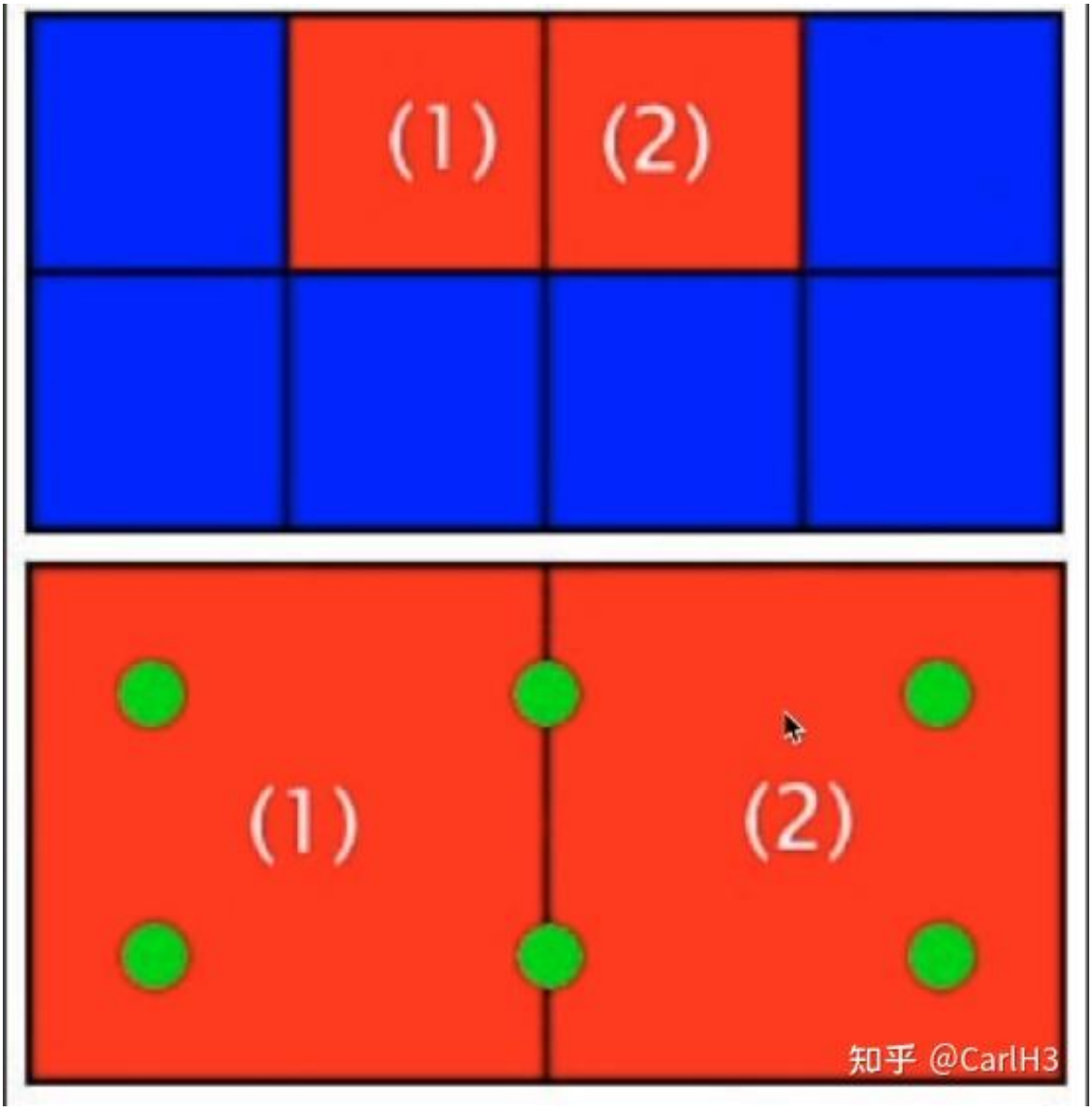
MSAA则是在SSAA的基础上做了一个近似从而使得其效率提升开销没那么大，如果我们用4X的MSAA时，帧率不会调得太多，但如果是SSAA，帧率会掉的十分严重。

- MASS:an improvement on performance，假设一个像素里有四个感知样本

- 对于同一个primitive（几何体）中，每个样本只做一次shading，如图，有四个sample，两个primitive。只做两次shading，第一次在三角形中的1号样本做shading，第二次在0,3,2号样本点找一个平均位置做shading。如果使用SSAA的话，此处需要做4次shading，所以MSAA提升了效率。



- MSAA允许进行sample　reuse，这个不是时间上的reuse而是空间上的。如图，在1和2两个像素内，在两个像素的连接处有两个采样点，这两个采样点既可以贡献给像素1也可以贡献给像素2，因此实际上等于通过reuse在6个采样点的情况下得到了8个采样点的结果，减少了采样点的数量，提升了效率。



Notes on Anti-Aliasing 02

我们记得有一种反走样是基于图像的，先渲染出有锯齿的图然后通过图像处理的方法将锯齿给提取出来并替换成没有锯齿的图，这种方法叫image based anti-aliasing solution。、

比较流行的方法叫SMAA(Enhanced subpixel morphological AA)

- History: FXAA ---> MLAA (Morphological AA) ---> SMAA

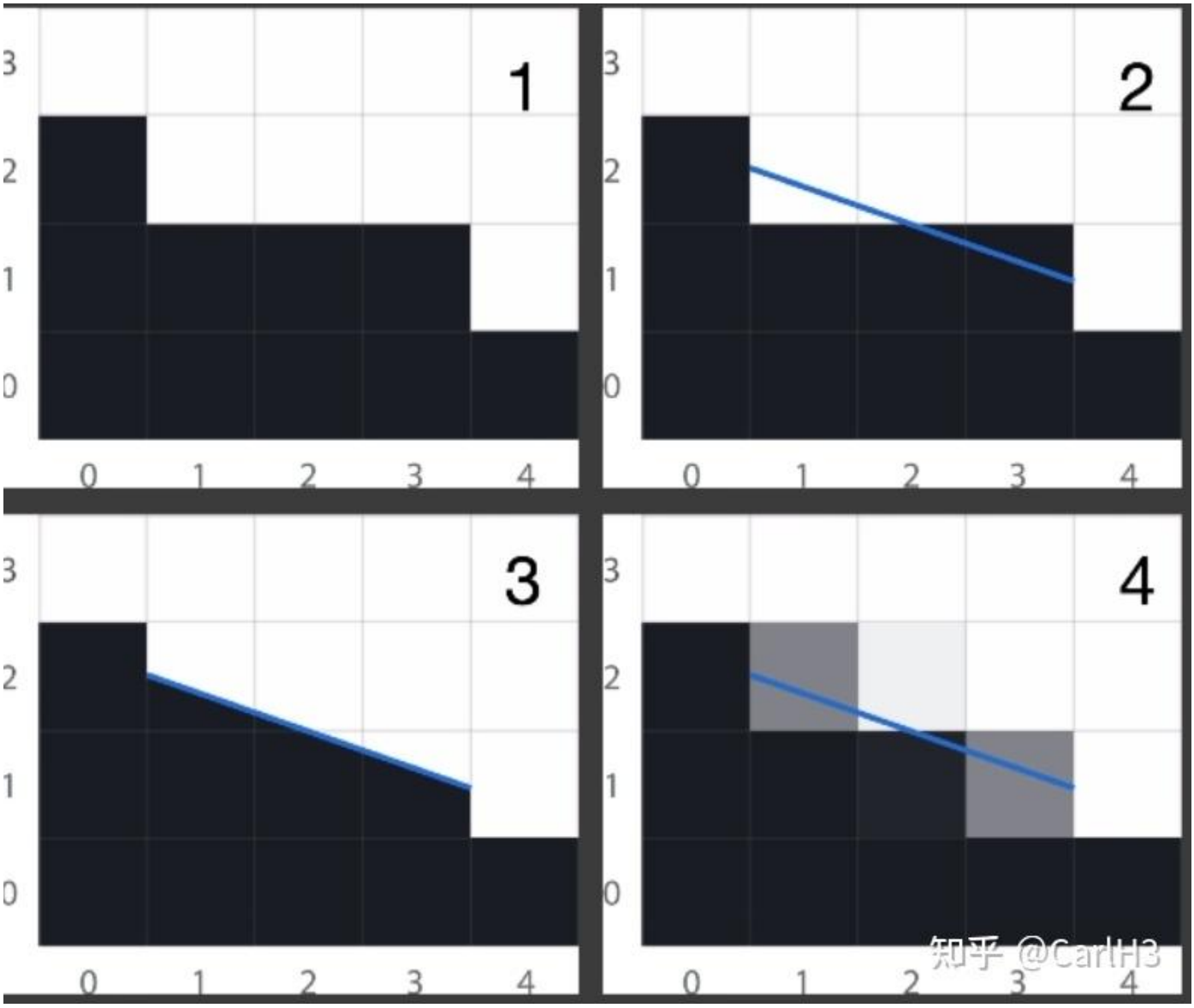


图1可以知道，这是有锯齿的，其想要的结果应该是一条斜着的直线，图像方法会先去识别它然后通过各种匹配的方法找到它正确结果的样子，图2是MLAA的方法，从而得到它应该变成的样子，然后通过其边界在各个像素内占的百分比进行shading。

Notes on Anti-Aliasing 03

- ***G-buffers should never be anti-aliased! G-buffers是绝对绝对不能反走样的！！！***

Temporal Super Resolution

- Super resolution (or super sampling) 低分辨率变成高分辨率
- 字面理解：为了增加图像的分辨率

DLSS就是这么一种技术，将一张低分辨率的图输入最后得到一张高分辨率的输出：

- **Source 1 (DLSS 1.0):out of nowhere / completely guessed**

DLSS1.0的思路是通过数据驱动的方法来做的，效果并不是很好。因为他只在当前帧中进行，不依靠temporal的累积，等于没有任何的额外信息来源，但我们将低分辨率硬拉成高分辨率，如果不想让最后的结果模糊，必须需要一些额外的信息，DLSS1.0是通过猜测来提供额外信息的。

也就是针对于每个游戏或者场景单独训练出一个神经网络，其会去学习一些常见的物体边缘从而在低分辨率拉成高分辨率之后将模糊的边缘换成不模糊的边缘，这就是DLSS1.0的思路。

- **Source 2 (DLSS 2.0):from temporal information**

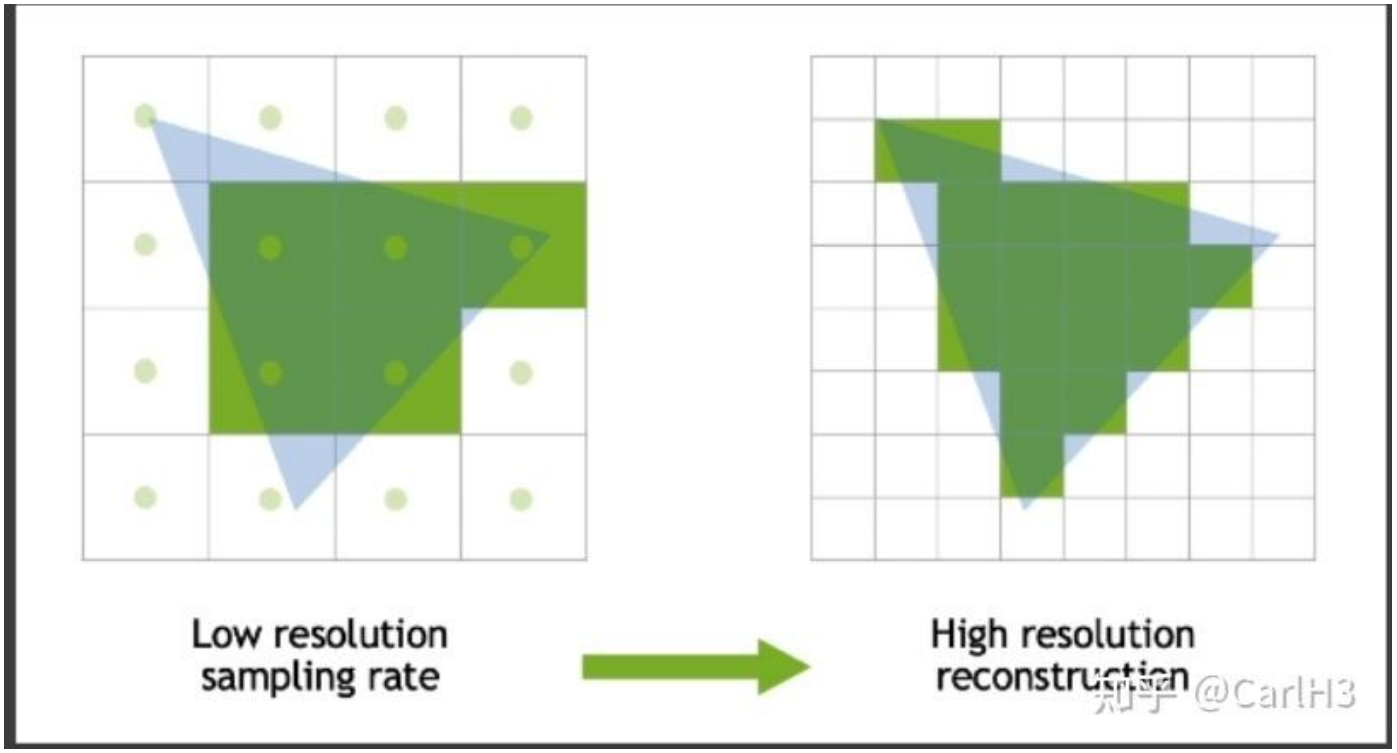
DLSS2.0则摒弃了通过神经网络猜测的结果，而是更希望去利用temporal的信息，核心思想不在DL上了而在temporal上。

DLSS2.0的核心思路在于TAA，更多的去结合上一帧的信息运用到当前帧中，仍然是temporal reuse。

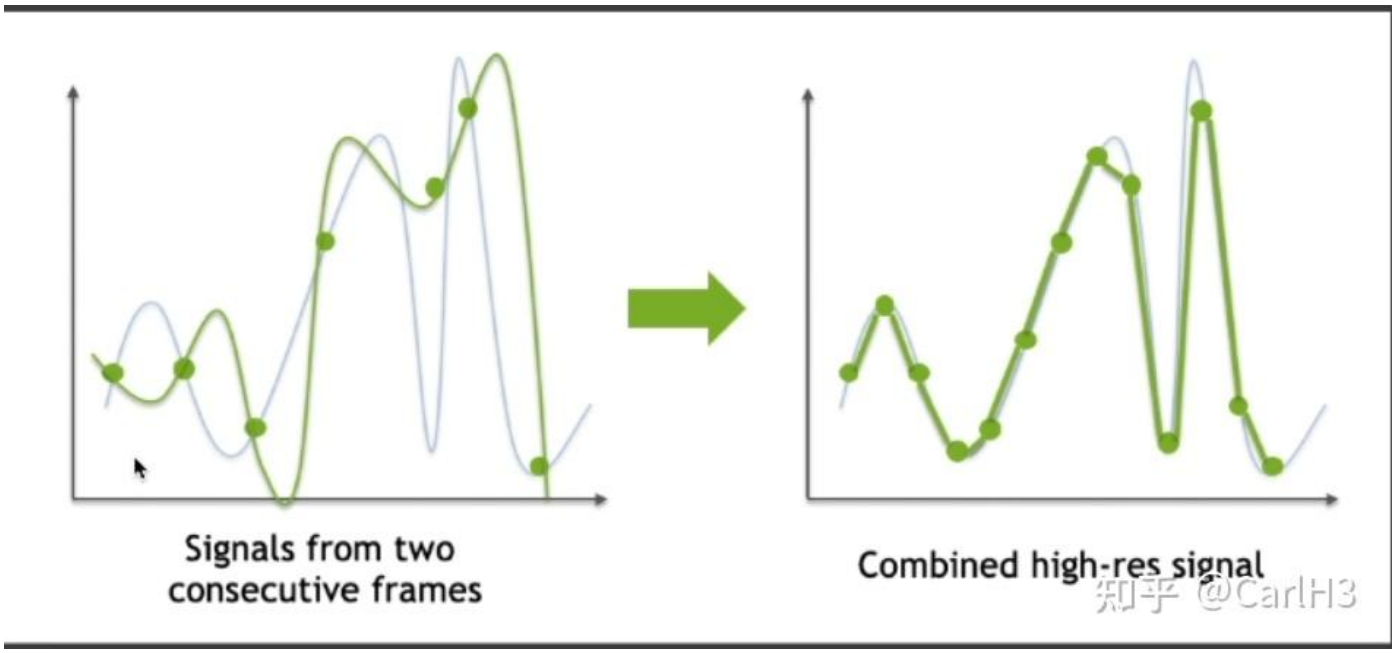
之前我们讲过，TAA对于静止场景中，连续四帧我们使用不同的感知sample点，当前帧使用上一帧的信息就等于是变相的提升了分辨率，这样想是对的，但是DLSS2.0中要面临一些问题。

1.如果有temporal failure时，我们不可以在使用clamping的方法来解决，也就是对temporal的信息利用要求更加严格。

2.因为我们最终要的是一个增大了分辨率的图，如图，分辨率提高也就是像素点增多，那么我们需要知道新增加的小的pixel的像素值是多少，如果此时我们用上一帧的结果盲目的clamping势必会因为一些小的像素的值是根据周围的点的颜色猜测出来的，而且猜测的值很像周围的点，也就是我们得到了一个高分辨率的图但是很糊。总结来说由于DLSS真正的提升了分辨率，因此我们要求新产生的像素的值是要与之前有本质的不同的，否则就会得到一个糊掉的结果。



3.因此我们需要一个比clamping更好的复用temporal信息的方案。



左边中的蓝色代表上一帧，绿色代表当前帧，绿点是当前帧给了一个采样信号得到的值，在上一帧也就是蓝色曲线中我们可以从另一个信号采样出来值，最后我们要把二者综合在一起得出一个当前帧增加了采样点后的值。

DLSS2.0中的神经网络没有输出任何混合后的颜色，而是去告诉你应该怎么将将上一帧找到的信息和当前帧结合在一起。

DLSS 2.0

- An importance practical issue
- 如果DLSS每一帧需要消耗30ms，那DLSS就太慢了，因此训练出这个网络之后去提升inference性能，针对nVidia的硬件进行优化，但具体如何做的……老师也没有权限了解。
- 其他公司的”DLSS“算法：
- By AMD:FidelityFX Super Resolution
- By Facebook:Neural Supersampling for Real-time Rendering [Xiao et al.]

Deferred Shading（延迟渲染）

Deferred Shading是一个节省shading时间的方法，是为了让shading变得更高效更快。

- 传统的光栅化过程：
- Triangles -> fragments -> depth test-> shade ->pixe

- 这可能会出现每一个fragment都需要shading的情况，比如你需要从远处到近处渲染时，需要将每一个fragment都进行shading，
- 此时的复杂度为：O(#fragment * #light) 因为每一个fragment都要考虑light
- Key observation
- 很多fragment在渲染过程中会通过depth test，但最终可能会因为被后续的fragment所覆盖而不会被看到，因此这些在过程中被shading的fragment浪费了很多的资源。
- 那么我们能不能只去渲染可见的fragment呢？、
- 解决思路：
- 将场景光栅化两次。Just rasterize the scene twice
- Pass 1: 在第一次光栅化中得到fragment之后我们不做shading，所有的fragment只对深度缓存（depth buffer）做一个更新。
- Pass 2：由于在Pass1中我们已经写好了depth buffer并且知道了最前深度，因此在pass2中只有深度等于最浅深度的fragment才可以通过depth test并进行shading，从而实现了只对visible fragment着色。

- Implicitly,this is assuming rasterizing the scene is way faster than shading all unseen fragments (usually true) ---->光栅化场景的开销是小于对所有fragment着色的开销的。
- 复杂度从O(#fragment * **#light) -----> O(#vis.frag, *** #light)

至此我们把shading延迟到了pass2中进行，并且把fragment的数量降低到了可见的fragment的数量。

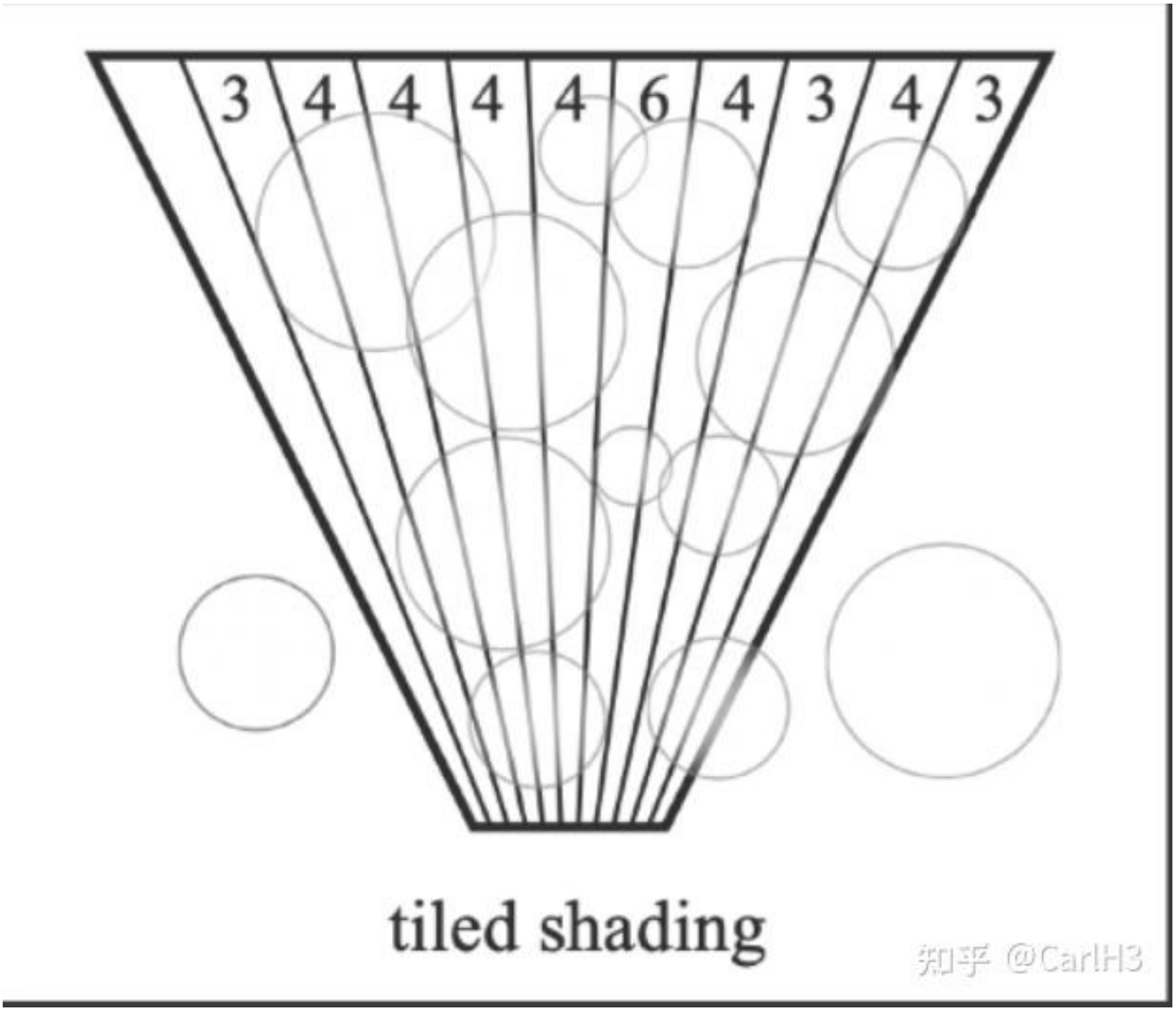
古尔丹，这次你想要什么？

- Issue
- 由于依赖于depth buffer(属于G-buffer),所以pass1和pass2都无法做anti-aliasing。
- 但是不是什么大问题，我们可以通过TAA或者是imaged based AA来解决。因此现在是工业界的主流操作。

我们对fragment已经优化了很多，既然而然为了更优方法，就将目光放在了light身上，从而引出Tiled Shading。

Tiled Shading

Tiled shading是建立在Deferred Shading的基础之上，将屏幕分成若干个小块，比如一个小块是32 * 32，然后对每个小块单独的做shading。



从camera看向屏幕的俯视平面图

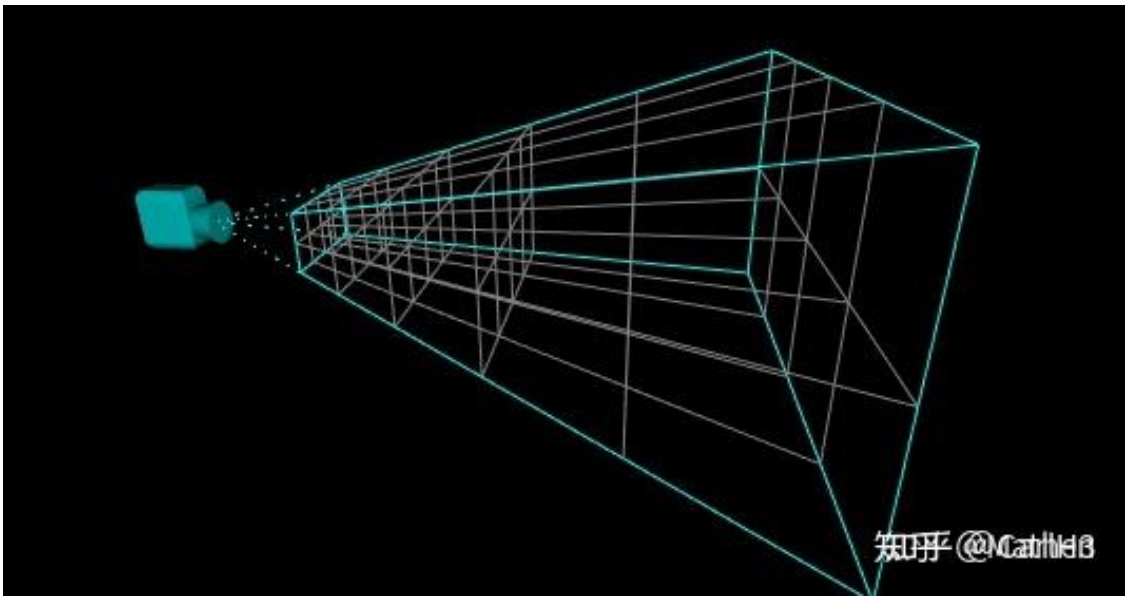
将屏幕分成了若干个小块，但由于是俯视因此是个平面图，看起来是分成了若干个小条，圆圈代表的是光源的范围，每个圆代表一个光源。

- Key observation

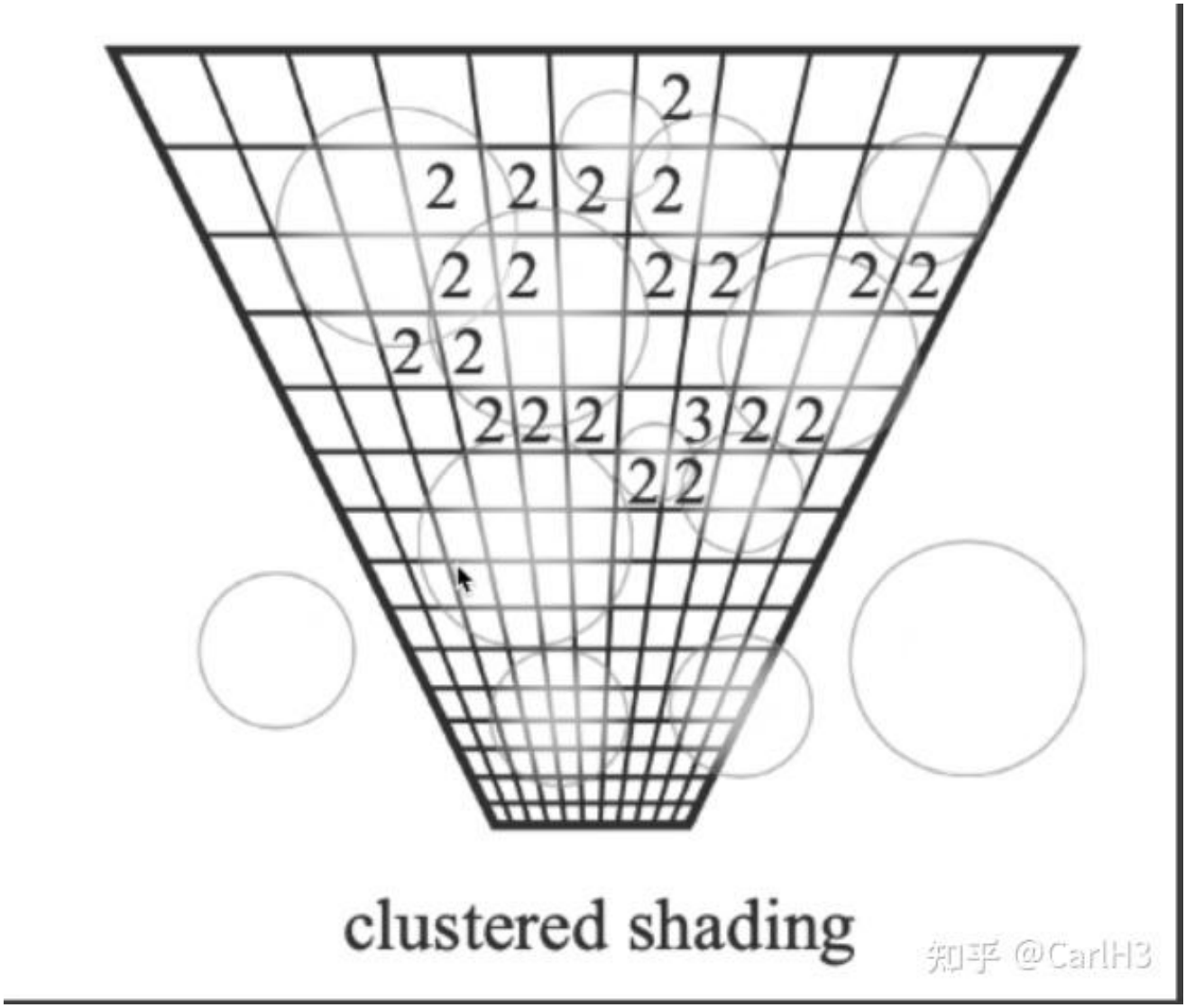
- 节省每个小块要考虑的light数量，每个切出来的小块代表场景中3D的区域，并不是所有的光源都会跟这片区域相交。
- 我们在做light sampling时知道光照强度会随着距离的平方衰减，因此面光源或点光源的覆盖范围是很小的，我们根据其随距离平方的减少找出最小值，也就是设定一个范围，光源的覆盖范围看做一个球形。因此我们在渲染时，只需要找会影响到区域的光源即可，不需要考虑所有的光源。
- 如图，数字代表区域内会影响到它的光源数量。
- 复杂度：O(#vis.frag. * **#light)->O(#vis.frag. *** avg light)
- 但在此之后，又有人对其进行了一个复杂的优化。

Clustered Shading

在刚才的基础上，我们不仅将其分成若干个小块，还要在深度对其进行切片，也就是我们将整个3D空间拆分成了若干个网格。



- Key observation
- 如果只分成若干个小块区域的话，区域内的深度可能会十分大
- 因此光源可能会对这个小块区域有贡献，但不一定会对根据深度细分后的网格有贡献。所以再划分过后我们可以发现每个网格内的光源数量更少了。
- **Complexity:O(#vis.frag. * avg #light per tile)->O(#vis.frag. * avg #light per cluster)**



这两部分讲了如何避免没有意义的shading从而提高效率。

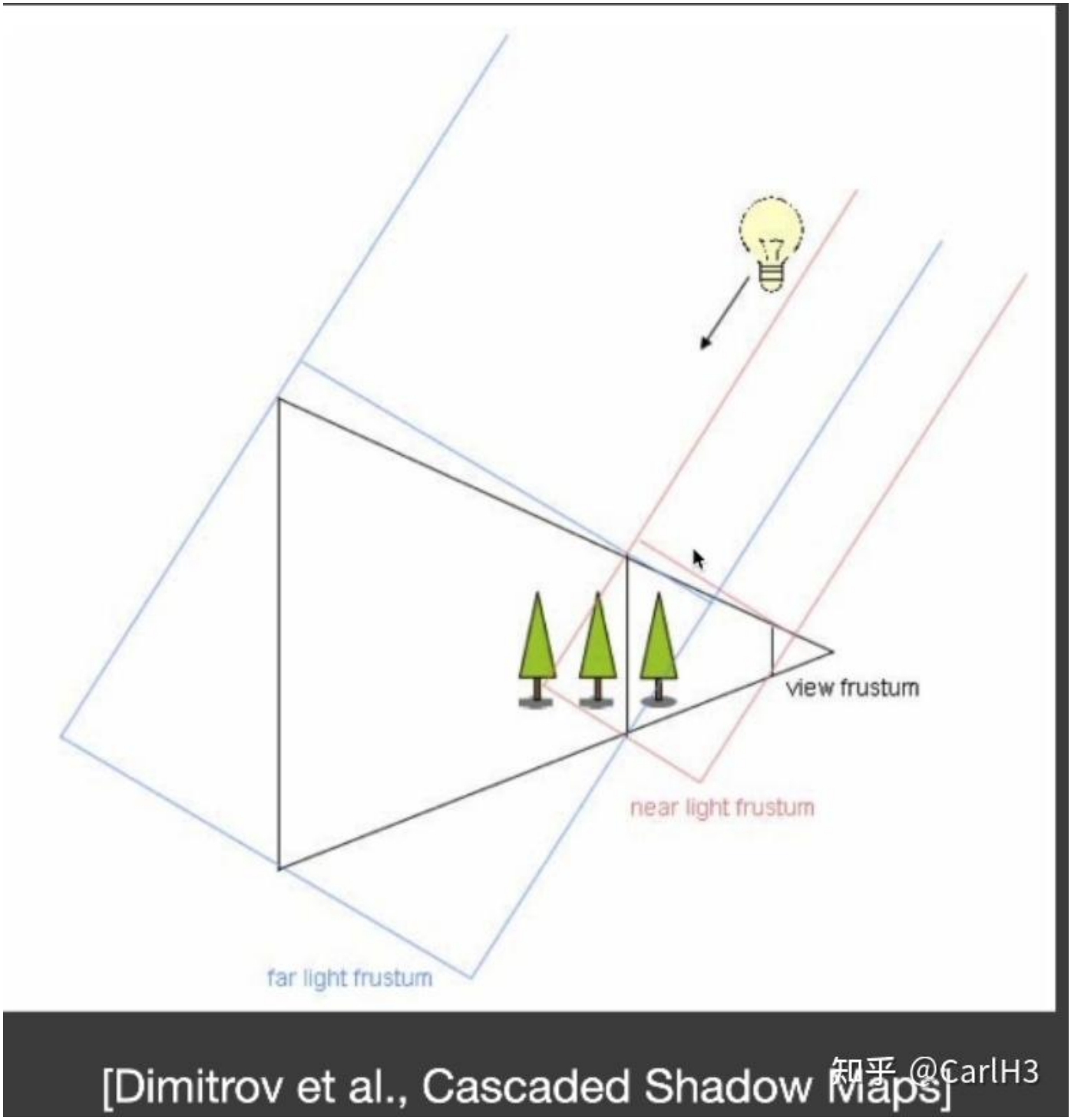
Level of Detail Solutions

- **Level of Detail (LoD) is very important**
- texture的mip-map就是一个LOD,在high level时detail少一点，但可以很快的计算出一个区域的平均
- LOD的核心思路就是为了在任何计算过程中能够快速准确的找出一个正确的level去进行各种运算。
- 在工业界中运用LOD方法或者在不同情况下选用不同层级的思路称之为”cascaded方法“。
- **Example**
- **Cascaded shadow maps**

我们知道在生成shadow map时我们需要给shadow map一个分辨率，当shadow map上的一个texel离camera越近，其在屏幕中所覆盖的内容越多，反之离camera越远，其所覆盖的内容越少，因此我们可以知道从camera出发看向场景，离camera远的我们可以用粗糙的shadow map。

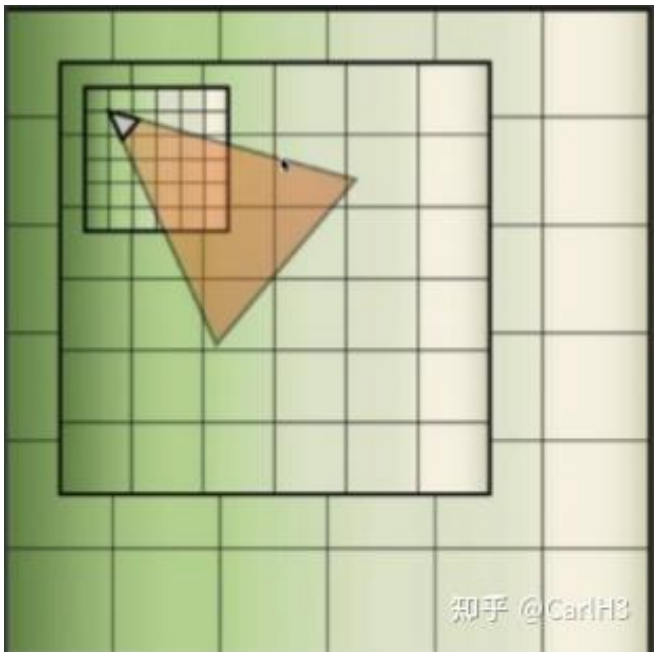
但我们是无法拥有一个会变化的shadow map的，因此在实际操作过程中，我们通常会生成两种或以上不同分辨率的shadow map进行使用的，如图，红色区域是一个高分辨率的shadow map，蓝色区域是覆盖范围更大但分辨率相同也就是更粗糙的shadow map.

从而我们根据物体在场景的位置来选用shadow map,远点的就用粗糙的，近处的就用精细的，但是从图中可以看到会有重叠区域内，因为在突然切换层级时会有一个突变的artifact,为了有一个平滑的过度，在这个区域内我们通过距离，也就是近处的以蓝色区域的shadow map为主将二者blend起来，从而产生平滑的过度。



- **Cascaded LPV**

同样的，先在细小的格子上，之后随着距离的增加到大一点的格子，再远就更大，从而省去了很多的计算量。



- **Key challenge**
- 在不同层级之间的过渡是一个难点
- 通过我们使用blending的方法实现手动平滑过度
- **Another example:geometric LoD**
- 如果有一个精细的有许多三角形的模型（高模），我们可以将其减化，减少其三角形使其变成一个低模，我们可以生成一系列的低模。

- 我们根据物体离camera的距离，来选择放什么层级的模型进去（高模还是低模），并且一个物体的不同部分也可以做不同的LOD，只需要提供一个标准，比如说这个标准是保证任何时候三角形都不会超过一个像素的大小。这样可以保证一个模型可以根据camera在哪，从而动态的告诉渲染管线需要渲染什么层级的模型或者部分。
- 但是这样会出现popping artifacts，因为transition是最困难的，如果camera推进，在推进过程中由于是突变的，会出现突然变化几何的现象，也就是popping artifacts，但是我们不用去管它，TAA可以很好的解决这个现象。
- This is Nanite in UE5(but of course,Nanite has way more)
- **FYI,some(strongly) technical difficulties**
- Different places with different levels,how about cracks?
- Dynamiclly load and schedule different levels. howto make the best use of cache and bandwidth,etc.?
- Representing geometry using triangles or geometry textures?
- Clipping and culling for faster performance?

Global Illumination Solutions

- From this course,we can see that
- Recall,when would screen space ray tracing (SSR) fail?

1. 出了屏幕
2. 反射物在camera背后
3. 由于是屏幕空间的几何表示，因此只用了depth表示了一层壳，在trace时如果在壳的后面是无法知道的。
4. 等各种情况。

没有任何的GI解决方法可以将所有的情况都给解决掉，除了RTRT（实时光线追踪），因为他理论上一个是一个正确的path tracing，肯定可以解决各种情况。

但是RTRT在现如今只适用于部分，如果整体都是用RTRT那么帧率自然而然会下降，因为RTRT太costly。

因此工业界经常使用hybrid solutions，也就是将许多中方法结合在一起。

- For example,a possible solution to GI may include
- 我们选做一遍SSR，得到一个近似的GI。
- 基于SSR上，我们对于SSR无法得到的结果，专用其他的ray tracing方法
- 用硬件 (RTRT) or software(?)去做tracing

- Software ray tracing
- **近处:在任意一个shading point，在他周围的一定范围内的物体我们用一个较高分辨率或称为较高质量的SDF，SDF可以让我们在shader中快速的tracing.**
- **远处：我们则用一个稍低质量的SDF将整个场景覆盖，至此不论近处还是远处我们都可以通过SDF得到最终光照打到的结果。**
- **如果场景中有强烈的方向光源或者点光源，也就是手电筒之类的光源，我们则通过RSM来解决。**
- 如果场景时diffuse的我们则通过DDGI来解决。Probes that stores irradiance in a 3D grid (Dynamic Diffuse GI, or DDGI)
- Hardware ray tracing
- **Doesn't have to use the original geometry,but low-poly proxies。**

我们没有必要去使用原始geometry，在RTRT中我们可以从任何一个shading point去trace整个场景，但是这样是没必要的，我们要算的是indirect illumination，没必要那么准确，因此我们可以用一些简化了的模型去代替原始模型从而加快trace的速度，这样RTRT就十分快了。

- Probes (RTXGI)，用RTRT的思路结合PROBE的思路，叫做RTXGI。

上述中加粗的四条结合在一起就是UE5的lumen思路。

It's the end of Games202,but it's not the end of my studying.

To be continued.....XD