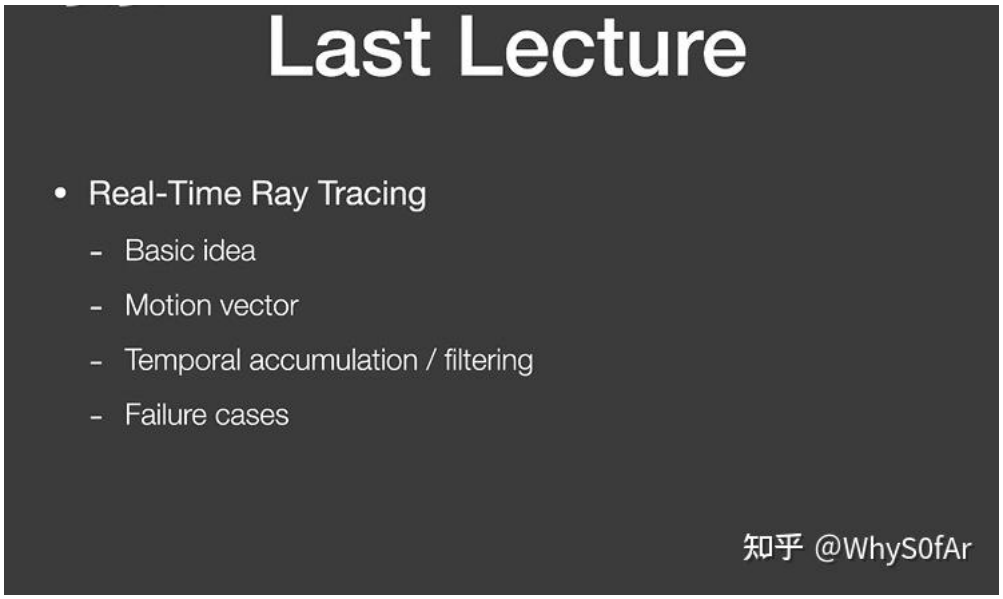
GAMES202 Lecture13：Real-Time Ray-Tracing 02

本文是闫令琪教授所教授的GAMES202 高质量实时渲染笔记GAMES202 Lecture13：Real-Time Ray-Tracing 02.本人属于新手上路暂无驾照，有错误欢迎各位大佬指正.

上节回顾:



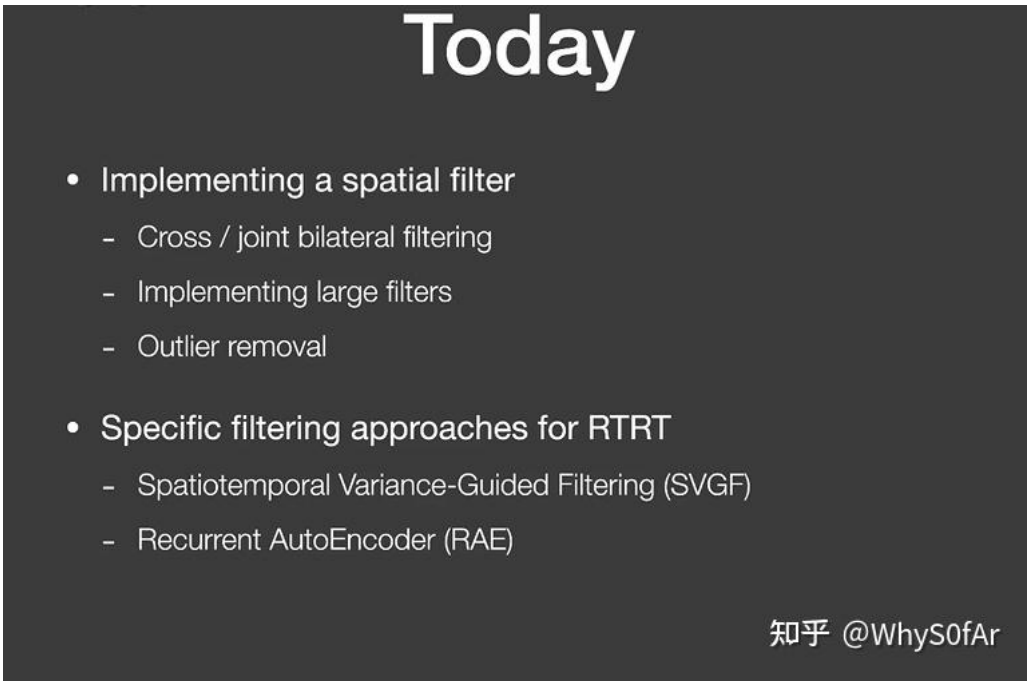
上节课我们知道了real-time ray tracing的核心思路是降噪,虽然RTX加强了硬件从而允许我们做RTRT,但是只是1spp的path tracing,他的结果是很noisy的,因此我们需要各种方法去增加spp从而得到更好的效果.

我们知道rtrt的核心是降噪,但是如果只依赖于当前帧的1spp结果去降噪的话结果仍然不是很好,因此我们通过temporal,也就是时间上的累积,通过motion vector我们找到当前帧的像素内包含的物体在上一帧的哪个像素上,从而将上一帧的结果和当前帧的结果线性blending起来得到当前帧的最后结果,但是时间上的复用也有

failure cases,比如:

- 突然切换场景或者光源会因为上一帧没有对应的信息从而出现错误的显示
- 对于倒退的场景无法在上一帧找到对应的像素.
- 出现残影/拖尾现象
- 场景不动的情况下 motion vector为0,如果shading改变会出现错误的显示
- 地板上反射出场景的情况来说,由于地板是不动的,其上面的每一个像素点的motion vector为0,当我们移动物体时,其地板需要一定的时间适应,之后再在地板上反射出当今场景中的物体,也就是反射滞后.

本节课内容:



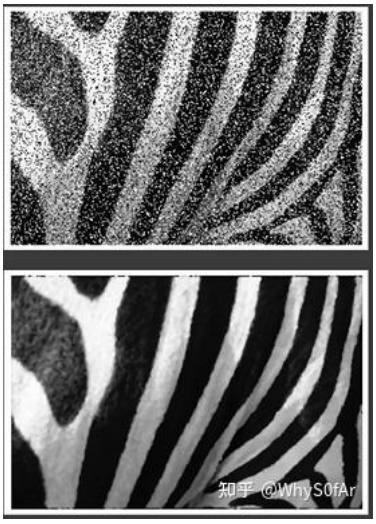
spatial filter和两个rtrt的具体实现

Implementation of filtering

Filtering

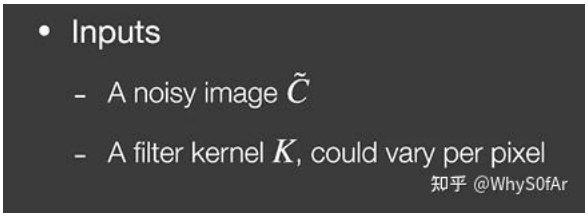
我们先从spatial filtering开始说,如何实现滤波和滤波到底做了什么?

首先我们来回忆一下滤波,滤波所做的概括起来就是做了一个模糊的操作,把一些噪声消除,也就是将一个比较noisy的图,降噪从而得到一个干净的图.如图



可以看到得到了一个比较干净的结果,我们在这里用的是低通滤波器,也就是只保留低频的信息,除掉高频的信息,随之而来的就有两个问题:

1. **高频信息中就全是噪声吗?那么高频中不是噪声的信息被除掉会不会导致信息丢失?**
2. **低频信息中就没有噪声吗?**

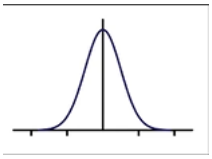


未处理过的图我们记做 $\tilde{C}$

滤波核记做K.

而处理过的图片我们记做 $\bar{C}$

一般来说我们为了给光线追踪由于蒙特卡洛产生的噪声降噪时,用的是高斯的滤波器,它的滤波核长这个样子:



类似于正态分布,中心值高,向两边衰减

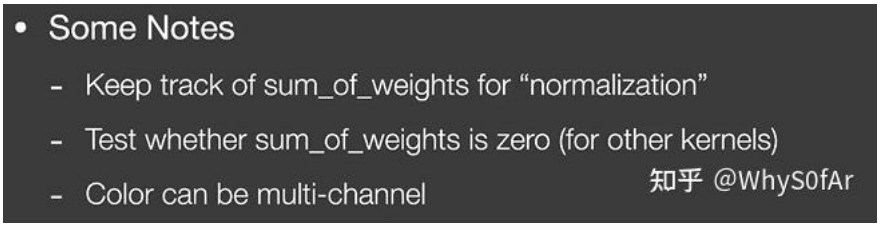
我们将中心像素成为i,其余的像素成为j,像素j肯定会贡献给i,具体贡献多少通过i和j之间的距离在高斯上找对应的值,从而知道j会贡献多少给i.

```
For each pixel i
    sum_of_weights = sum_of_weighted_values = 0.0
    For each pixel j around i
        Calculate the weight w_ij = G(|i - j|, sigma)
        sum_of_weighted_values += w_ij * C^{input}[j]
        sum_of_weights += w_ij
    C^{output}[I] = sum_of_weighted_values / sum_of_weights
```


伪代码

伪代码解析:

对于任何一个中心像素i
    我们需要定义 权值和(sum_of_weights),加权贡献值的和(sum_of_weighted_values)
    对于中心像素i周围一圈的任意像素j(包括像素i本身)
        我们需要根据像素i和像素j之间的距离和高斯的σ找对应的j贡献给i的值(权值)w_ij
        将权值w_ij与像素j对应的颜色rgb值相乘得到j的加权贡献值,并加到sum_of_weighted_values里
        将权值加到sum_of_weights
    进行归一化sum_of_weighted_values/sum_of_weights从而得到像素i最终的结果

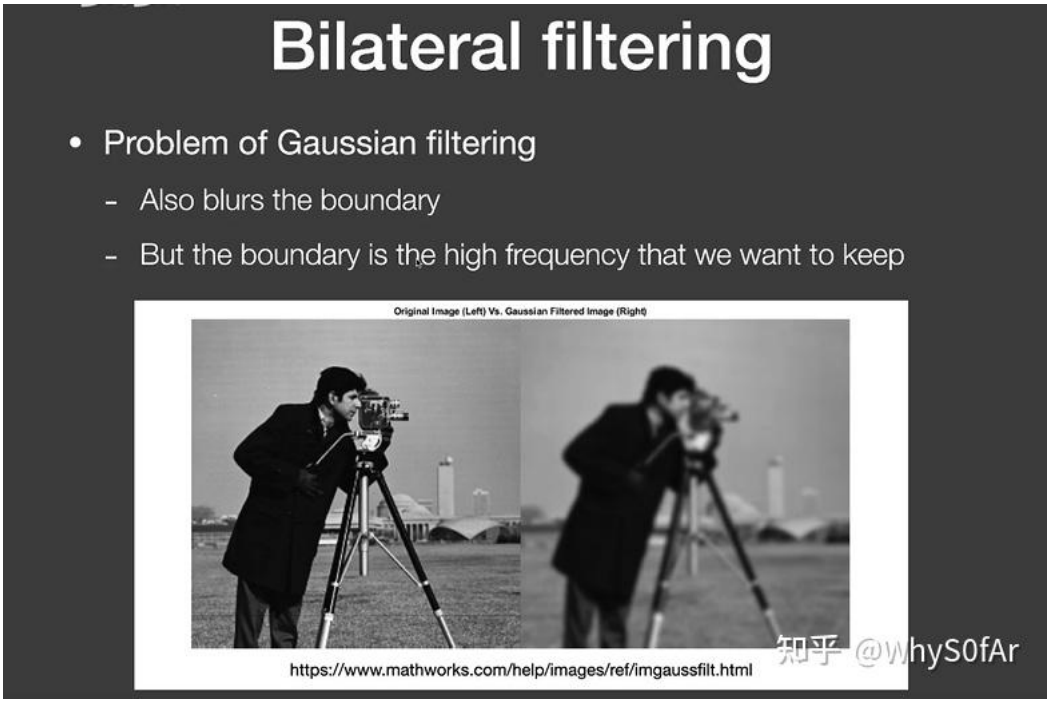


1.***不论是积分式还是求和式,平均写成sum_of_weighted_values/sum_of_weights是没问题的.*** 提到归一化我们在之前将渲染方程中的visibility拆出来后要对visibility进行一个积分并除以一个空积分来进行归一化处理,也就是分子是一个加权的visibility求和,分母则是权的求和,也就是sum_of_weighted_values/sum_of_weights.

2.***高斯的滤波核下,sum_of_weights不会为0,但在其他的滤波核下可能会为0,*** 因此在进行归一化从而得到像素i最终的结果之前通常会判断,sum_of_weights是否为0,如果为0则直接输出像素i的值为0.

3.像素j输入的值i可能是一个多通道的值,也就是sum_of_weighted_values可以是一个多通道的值.

Bilateral filtering



从图中我们可以看到,通过高斯我们得到了一个整体都被模糊的结果,但是我们想让边界仍然锐利,因此除了高斯我们需要其他的方法来帮助我们保留下边界的这些高频信息.

因此我们需要引入一个新的滤波核或者叫做滤波的方法,叫**双边滤波(Bilateral filtering).**

根据观察**:我们将颜色变化特别剧烈的地方认为是边界**

1. **那么我们如何保留边界的信息?**

由于我们认为颜色变化特别剧烈的地方认为是边界,如果二者差距不是特别大我们继续用高斯处理j到i的贡献.

但如果像素i和像素j之间的颜色值差距过大,我们认为这两个像素分别在边界的两边,从而让像素j给i的贡献变少,就不会出现边界的模糊情况,具体做法其实只需要在高斯的滤波核上加一些东西:

$$w(i,j,k,l) = \exp\left(-\frac{(i-k)^2+(j-l)^2}{2\sigma_d^2} - \frac{\|I(i,j)-I(k,l)\|^2}{2\sigma_r^2}\right)$$

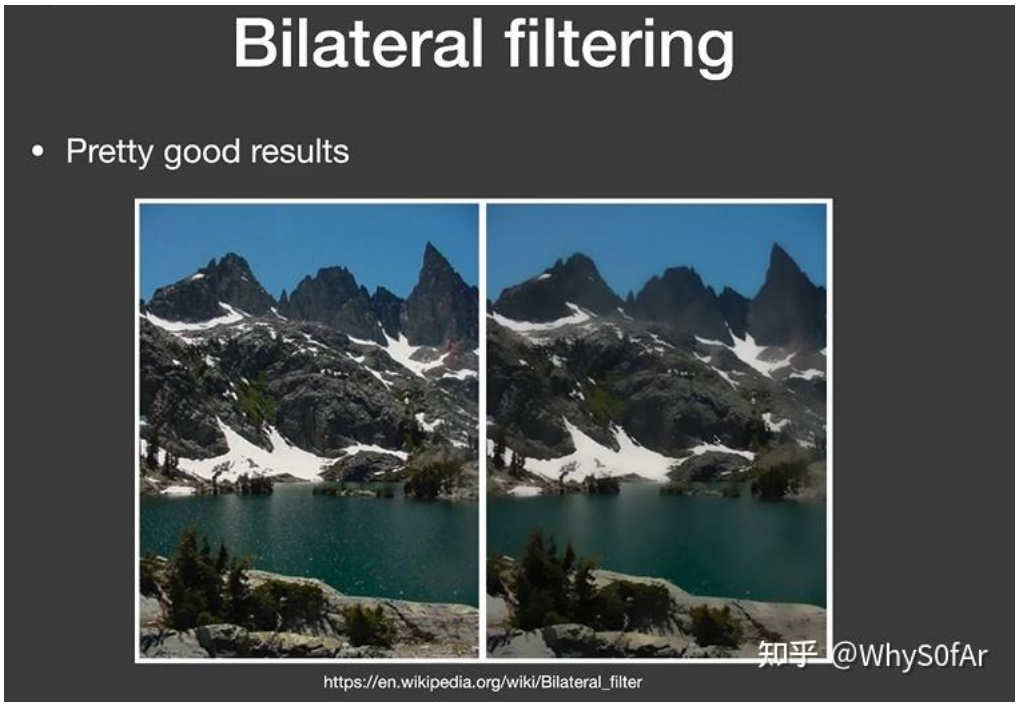
在这里i和j代表一个像素,k和l代表另一个像素,我们可以看到,这里仍旧是高斯滤波核:

$$\exp\left(-\frac{(i-k)^2+(j-l)^2}{2\sigma_d^2}\right)$$

我们想要的是如果二者之间的差距过大,则让j给i的贡献变小,因此增加的这部分:

$$-\frac{\|I(i,j)-I(k,l)\|^2}{2\sigma_r^2}\Big)$$

I(i,j)表示第一个像素的值,I(k,l)表示第二个像素的值,他们之间的差异就是分子部分,如果差异过大,就相当于在原本的高斯核上乘上了一个指数^-(距离)平方,那么距离差异越大,不就是使得其整体变小接近于0吗.



通过双边滤波我们对左边的原图进行处理,可以发现山上的细节和湖面的细节被很好的模糊了,但是边界仍完美的保留了下来,这就是我们想要得到的结果,保留了低频信息但也保留了边界.

但是当我们分不清噪声和边界的时候,该如何去进行处理?我们留到下文的svgf再讲.接下来我们讲联合双边滤波.

Joint Bilateral filtering

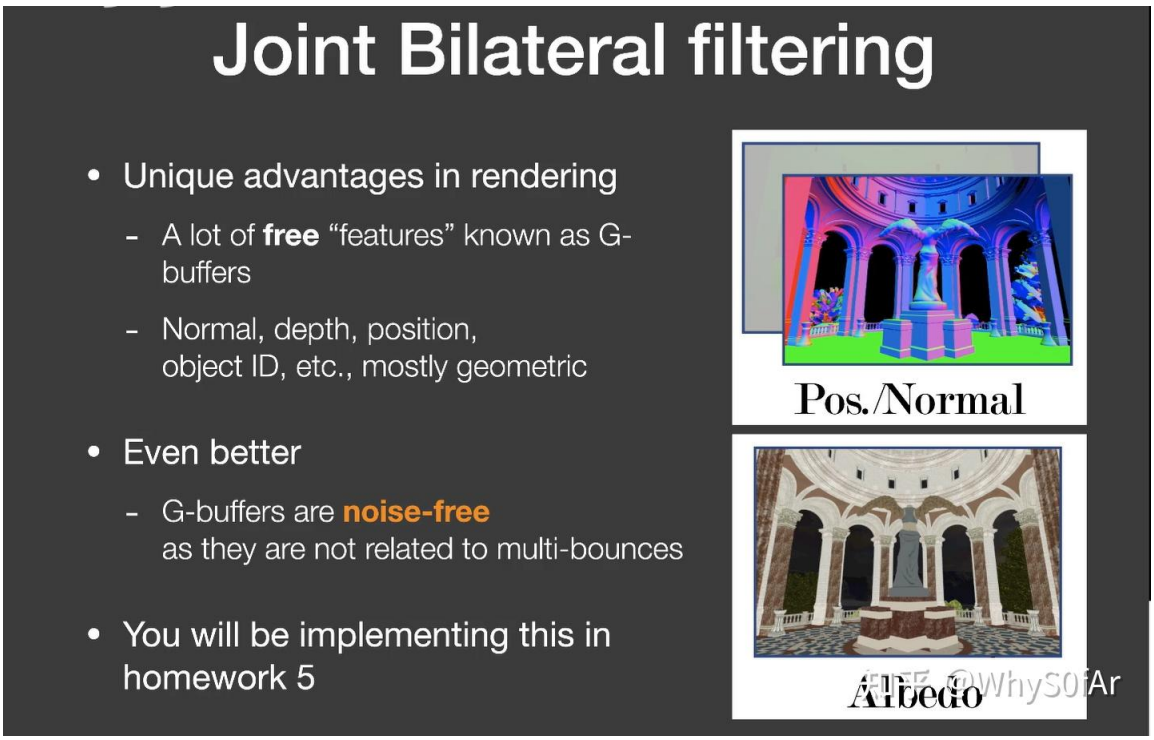
我们可以观察到

Gaussian filtering:1 metric (distance) 高斯滤波核是通过判断两个像素之间的绝对距离来查找其需要贡献多少

Bilateral filtering:2 metrics (position dist. & color dist.) 两个不同的标准，两个像素之间的距离，和颜色之间的距离从而在只保留低频信息时候保留了边界的信息.

Bilateral filtering相当于在原本的核上加了一个新的核,那么我们有没有可能利用更多的features来改变核,从而filtering得到更好的结果呢?

可以的,这就是Joint Bilateral filtering联合双边滤波,而且Joint Bilateral filtering在解决path tracing中由于蒙特卡洛所产生的噪声上有很不错的效果.



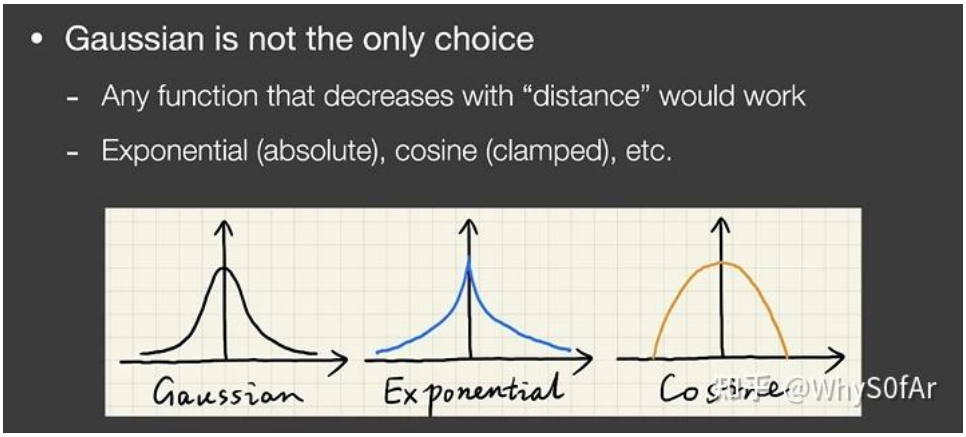
看到这两个图就知道与G-buffer有关了,我们知道在渲染时候可以得到很多免费的信息并存储在G-buffer中,比如**世界坐标 ,法线,深度等信息.** 这些信息都可以对滤波进行一个更好地指导.

更重要的是G-BUFFER有一个很好的特点:**完全没有噪声**

因为我们在进行rasterization(所有像素的primary ray)时候,可以顺手把一些信息记录下来存到G-BUFFER中,因为只是光栅化不涉及多次弹射,因此不会出现噪声.

some notes:

1. **联合双边滤波也不需要考虑核是不是normalize的,因为我们在代码中实现时最后会进行归一化操作.**
2. **我们不一定必须要使用高斯,只要这个函数满足随着距离衰减,就可以使用,如下图的几个例子.**

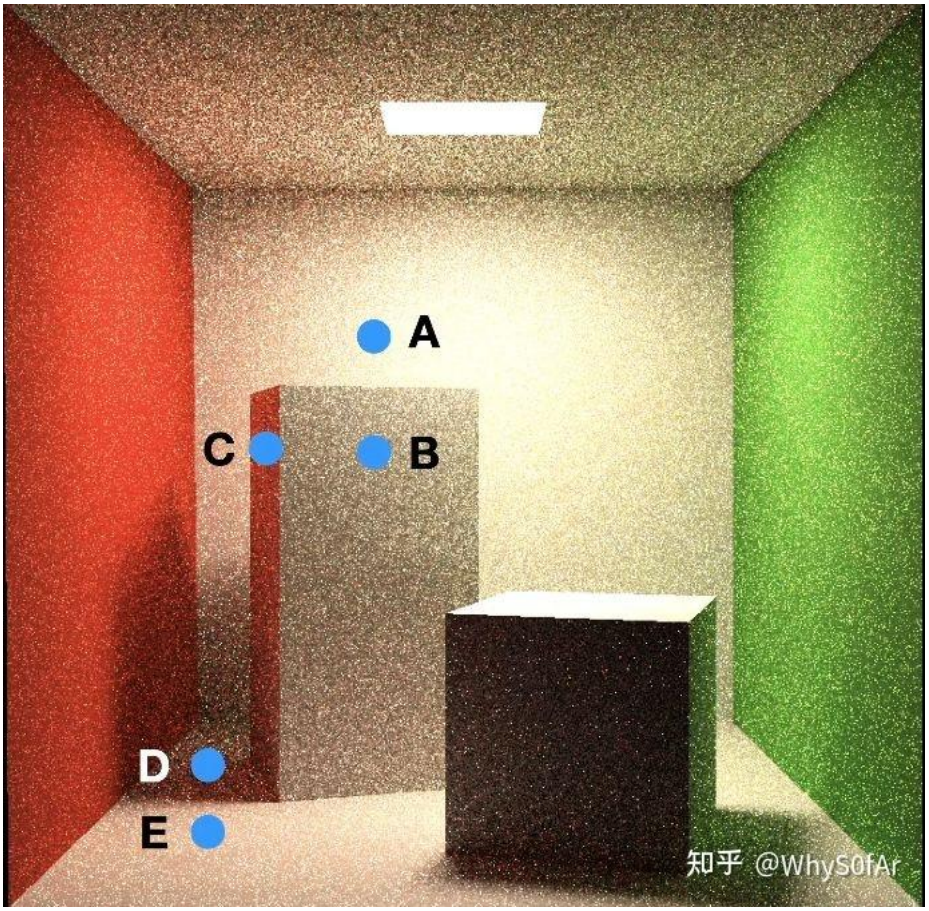


高斯 *指数函数* *cos函数*

具体如何使用这些features和G-buffer来指导我们进行滤波,我们看下面这个例子:

假设我们在g-buffer中知道三个信息:

1. 深度
2. 颜色
3. 法线



这是在RTRT中得到的结果,可以看到有很明显的噪声.

如果以A和B为例:

如果用高斯,我们是通过两个像素之间的绝对距离找对应贡献.

双边滤波的话在考虑距离的同时还会考虑两个之间的颜色变化,但是在这里双边滤波是不行的,我们可以看到A和B两个点都是十分noisy的,就算A和B没有在边界的两边,在同侧,他们之间的颜色差异也可能十分明显,这是双边滤波得到的结果是不准确的.

因此我们引入**深度** 来辅助我们做另外一个metric.

从A到B:

我们可以从G-buffer得到各自的深度信息,我们可以看到B的深度比较浅,而A的深度比较深,我们不希望A有太多的贡献到B上,因此我们把深度做成一个高斯或者任何随距离衰减的函数去使用.如果A和B之间的深度差异比较大,那么我们不希望A的值贡献很多到B上,

从B到C:

这里我们从G-buffer得到的是normal信息,因为我们可以看到B,C二者之间的深度是差不多的,因此不能继续使用深度这一feature了,如果此时我们不想让C点的红色部分贡献到B上面去,该怎么办呢?换句话说B和C在什么标准下他们之间的差异是比较大的呢?

Normal,我们发现他们的法线朝向是不一样的,也就是他们的法线夹角接近于90度,因为就认为他们之间的法线差异很大,因此使用normal的信息.因此我们把Nnormal做成一个高斯或者任何随距离衰减的函数去使用.(我个人认为应该是使用了cos函数)

至于从D到E:

就是很简单的颜色值了,就不再细说了.

联合双边滤波的本质就是在kernel里多算几组不同feature下的贡献,并将其相乘得到最后的结果.

Implementing Large filters

如何去实现比较大的滤波器,我们知道对于任何一个像素,我们需要考虑他周围N * N个像素的贡献值,加权平均之后写回这个像素上.

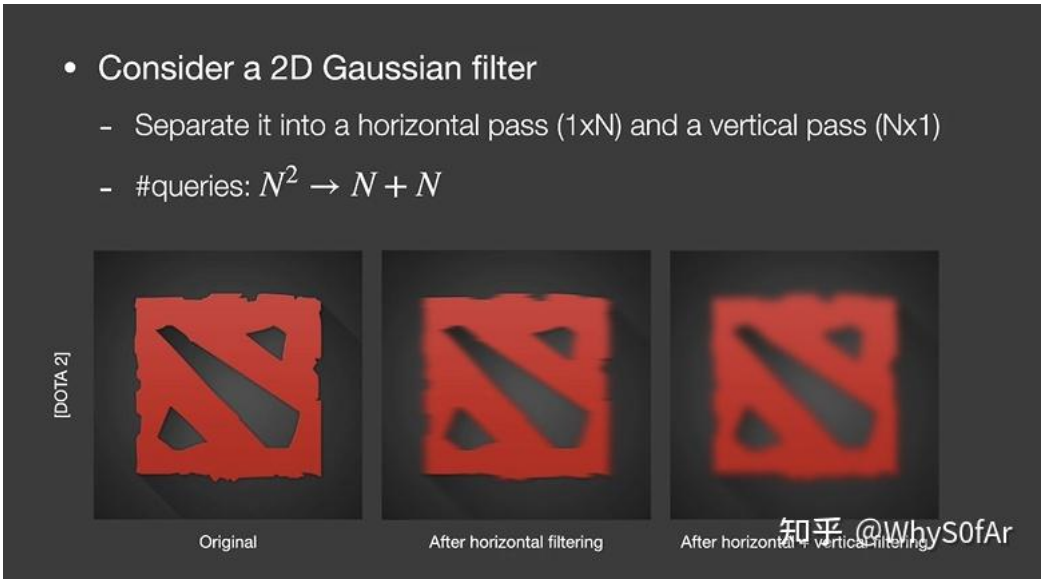
如果是small filter,也就是这个N * N比较小,比如是 7 * 7, 我们把范围内的像素贡献值加权平均值后写回这个像素,这样做是OK的.

但是如果是large filter,也就是如果这个N * N很大,比如说是64 * 64,就会使得filter变得特别慢,因此我们的首要任务就是让一个large filter的速度变快,那么该怎么去做呢?工业界有两种应用比较广泛的解决方法:

Solution 1: Separate Passes(拆分实现)

现在我们先考虑2D的高斯filter,先不考虑双边和联合双边滤波.

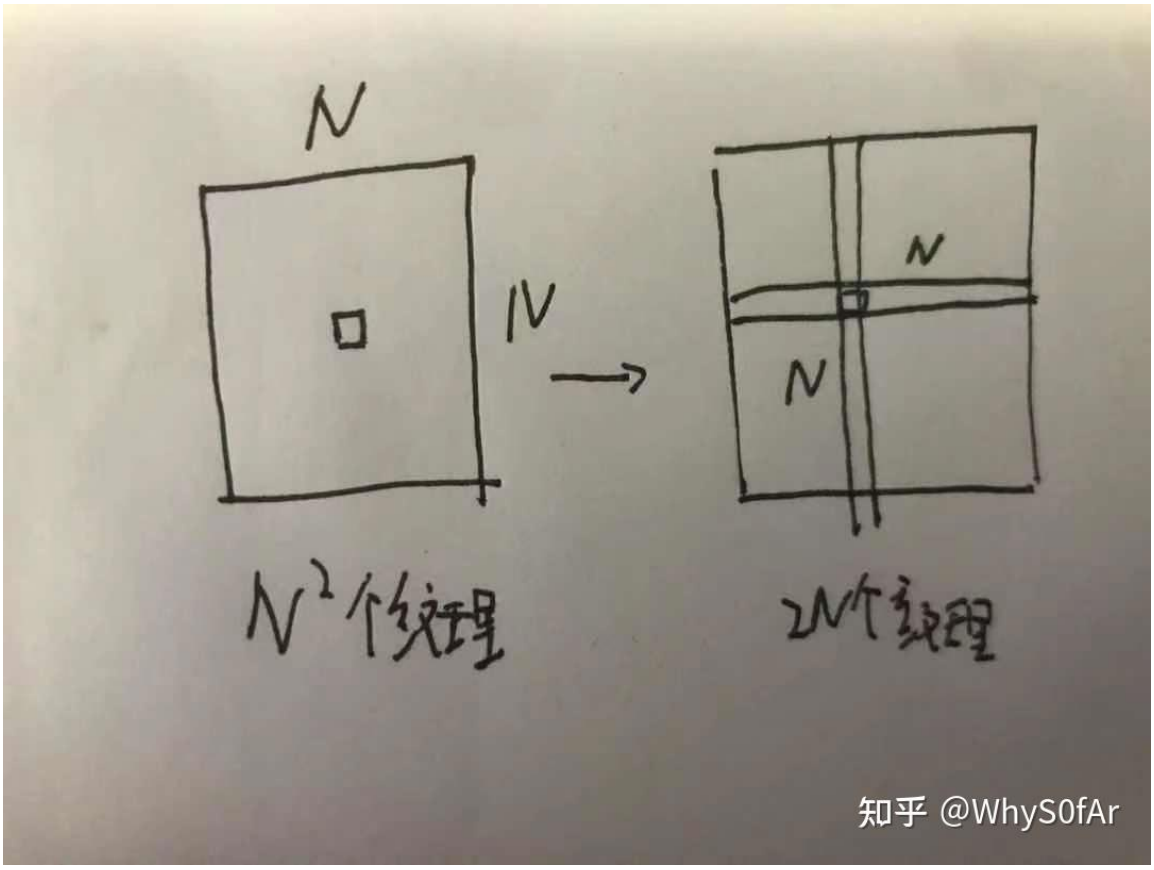
对于任何一个像素,我们其实可以不去做一个N * N的filter,假如现在有一张图,我们先将它在水平方向上filter一遍,之后再在竖直方向上filter一遍,也就是将本来一趟N * N的,分成两趟来做,水平的1 * N和竖直的 N * 1.



如图,我们现在原图的基础上,水平做 1 * N的filter,之后再在水平filter后的结果上竖直的做N * 1的filter,最后的结果相当于N * N filter的结果.

如果我们使用N * N的filter,对于任何一个像素我们需要访问N^2个纹理.

但如果我们将其拆分为两趟,第一趟访问N个纹理,第二趟访问N个纹理,总共访问了2N个纹理.



A deeper understanding

- **为什么我们可以把一个2D的高斯filter拆分为两个1D的高斯filter呢?**

- 因为2D的高斯函数有一个好的性质,在数学上就是拆开定义的,**因此其本身就是可拆分的**
- $$
  G{2d}(x,y)=G{1D}(x)\cdot G_{1D}(y)

  $$-Recall:filtering == convoluation(我们在games101中说过**滤波 ==$$
- 我们想要求一个像素周围一圈像素对自己的加权贡献,就相当于我们对2D函数F和高斯核在2D上进行一个卷积,由于2D高斯核可以拆分为两个1D的高斯核相乘,我们发现两个1D的高斯核分别与X和Y有关,因此当我们先做对X的积分时,Y是不参与的,所以把与Y相关的先拿出去,也就是先对X积分得到X积分后的结果,再在这个结果上对Y进行积分.

但是这种做法理论上只适用于高斯

对于双边滤波,两个高斯相乘,X和Y不容易拆分出来.

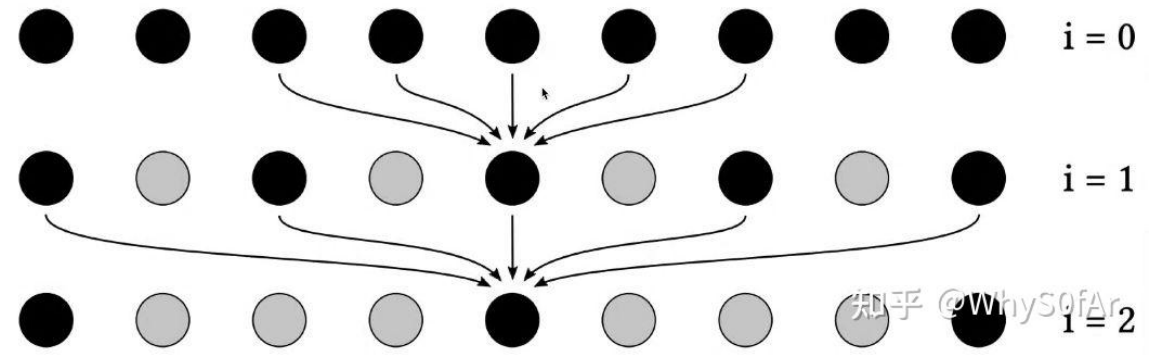
联合双边滤波(要考虑的东西更多,深度,normal各种信息,拆分成X和Y的函数也是很困难的,因为他们的滤波核本身就很复杂了

但是只是理论上,实际上我们还是可以强行去用的,只要范围不是太大,我们还是可以这样分成水平和竖直去做的.

Sol. 2: Progressively Growing Sizes

同样的目的,我们不希望对于任何一个像素进行N * N次的纹理查询,因此在方法2中,我们是用一个逐步增大的filter,比如先用一个小的filter,然后用中号的filter,最后是大号的filter,通过多趟的filter得到N*N的filter得到的结果.

- Specifically,a-trous wavelet
- Multiple passes,each is a 5x5 filter(多趟操作,每趟都是5 * 5大小的filter)
- The interval between samples is growing (2^i)(save e.g. $64^2 => 5^2 \times 5$)
- 对于任何一个像素在通过方法2进行filter时,我们通过多passes操作代替n*n的filter.我们认为第一趟是i=0,第二趟i=1依此类推,不同趟数的*5*5的filter中间是有间隔的,比如i=0时,每隔2^0,也就是1个间隔采样一个,i=1时,2^1间隔采样一个但filter大小仍旧是5*5的.依此类推,在第i趟时,样本之间的间隔是2^i,具体如图所示.



假设我们现在要做5趟,每趟都是5 * 5的filter,那么i在第五层时候,样本之间的间隔是 2^4 = 16,第五层一共要五个样本,也就是4个间隔,因此大小为64,也就是说在第五层时候我们相当于做了一个64 * 64的filter,但我们对于这个像素来说,我们做了五趟,每趟5 * 5大小的filter,一共做了 125次的纹理查询,对于 64 * 64 = 4096次这个数字来说是很小.

我们通过这种方法以一个像素很少的纹理查询得到了原N * N的filter得到的结果.

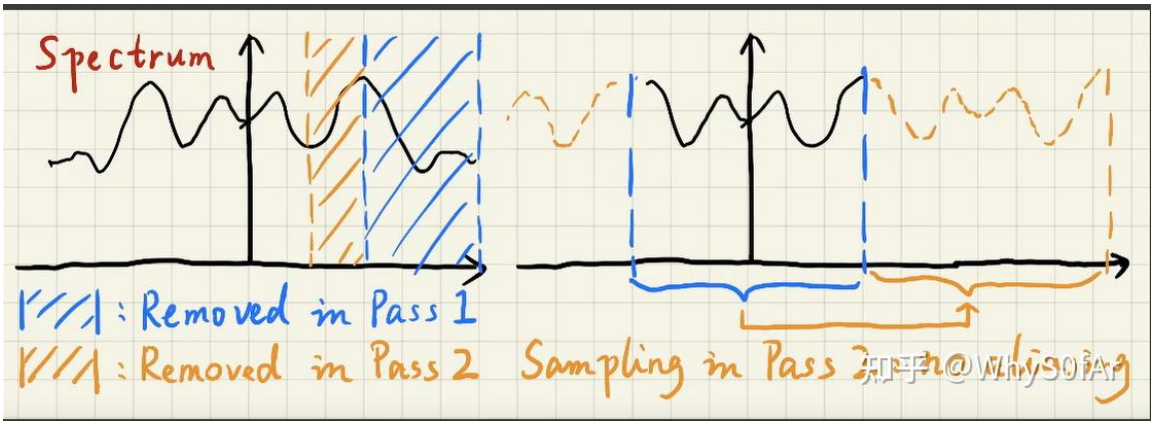
- **A deeper understanding**

1.为什么要逐渐增大filter,而不是一上来就使用第五趟中间间隔16个样本的filter?

- **用更大的filter == 除掉低频信息**

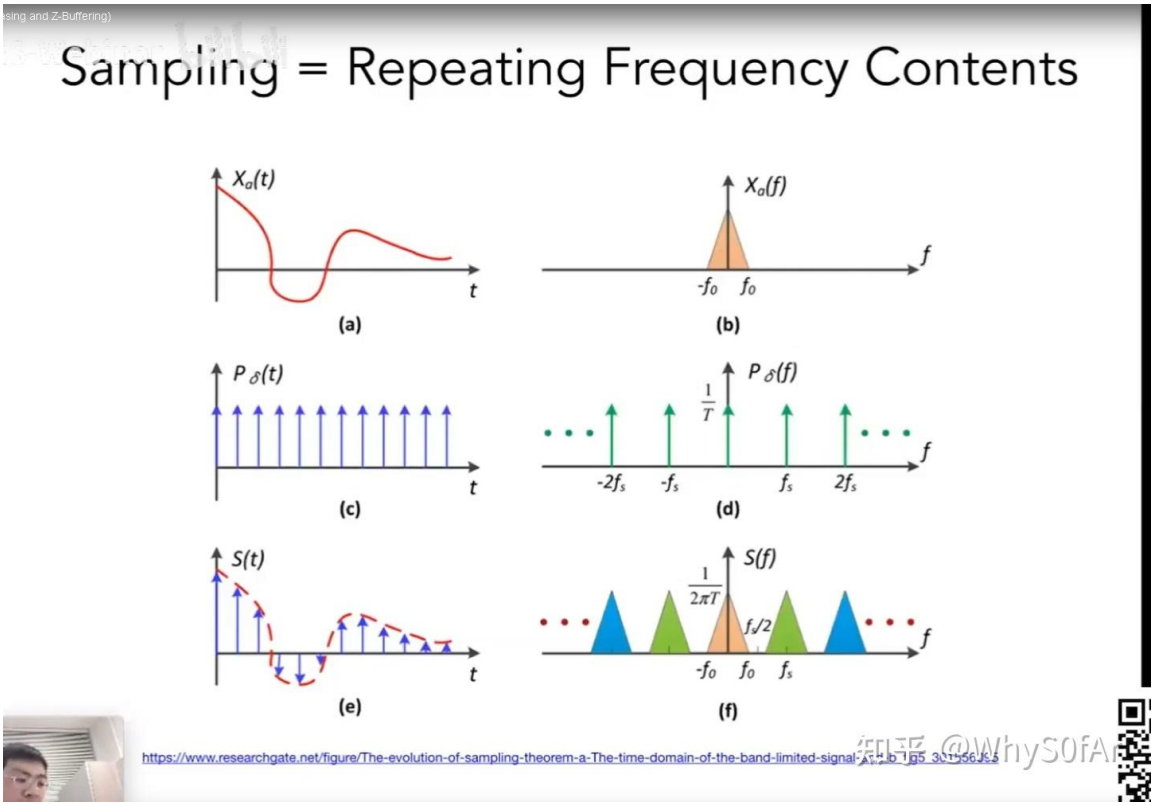
2.为什么可以间隔那么多的样本?

- **采样 = 重复的搬移频谱**



1.用更大的filter == 除掉低频信息我们可以看到,在第一趟时候,我们通过一个小范围的filter,将频谱上的高频信息,也就是蓝色区域除掉了,依次类推直到最后一趟除掉最低频上的信息.

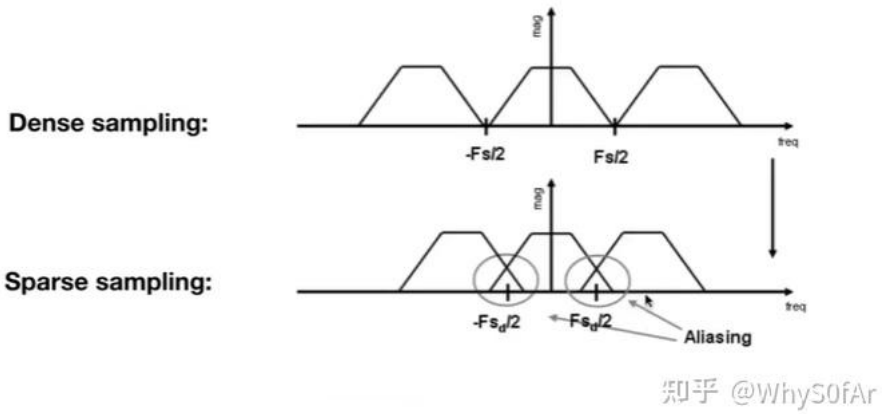
2.采样 = 重复的搬移频谱,第一趟filter我们将左图的高频信息也就是蓝色区域除掉之后,我们剩下的就是右边蓝色段的信息.在第二趟pass时候,我们相当于在9 * 9的filter里做了一个采样,采样出5*5的filter.



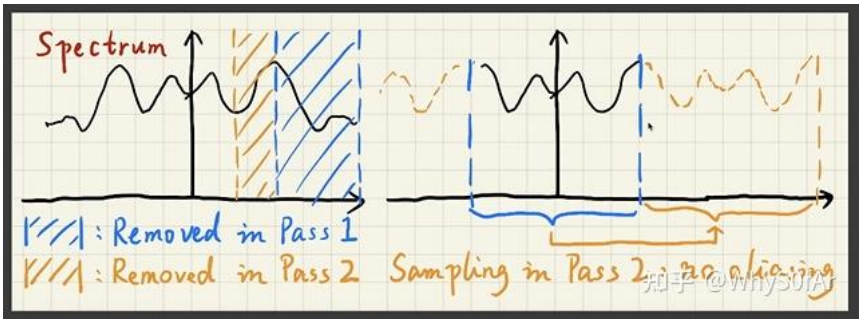
我们来复习一下,a是时域上的,如果我们想对a进行采样,也就是乘上一个冲激函数,从而得到了一系列离散的点,这就是采样.

通过傅里叶变换我们知道,时域上的乘积 = 频域上的卷积,b是频域上a的频谱,d则是频域上的冲激函数,他仍是一个冲激函数只是间隔不同,那么b和d做一个卷积我们得到了f,我们可以发现采样实际上就是在对频谱以一定的间隔做一个重复的搬移.

那么之所以会出现走样现象是因为当我们的采样比较稀疏,也就是采样点之间的间隔比较大,我们知道时域和频域上很多关系是相反的,因此采样点间隔大就意味着频谱之间的间隔小,因此会产生频谱重叠的现象从而发生走样.



回到202讲的东西



我们在第一趟pass中出去了高频蓝色区域,然后在第二趟pass中相当于在9 * 9的filter中采样出 5*5的filter,我们采样的间隔对应到频谱上正好是右半图的蓝色部分,也就是在第一趟除掉高频信息后的区域的2倍(tips:频谱是左右对称的),正好是尾对着下一个频谱的首,避免了走样现象.

Outlier Removal

平常我们在是用蒙特卡洛方法渲染一张图时,得到的结果会出现一些点过亮或者过暗,这些过亮或者过暗的点如果经过使用filter这种降噪的方法去处理的话不好处理,以*7*7*的filter为例,如果有一个点过亮,在经过这个*7*7的filter处理过后,会影响一块区域,使得它这一块区域变亮.

那么我们是否可以在filter之前处理掉这些过亮或者过暗的点?而且我们如何去定义这个outlier也就是这个边界呢?

Outlier Detection and Clamping

1.Detection

outlier不是就非主流的颜色吗,那么我们首先知道它主流的颜色的范围,因此对于每个像素,我们都取他们周围一个小范围的区域,例如7*7*或者*5*5,然后把这个范围内颜色的均值(mean)和方差(variance)给算出来,Value outside$[\mu - k\sigma, \mu + k\sigma]$->outlier,这句话的意思是,我们认为正常的范围在均值+-若干方差内,超过这个范围的我们认为他是outlier,这样我们就判断出了哪些是outlier的.

2.Clamping(outlier removal)

如果在范围内我们找到了outlier的点,我们把这个点的值给clamp到接近范围的值.

假设我们认为(-0.1,0.1)是有效范围,现在我们有一个outlier的值是10,我们需要把10变成0.1另一个值是-0.3,我们则需要把他变成-0.1,也就是clamp到正常范围内的值.

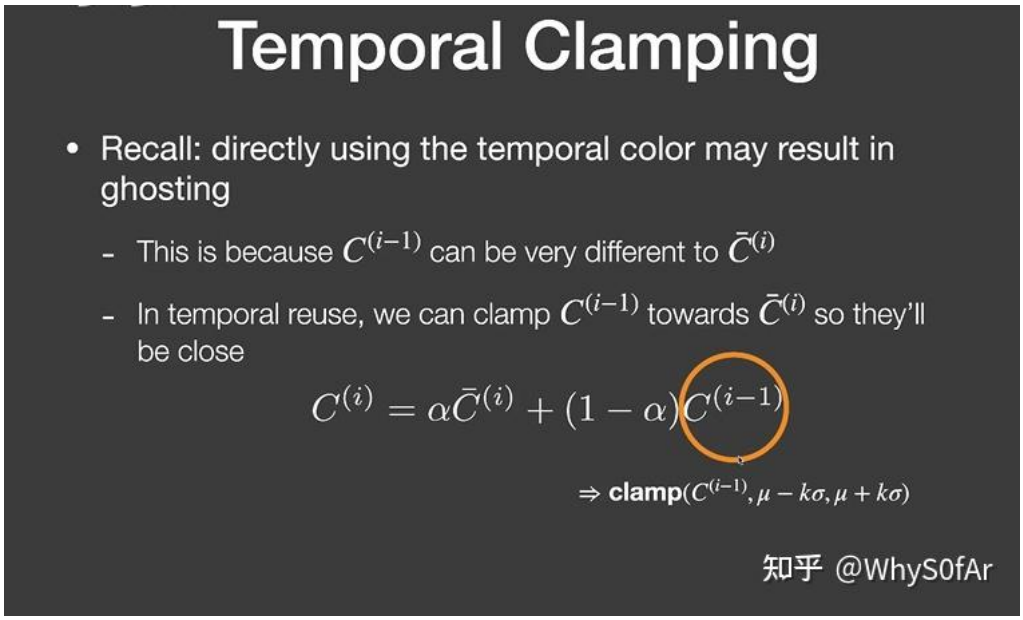
由于叫做outlier removal,所以从名字上很容易认为是把outlier的值给舍弃,其实不是的,而且这部分操作是在filter之前操作的,从而能得到正确的filter结果.

Temporal Clamping

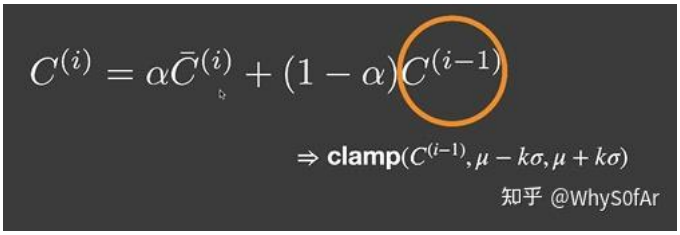
$\tilde{C}$:没有filter过的帧

$\bar{C}$:filter过的帧

C:noisy-free,也就是已经经历过temporal累积后的帧



我们在上节课中讲到,当motion vector为0导致的残影现象,当前帧与上一帧中对应的信息差异过大时,我们把上一帧中的信息值给clamp到接近当前帧的信息值,



上一帧的结果肯定是noisy-free的,这一帧的结果是filter过的,那么我们将上一帧对应点的结果clamp到这一帧对应点周围的有效范围的内内的操作和outlier removal是一样的,当前帧在spatial filter之后,我们在对应点周围找一个很小的范围,找出他们均值和方差,仍然认为 $[\mu - k\sigma, \mu + k\sigma]$是有效范围,如果上一帧对应的值超出这个范围,则把他clamp到范围内,再和当前帧做线性blending,从而得到当前帧noisy-free的结果

TIPS:

temporal clamping不是一个解决方法,而是一个介于noisy和残影的tradeoff.

我们是将上一帧的不可信的结果值给clamp到当前帧filter后的正常范围内.