常用渲染优化手段介绍。

计算机图形学，号称程序员的三大浪漫之一，发展了几十年。其中，各种算法不断被发明，然后不断被淘汰。其中，就包括了非常多的渲染优化手段。

本文介绍一些常用的渲染优化手段。介绍的时候，遵循这样一些原则：

1. 排名不分先后，想到哪写到哪。有忘记的说不定以后再补充。
2. 已经完全淘汰了的，不再可能使用了的，不介绍了。这种情况，一般是有更优的替换方式了。
3. 我会对每一种优化方案提出我自己的看法。我的看法不代表正确，只能代表我自己。但是，我每一个看法，都是谨慎的，并且会详细阐述我的理由。我会把这些方案分三类：好用、鸡肋、不好用。
4. 这里不会太详细的讲解每一种方案，因为很多方案，重要的，我可能会在其他章节讲过，例如延迟渲染。如果每一个细节算法这里都讲，我估计要累死，也没什么必要。有兴趣的，自己再去详细查找该算法的具体实现完事。

好了，言归正传，我们先讲一些非常大众的方案。

1. 视锥体裁切。

这个方案，应该是最最最最常用的优化方案了。如果仅仅是提这样一个大方向，其实你会发现，这不仅仅是一个大方向：围绕视锥体做优化的渲染方案，估计能找出来一百几十种。如果熟悉光栅化原理，其实很容易理解这个方案。如果是单纯讲大方向的视锥体优化，以下方案都跟视锥体裁切相关：镜头裁切，背面剔除，四叉树、八叉树、LOD、tessellation、延迟渲染、Forward+……。这一大堆，基本都能跟视锥体挂一下钩。

但是，我不能这么无耻，我们还是一个一个来讲，分开一些，尽量详细一些。所以，这里第一个讲的，就是镜头裁切。

镜头裁切的原理很简单：假设一个镜头在场景中漫游，场景很大，但是镜头看到的东西，可能仅仅只有场景的百分之一、千分之一甚至万分之一，这个时候，整个场景全部渲染，那是非常傻逼的行为。典型的做法是：先把所有Instance（实例）跟镜头做一个裁切，在镜头内再渲染，不在镜头内，渲染了也是白搭。这个裁切，一般是用AABB跟视锥体做一个相交计算即可，各大引擎都有通用的计算方式。

早期时候，大家技术都不咋的，这个部分做出来一些不合时宜的东西。典型的例如：镜头旋转的时候，从一个空旷的荒野，转到一个热闹的集市，会出现渲染帧率从极快到极慢的转变，非常影响游戏体验。这个就等于一边渲染几个mesh，转一下渲染几千个，突然的帧率下降导致体验差。现在基本都有限帧，不会出现这类东东了，仅仅怀念一下，

这个技术，属于“好用”级别，非常常见，必须。

1. 四叉树。

四叉树属于地形渲染必备方案。我刚开始搞3d的时候，做地形，直接用的3dmax，做plane。那会主要做虚拟现实，不属于游戏，主要就一个漫游，那还是2006年。我那会都不知道地形是用四叉树来做的。

用四叉树做地形，有什么优势？很简单：一块豆腐，两刀切四块，切成“田”字的方式，然后每一个小块，再细分……这么做的好处主要有两点：1、可以大块做镜头裁切，判断是否在镜头内。判断大块在了，再递归判断小块，大大提升了裁切的效率。2、地形距离人物太远的话，可以只渲染大块，距离近了，再渲染小块。这个，实际上属于地形LOD了。

四叉树地形优化，有明显的优点，没有明显的缺点，属于必备技术。这里，也不打算详细讲了，也没什么好讲的，烂大街了。

1. 八叉树。

八叉树是四叉树的延申。大概这样理解一下：两刀把一个豆腐切成4块（田字）切，三刀切成八块，那就是八叉树。

那么问题来了，八叉树有什么好处？

1. 、裁切（包括但不限于镜头裁切）。
2. 、场景管理。这个好处就多了，因为场景是需要交互的。例如你走到某个地方，可能被某个NPC看到了，使用八叉树，就能快速实现。例如你做一个FPS游戏，你一枪打过去，打到了哪个小块，只需要遍历哪个小块的Mesh做相交计算即可。

问题来了，八叉树看起来也是挺叼的，是不是属于必推方案？我不这么看，八叉树看起来叼，有其他一些让人不爽的地方。例如，很容易出现一个人同时在A块，又在B块这种情况，甚至一个人同时在好几块，会增加计算开销。但是四叉树不会出现这个情况，只处理地形，相对简单得多。

所以，在我看来，八叉树属于一个可选方案，使用情况根据实际情况而定，这个并非万能方案，只要做游戏了就必用，没到那地步。

1. LOD。这个全称叫Level Of Detail。其实是一个197x年代的老算法了。但是，各种优化，实现方式层出不穷，让这个方案宝刀不老。典型的例如tessellation，贴图mipmap。这个我认为也就是一种LOD。LOD的精粹，其实就是看得清楚的地方，用精细模型（贴图），看得不清晰的地方，例如比较远的地方，用粗略模型（贴图）。

LOD的实现方式非常多，抛开默认的方式，最垃圾的方式就是距离远了卸载精度模型和贴图，加载粗模和贴图。近了就反过来。这种最垃圾的方案，我在2006年的时候自己写过一遍。当初在水晶石做一个大项目，根本跑不动，当初最牛逼的显卡，也就512M显存，能跑多大的场景显而易见。那会我也是个菜逼，也不会其他更好的方式，引擎貌似也不支持多牛逼的方式，就用了这个。使用了四套模型，分级加载。那会镜头跑得很慢，能看到模型一点点的改变。但是这个项目后来过了，原因无他，那会好像大家都很菜，其他人，其他公司也压根没什么好办法。我当初为了实现这个，还自己写了个多线程加载，渲染的时候还不卡顿，还弄了个内存池，当初还沾沾自喜。没办法，这就是菜逼的世界。

平心而论，LOD是个好方案，但是使用的时候，一定要谨慎，不然即使是很牛逼的大佬，也可能做成负优化。最近比较有感触的就是UE4.26的一个优化。去年搞UE4.26的时候，发现居然做了一个傻逼更新：光线跟踪的时候，如果你GI加入Radiance，然后会发现，距离近了什么都还可以，远了一会有一会没有，非常奇葩。为了查找这个问题，我基本把光线跟踪的核心实现算法都看了一轮，甚至TLAS，BLAS之类的生成都看到了，还是找不到原因。最后发现UE自己做了一个裁切，把小物体，半径小于50cm的，距离超过100m的，默认裁切掉了。这就是一个典型的负优化。我认为以后会放弃的。这等于为了一点点性能，回到了20年前。

1. 延迟渲染，Forward+。

两个放一起讲，是因为两个经常被一起对比。我的章节里，有专门讲过延迟渲染的。所以这里还是略讲。这两个的优缺点、对比等等，已经太多了，我也讲不出来什么新花样。在我自己的看法里，我是力推延迟渲染的，原因在于，我认为延迟渲染的一些不足，可以通过其他方式弥补或者改进，带来的好处是巨大的，不可替代的。这里，我不打算延续这个争论，只是表明我自己的看法，这里我只是介绍一下这两种非常有效的优化方案。

1. SIMD优化。

这个优化，我一般认为是鸡肋优化方案，甚至更偏低，基本上属于：没什么卵用。但是，这个优化方案曾经也风靡一时。这个方案的核心是什么呢？其实就是CPU厂家提出的单指令多数据（Single **Instruction** Multiple Data）。例如之前，你做一个Vector4的加法，你需要这样干：

inline Vector4 operator + ( const Vector4& rkVector ) const

{

return Vector4(

x + rkVector.x,

y + rkVector.y,

z + rkVector.z,

w + rkVector.w);

}

但是现在，你不需要了，你只需要这样干：

*\_mm\_add\_ps*( V1, V2 );

看到没，一次搞定。上面用的SSE指令。看起来高大上，牛逼，为什么我认为是鸡肋甚至连鸡肋都不如呢？

很多年前，在某个qq群，有一个qq 的大佬，一个迅雷的大佬，在讨论这个优化。他们对这个优化大加赞赏，号称：迅雷已经封装好了一整套方案，到处都用。我当时还是挺菜逼的，也只是略懂，当初好奇问了一句：这个优化方案，提升非常明显吗？什么时候提升最明显？大佬告诉我，在图片alpha混合的时候，特别明显，效率可以提升四倍！我就更好奇了，我记得我当时问了一句：图片混合为什么不直接用GPU呢？这不是更快？大佬的说法是，他们做UI系统的，主要用GDI，没用GPU。

当初我还不能很理解这个优化方案，也无法判断这个东西到底是真的好用还是不好用，自己觉得有点一知半解。那次之后，我非常深入的了解了一遍这个SIMD优化，并且自己写过一些example之后，得出了结论：这个就是个鸡肋优化方案。

为什么呢？很简单，拼性能，远远不如GPU，但是大大降低了代码的可读性、兼容性等等。举例：如果你做一个图片alpha混合，你用GPU，用compute shader，效率绝对是你simd的一百几十倍起步。但是，如果你只用来做一个向量的加减法，用不用基本上这点提升可以忽略不计。以前的3d引擎，以OGRE为例，早期的时候，数学库用的汇编。后来全部不用了，就是明证。这东西提升有限，降低可读性，兼容性，就是CPU牙膏厂的噱头。而当初讨论的时候，迅雷跟qq大佬为什么力推这个，主要是他们都是做UI系统开发的，不熟悉GPU开发，或者说，那会compute shader好像还是DirectCompute时代？年代太久，我都忘记了。总之，种种意外，才导致了这个优化方式风行一时，但是大浪淘沙，现在谁还力推这个方案，我认为非常不靠谱。

1. 反射渲染优化。

这个，适合很多时候。例如镜子反射，水面反射，汽车的后视镜反射，cubemap反射……。同样的，这个反射方案，我研究过无数次，一次一次的搞，一点一点的提升。到了现在，成了纯理论提升，已经好久没搞过了。

开局的时候，做3D，刚开始压根没理解这个是怎么实现的，只会抄代码，CV大法。后来，慢慢理解了，才知道每一次实时反射，其实是重新渲染了一遍镜头，得到一个渲染贴图（专业术语叫：RenderTarget或者RenderToTexture，简称RT或者RTT，都是这个东东），再反贴回模型上（这里，有一些技巧，例如这个反射镜头的计算，模型的uv计算等等）。理解了这个过程，就能理解为什么水面渲染，镜子渲染这种，为什么消耗那么大了，这等于场景再渲染了一遍！那么，有什么方法优化吗？最早期的时候，理解了这个原理之后，很容易想到的优化方案，是降低渲染贴图的分辨率。例如，一般用256 \* 256甚至128 \* 128。因为，大多数时候，人们不会关注这个反射的效果。甚至于水面的反射效果，还会有一些模糊，混合等等操作，本身就是看不大清楚的。但是，这个优化方案是有限度的，例如，你总不能渲染一个64 \* 64的贴图，那实在是太敷衍。

还有一种优化方式，就是隔帧渲染。例如，你场景有N个RTT，你每帧更新N / 2个，分开来更新。什么时候会有那么多RTT？实时阴影渲染，每个灯一个，镜子，水面，每个都有，另外，还有那种cubemap的神坑，RTT，一次渲染6个……。除以2的好处，是避免帧率大起大落造成的镜头不均匀。为什么视频25帧，画面看起来很好，很多游戏40-50帧，看起来都不平滑？就是渲染的大起大落，不稳定，让画质变差。

以上两个方案，是我还很菜的时候，自己做过的优化方式。后来，用上了Unity，UE之后，自己再没有操心过这类优化。但是，随着技术的提升，我还是想到了其他更多的优化方式。不过，都是些我没有验证过，实现过，也没有看过理论的方式，纯自己推敲出来的。这个方案，是我自己在研究光场渲染的时候，自己想出来的。不过我认为大概率是没问题的。

下面来讲讲这套方案的理论实现。首先，我们来抛出一个问题，场景只有一个天空盒，什么都没有，这个时候，做RTT渲染的时候，再渲染多一次场景，是不是速度极快，仅仅只需要渲染一个box即可？所以，我们优化的方案，其实也是基于这个情况。大部分情况下，场景的mesh，都是静态的。所以，如果某个地方需要做反射，我先把静态的mesh，渲染成一个cubemap图。做镜面反射的时候，直接渲染这个cubemap，不就速度极快了吗？这么做的话，有一个隐患：动态的物体怎么办？例如人物。这个简单，先渲染了静态的，再渲染动态的。但是，这里遮挡关系怎么办？渲染cubemap的时候，保存深度图即可。

以上方案，纯属我无聊的时候理论推敲的，也有一些其他问题，例如透明物体，多重透明物体等等。但是尽量避免即可，或者一些小bug也无伤大雅。我认为是可以实现的，不过确实没试过，也没仔细研究过，不敢保证。

1. GPU优化。

包括但不限于：Compute shader，CUDA，openCL，CUDNN……

CS我认为是图形学近十年来，最优的优化方案，没有之一。CS能做的优化非常多，大部分提升都很明显，例如Forward+就是基于CS实现，例如粒子系统，现在基本上都是基于CS实现粒子系统。我自己写代码测试过，几万个粒子，使用GPU来优化，效率提升几百倍起步。

GPU优化甚至提升了整个AI行业。基本上，绝大部分深度学习框架，都有基于CUDNN的训练方案。英伟达股价不到十年升了几十倍，可见一斑。

一句话：所有使用到大量并行计算的地方，我们都可以用CS优化。由于我其他章节专门讲过CS，也专门讲过粒子系统如何使用CS实现GPU粒子，所以，这里不打算再往下讲了。有兴趣的朋友，可以去看其他章节。

我的结论是：CS属于必备优化技能。优先级最高。

1. 合并渲染批次。

这个我应该在其他章节讲过。如果没讲过，这类常见的优化方案，资料太多了，这里也不讲了。

1. 多线程渲染。

这个多线程渲染，我分为两个阶段。

第一个阶段，那是dx9时代。当时这个时代，渲染是同步的，也就是说，当你调用Draw()函数的时候，是会等到GPU执行完成之后，再返回的。这属于3d引擎的早期阶段。这个阶段，普遍的引擎流程是这样的：

While(true)

{

Update(float delta);

Render();

}

Update里面，执行网络消息收发和处理（网络游戏），骨骼动画，粒子系统的更新，聊天信息的更新，镜头裁切计算等等。而Render里面，执行场景的渲染操作。

这里面，多线程渲染可以怎么实现呢？可以这样：

Thread1:

While(true)

{

Update(float delta);

}

Thread2:

While(true)

{

Render();

}

看起来，是不是很完美？很简单？事实不是如此。因为Update里面，你很可能是需要进行显存操作的。而这个会跟Render造成冲突。所以，这个方案没那么简单。这个需要每当Update里面涉及到显存操作的时候，把这个操作记录下来，在Render的时候，再实现这个操作，这样就不会造成冲突了。但是，这么一来，大大增加了设计的复杂度，而一旦CPU开销并没有太大的时候，优化效果又不突出。因此，这个优化很多引擎连做都懒得做。

我个人认为，这个阶段的多线程渲染，只是个鸡肋，只适用于大量CPU使用导致了CPU瓶颈的项目。

第二阶段，是dx12，vulkan阶段。这个阶段，CPU跟GPU已经进入了异步阶段。典型的是：采用了RenderCommand/RenderQueue的渲染模式，而不是使用传统的Draw同步模式。例如你有很多渲染命令，你可以一个一个放进命令队列，然后异步执行渲染队列。这个时候，CPU和GPU基本是完全独立的两个东东，当然了，你可以用“Fence”来实现同步。

这个设计的好处是显而易见的。首先，你不再需要纠结在Update里面实现显存修改怎么办。现在你只需要把这个修改放进队列，如果实在需要内存跟显存之间的复制，还可以放一个fence，等fence触发的时候通知一下等等，都不会对整套流程造成严重堵塞。甚至，你可以用多个RenderQueue来执行不同的渲染任务（不宜太多），例如一个执行CS，一个执行渲染等等，灵活性高了太多。

但是，现在还写这些代码的人，做这些研究的人，全世界都不多了。为什么？无他，用引擎就对了。你能想到的，别人引擎大概率想到了。自研引擎已经成了一件非常非常奢侈的事情，花钱还出不来大效果。为什么？因为底层技术，包括硬件，基本都在美帝手里，就好比你再逆天，做的引擎不可能拼得过UE4，因为人家跟英伟达，微软等等，都是一伙的。DX12刚出来，你刚拿到文档，人家产品都出来了。估计dx12还是他们一起商量着开发的。我看UE4的代码的时候，经常看到各种NV代码等等。进不了这个圈子，一切都是徒劳。Unity发源于欧洲，现在总部在美帝，是有道理的。

所以，珍惜最后的时光吧。过几年，说不定更少了。现在像我这样还经常写写dx11，dx12代码的人，估计已经非常少了。

1. AI优化。

这个，我认为是未来图形学的大方向。目前，NVidia自己搞了一套东西，叫做NGX，里面包含了很多AI相关的优化，例如DLSS，denoise等等。AI优化能做什么呢？最典型的，大概可以：

1. 、抗锯齿。以前的各种抗锯齿技术，FSAA，MSAA，FXAA，TXAA等等，开销大，各有各的缺陷。而AI天然就适合做这个，效率还贼快。
2. 、把模糊图像变高清。这个可以干吗呢？可以渲染低分辨率的图，用AI转成高清的。例如4K。如果渲染4K画面，肯定效率很低。但是如果渲染2K的，再扩大一倍成4K的，是不是一个可行的方案？现在已经有了吧。
3. 、调色。渲染的画面，如果人眼用PS修饰一下，效果估计好不少。AI可以替代人眼的这个工作，让画面效果更好。

但是，AI优化也有很多不足。就以现在的DLSS为例，并非所有的场景都能表现很好。英伟达自己有很多个版本的UE4，集合了自己的很多feature。其中，就有加入了NGX的，包括DLSS，Denoise等等的，都有。我尝试了一下，效果时好时坏，这也是目前AI的常见bug了。

暂时只想到这些。为了研究这个AI，我通读了图灵奖得主的很多代码以及文档，自己推导了所有公式，徒手C++实现了一个CNN的训练。这部分代码和文档，我也放在了我的github里面。什么方向导数，梯度下降，激活函数，dropout，最小二乘法，交叉熵拟合，BP算法反向求导等等，全部撸了一遍。这么做的目的只有一个：彻底理解AI的流程。做了之后，至少知道了训练的消耗主要在哪里，BP算法的复杂度在哪里，如何提升训练的准确率，数据标注，梯度如何下降等等。

好了，优化方案大概就写到这里了。我仔细回想了一下，有一些已经写过单章的，例如法线贴图，这里提一下也没什么意思。还有一些自己觉得没太大用的，提一下，diss一下，好像也没什么意思。例如有一个叫做early-z相关的各种技术方案，有很多奇奇怪怪的变种，慢慢都没什么人用了。我印象深刻的是intel还力推过一个CPU渲染AABB检测模型遮挡的方案。这个方案的意思是：场景太大，模型太多，可能会有遮挡。先这样简单光栅化一遍，把被遮挡的模型剔除出渲染队列，以达到提升渲染效率的目标。但是老实说，这类算法大多数时候是个负优化，因为你这个提前的预渲染，会造成很大的额外开销。这类方式，只在sample里才有用，因为你sample都是专门为这种优化定制的场景：一个超大场景，遮挡又非常多，当然看起来有效果了。但是实际上，大多数场景，我认为使用这种优化方式，都是得不偿失。此外，还有很多各种优化相关的小技巧，这里好像提了也没有太大意思。例如看着一个透视变换矩阵，其实那个z轴，真的能玩出花。绝大部分人，调用几个API，计算出来一个矩阵，完事了。更高级一点的，自己推导了两遍这个矩阵的实现，也就差不多了。其实，这个还是有一些搞头的。典型的例如z-fighting的解决方案，超远距离视距如何优化这个z等等。这类属于小技巧，应对一些特殊情况，不属于优化方案的范畴，放这里也不合适。

最后总结一下：渲染优化，提升渲染效率，并没有固定的方式。只能说，以上方式，引擎基本上都做了。那么，碰到渲染效率低下的时候怎么办？当然是用性能分析工具，分析一下瓶颈在哪里啊！显卡分析工具很多，方法很多，主要有：

1. 引擎自带的性能分析工具，基本上是个引擎都有。这个非常重要。
2. 一些工具，例如GPU-Z，Pixel for windows等。

3、还有一些处理，例如GPU counter。英伟达就有API能够调用搞这个。随便搜一下，都能搜到相关的例如：

### [Optimizing Performance with GPU Counters](https://developer.apple.com/documentation/metal/render_pipelines/optimizing_performance_with_gpu_counters)

基本上，上面三板斧下来，基本上都能找到你渲染效率低下的原因了。如果是显卡太差，场景太大等等现实原因，那无解。大多数时候，都是有解的。

追加的分割线

通篇下来，我没有想到的是，SIMD优化居然有那么大的争议。我不可能一条一条去回复评论，这也不是我的风格。所以，这里打算再详细深入的讲讲这个东西。另外，存在争议不是什么坏事，连C++这么应用广泛的东西，一样有各方大佬连续diss的，国外有linus，国内有云风，又不是美金RMB，人人满意不大现实。很多争议，往往没有答案——致敬知乎广告词。

先科普一些基础的东西，也就是说：这部分属于公理。先认可了这部分公理，再往下推。如果这部分公理都认识不到，那再往下讲，那就是鸡同鸭讲，没意义了。这里，我分三方：A、普通CPU指令。

1. SIMD指令，包括但不限于：SSE，AVX256，AVX512等。
2. GPU优化，包括但不限于：Compute Shader，CUDA，OpenCL等。

A的优势：兼容性好，随便写，什么都能跑，直接访问RAM，对齐不对齐我都无所谓。

A的劣势：效率低，尤其表现于处理大量数据的时候，只能串行，例如你做一个10000个浮点数求和，你需要一个for循环，一个一个加。

B的优势：效率比A高。例如AVX512，一次可以处理512位数据。也就是16个浮点数。

B的劣势：不能直接对RAM进行处理，而是需要先把数据从RAM加载到CPU，会造成大量的内存带宽消耗，也会有数据传输这部分开销。这也是为什么没做好的话，SIMD优化可能做成负优化的原因之一。（这是我的理解，如果有文档说明可以直接对RAM进行处理的，那就是我的错）。

C的优势：处理大规模并行计算的时候，优势非常大。最典型的例如图片处理，一张图片1920 \* 1024那么多像素的时候，这显然比A和B都更快。

C的劣势：同样需要数据的传输，消耗显存带宽。

这里有一个认知要注意：同样是需要做传输，消耗带宽，但是显然B的传输比C的传输更快。因为C是通过PCI-E进行的数据传输，这个传输效率，有兴趣的可以查看相关文档。

所以我们比较容易得出结论：如果只是两个float相加，A方案无疑是最好的。如果有100个float相加，估计B方案是最好的。如果有10000个float相加，那么C方案显然最合理了。

所以，毫无疑问，A方案是应用最广的。所以你见得最多的代码，都是A代码，B代码和C代码，只在某些场合会用到。

那么，这么清晰的情况下，为什么会有那么大争议呢？我估计是很多人不认同我对于SIMD优化的态度。我的态度开局就讲了：我认为这就是个鸡肋优化，甚至更低。很多人讲了各种理由反驳我，我总不能一个一个回复。这里，我提一些我自己的看法。

1. 有认为SIMD可以做字符串优化的。这种，可以举例，例如两个字符串对比。我一次可以对比16个字符，就问你怕不怕。

先说一个结论：请问，那么多xml解释器，jason解释器，有几个真的用了SIMD指令优化的？是他们都不会吗？那么多基础string类，有几个底层使用了SIMD指令优化的？是他们都不会吗？

我的看法是：这东西更加鸡肋。首先，字符串的长度是不定的，我五个字符对比，请问你用SIMD不是找坑？16个字符对比起步，越多越好，还有传输的开销，这不是鸡肋是什么？其次，就算很长的字符对比，我一个for循环，说不定第一个字符就不同了，不是比你更快？此外，还有字符串的很多使用，是没办法用SIMD优化的，例如我要查找一篇文章里面使用了几个逗号，这不是很常见的需求吗？你确定SIMD能满足各种奇奇怪怪的常用需求？

所以，我认为用SIMD指令优化字符串处理，是intel伪造的需求。你非要伪造一个使用场景，证明这个优化有用，这种手段我见intel做得太多了。

1. 有人说，SIMD优化普遍存在于各种数学库，存在即合理。这个我认，所以我才会说SIMD优化曾经风靡一时。但是，你仔细去看看UE4的数学库，SIMD优化基本头文件里面都有实现，但是CPP里面使用到的，只是极少数一部分。典型的你看看FVector这个基础类，按道理这个基础类，全部强制使用SIMD不就完事了吗？显然没有。这是为什么？

我的理解是：如果你一个函数，300行代码，只使用了一次向量加法，这个时候使用了SIMD，得不偿失。这种做法，属于得到了微量的提升，但是占用了内存带宽。

那么，什么时候才是真的适用呢？我找了一下UE的源码，找到了这样一些情况：

const VectorRegister m1 = VectorLoadAligned(M.M[1]);

const VectorRegister m2 = VectorLoadAligned(M.M[2]);

const VectorRegister m3 = VectorLoadAligned(M.M[3]);

const VectorRegister Half = VectorSetFloat3(0.5f, 0.5f, 0.5f);

const VectorRegister Origin = VectorMultiply(VectorAdd(VecMax, VecMin), Half);

const VectorRegister Extent = VectorMultiply(VectorSubtract(VecMax, VecMin), Half);

VectorRegister NewOrigin = VectorMultiply(VectorReplicate(Origin, 0), m0);

NewOrigin = VectorMultiplyAdd(VectorReplicate(Origin, 1), m1, NewOrigin);

NewOrigin = VectorMultiplyAdd(VectorReplicate(Origin, 2), m2, NewOrigin);

NewOrigin = VectorAdd(NewOrigin, m3);

VectorRegister NewExtent = VectorAbs(VectorMultiply(VectorReplicate(Extent, 0), m0));

NewExtent = VectorAdd(NewExtent, VectorAbs(VectorMultiply(VectorReplicate(Extent, 1), m1)));

NewExtent = VectorAdd(NewExtent, VectorAbs(VectorMultiply(VectorReplicate(Extent, 2), m2)));

const VectorRegister NewVecMin = VectorSubtract(NewOrigin, NewExtent);

const VectorRegister NewVecMax = VectorAdd(NewOrigin, NewExtent);

大概意思是：在一个BOX的计算里，连续多条指令，都是SIMD指令，而增加的传输开销，仅仅是只需要执行一次。也许UE的开发/维护者认为，这种情况比较适合SIMD优化。而大多数情况，恰恰不是连续的向量计算，而是少量的向量计算，夹着大部分的游戏逻辑。这种情况下，使用SIMD显然不合时宜。

那么问题来了，SIMD如此无用，是怎么起来的？还延续了那么多年，就真的这么废柴吗？显然不是的。首先，GPU底层就是使用的SIMD指令。其次，SIMD优化崛起于十多年前，那会,GPU做不了什么通用计算，例如什么图片处理，音频处理，视频的一些处理等等，使用SIMD显然是非常适合而且爆表的，现在如果没有GPU的崛起，SIMD估计还是王道。

那么现在为什么还有那么多人强推SIMD优化呢？有一种人，叫intel。这本身就是他们强推的东西，现在还在强推AVX512，我看了那个强推AVX512的intel blog，下面不少留言嘲讽的，看着老外也都不是省油的灯啊。就连我的文章下面的评论，还有好几个提到intel的某某库，使用了SIMD优化，多牛逼的。大哥，我写了一个3d引擎，然后我写了一堆文章推我的引擎，然后你们引用这些文章来证明我的引擎多么多么牛逼，这真的好吗？这类我基本首先无视。

还有一类，叫既得利益者。我苦练SIMD优化十几年，现在你告诉我这东西没用，我如何能忍？我想起了老杨的名言：The party is over！坊间传言，老杨在中科大的演讲，演讲结束，某中科大硕士问他，搞高能物理前途何方？老杨告诉他，the party is over。然后该硕士问怎么办？老杨说，转行吧。我想，该硕士没有当场骂脏话，已经是他敬老。NND，我披荆斩棘考上中科大，苦练内功三十年，你让我转行，我还有活路吗？我除了跟你死磕，已经没有路了！

所以，洗洗睡吧，该干嘛干嘛，该吵架吵架，该捧捧，该喷喷。你们以为就我一个看不上intel这些套路？就我一个diss这个SIMD的穷途末路？我今天看到一个文章，是大嘴Linus的，他的话更毒：Linus Torvalds: ‘I Hope AVX512 Dies a Painful Death’。我英文菜，我的翻译是这样的：我希望AVX512在痛苦中死去。更好的翻译应该是：AVX512，你TMD去死吧？笑死哥了。原文连接在这里：https://www.extremetech.com/computing/312673-linus-torvalds-i-hope-avx512-dies-a-painful-death。反正啊，这个江湖，多样化才是合理的，没有多样化，I7和14nm工艺，intel还能再卖10年，反正你们不用我的，还能用谁的？P民只能用脚投票了，intel的股价就是P民的投票。