粒子系统

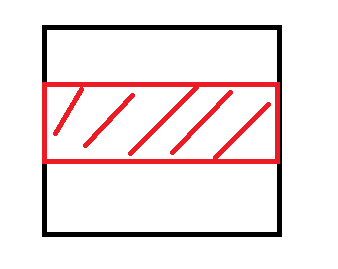
我做过粒子系统的时候，应该都追溯到10多年前了。这些年，估计会有些变化，但是底层理论按道理很难有大的提升。没看卡马克都图形学转VR再转AI了吗？可想而知，图形学跟VR都是日薄西山，没什么大希望的了。

各位大佬不用会心，这些年我虽然没有搞过粒子系统了，但是陆陆续续还是写过一些，看过一些，按道理这个章节会写得还可以的。不可以的话可喷。

先从基础的渲染讲起。众所周知，3D渲染其实就是一个三角形的渲染。渲染这个词不好理解，英文叫Render，本质意思其实就是把一个三角形，一条线一条线的填充上颜色，这就是软光栅。我应该之前讲过。

也就是说，一切模型可以分解成N个三角形。那么，现在要做的，是用GPU并行一个一个画三角形到屏幕即可。

粒子系统作为3D渲染的一部分，是脱离不开这个范畴的。粒子系统，核心渲染的几何体，一般是这三个：**点、线、正方形**。划重点，这个要考！问题来了，为什么是正方形而不是矩形？其实，矩形也不是不可以，也许有引擎可以用矩形，但是正方形有其他一些优势，例如计算简单，例如可以一个边长1的正方形，任意缩放。而正方形同样可以满足矩形的渲染需求，只要贴图是矩形即可，如图：



红色部分为贴图有效部分，白色部分为alpha部分，直接alpha掉就完事了，一样可以做出来矩形的效果，甚至任意形状的效果，只要alpha。

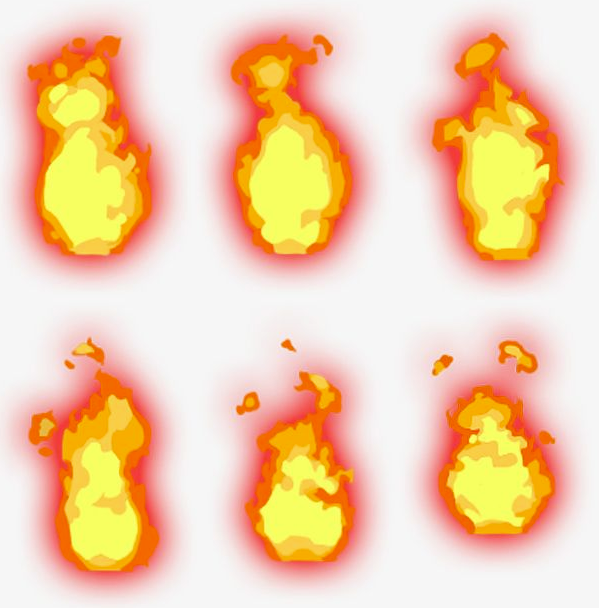
那么，如何用简单的点、线，正方形，实现各种看似复杂的效果？这里，有点复杂，有N种套路，有N种方法，有N种优化，有点不知道从何说起的感觉。

这里，我决定以火焰为例子，讲一讲粒子系统的一些常见做法。涉及到的技术点我都讲一下，没涉及到的看看有没有机会带到。

为什么选择火焰？因为火焰是非常常见的效果，游戏里面随处可见。

为什么火焰？因为火焰的实现方式就有好几种，而不像一些其他粒子，实现方式花样不多。我一种一种的讲讲。

1. 序列帧火焰。这个，顾名思义，就是序列帧动画了，连粒子系统都算不上。放在这里，无非就是做一下对比。实现方式简单，不断换图片即可，或者是不断换UV，而不需要换图片，如图：



图片来自网络，侵删。

这个非常好理解，不过有一个技术点，叫做billboard。这个不记得其他地方有没有讲过了，这里还是讲一下。Billboard的中文翻译，叫公告板。这东西的意思是，永远面向镜头方向。这个东西，在3D之中应用非常广泛。Billboard有好几种方式。例如有永远垂直镜头，有Y轴固定，有 billboardchain，还有Trail。什么意思呢？这里大概解释一下。

1. BillBoard的常见形态：永远面向镜头。假设你看着远处的一颗星星，星星一闪一闪。你在地球的任何地方，看着这个星星，几乎都是一样的，都是那么大，都是那么一闪一闪。为什么呢？其实，因为星星太大，并且距离你太远，你看到的，跟公告板没什么区别。看到了吗？渲染一个正方形，然后贴这张图，做alpha。那么，无论你在哪里，哪个方向，看到的都是这样个样子的图，是不是跟你在地球上看星星差不多？这就是最简单的公告板渲染了，这个正方形，每帧的时候Update，获取镜头的方向，然后计算这个公告板的面方向，让面垂直于镜头方向即可，伪代码大概这样：

Vector3 v = Camera->CalcDiirection();

//假设你创建公告板的时候，也是跟镜头一样的坐标系，

Billboard->SetDirection(v);

//如果不是，根据实际坐标系来算。本地坐标系都没搞清楚的，建议先搞清楚本地坐标系。

计算镜头方向一般怎么做的呢？注意，常见的计算镜头方向，假设镜头是看向-z方向的，一般都是这样：

Vector3 v = Quaternion \* Vector3(0, 0, 01);

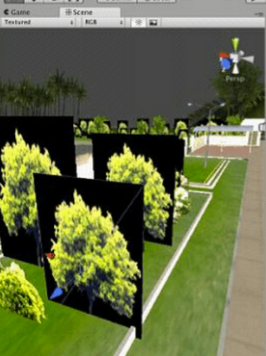
Quaternion 是镜头的旋转四元数。

但是，这里计算公告板，要根据具体的情况来实现，不能直接这样写死。为什么？因为有些情况并非需要直接垂直镜头的方向，还有其他一些情况，这个方向是这样算的：

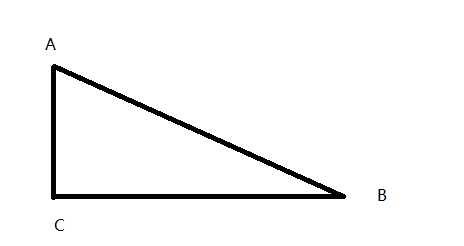
Vector3 v = Camera\_Position - Billboard\_Position;

这就跟一些灯光渲染的时候类似了，镜头方向减去公告板的位置！所以，我这里的代码只有借鉴意义，可以大概理解一下，而不是直接死记硬背，这样毫无意义。

1. BillBoard的另外一种情况。例如一棵树，你远看的话，是看不清楚细节的，一般也不怎么注意字节。这个时候，可以用公告板。但是，树不一样的是，不可能永远直接面向镜头。例如你从高空往下看一棵树，树岂不是横在地上了吗？这种时候，就是这种公告板树穿帮的时候，一般都不渲染了，或者用十字树，穿帮就穿帮，谁怕谁。这种公报版，叫做Y轴固定公告板（这个名字是我叫的，我不知道别人怎么叫，大概这意思），意思是Y轴是固定的，只能左右旋转，其他不能旋转。只能yaw，不能pitch，不能roll。



就是这种了，上图没有做好alpha混合，所以周边黑漆漆的，没有混合掉。图片来源网络，侵删。

看图：

假设A是镜头位置，B是镜头看的方向，C跟B同一个水平，就是Y跟B相等，其实现在就是已知向量AB，然后计算向量CB。为什么呢？因为CB必须水平方向啊，树只能yaw。

这种伪代码大概这样：

Vector3 v = Camera\_Position - Billboard\_Position;

Vector3 y = v.crossProduct(Vector3(0, 1, 0));

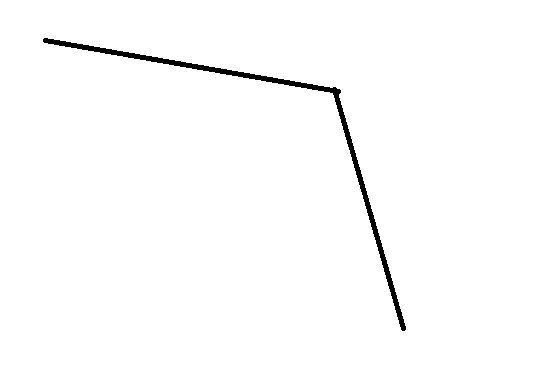
Vector3 CB = y.crossProduct(vector3(0, 1, 0));

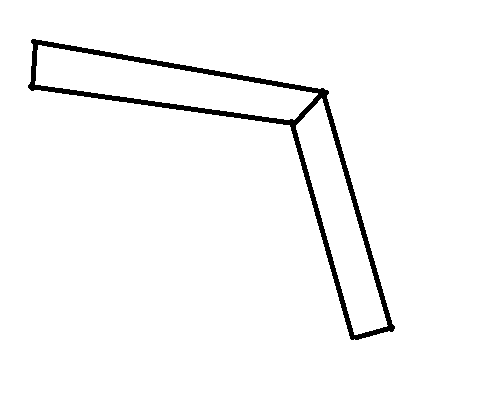
向量叉乘，这是必须掌握的知识点。这个不能掌握，图形学毫无希望。

第一次叉乘，是计算一条垂直于AB跟Y方向组成平面的向量y，得到这个向量之后，再跟Y方向叉乘，就能得到CB向量。

以上代码都是我自己随便想想就写的，按道理没有错，但是不敢保证。看到错了的，欢迎指出。此外，这些代码也是让人大概理解，并不是说所有的都是这么算的，不同引擎，不同要求，处理方式可能不同。

1. BillBoardChain。关于这个，我有一个刻骨铭心的故事。07年，我自己做公司的时候，接了南方电网超高压的一个仿真项目，里面有一个需求，就是漏电模拟。是需要做这样的一个漏电：

反正不是直线，而是曲线。漏电可以用序列帧，但是曲线怎么弄？我思来想去，后来自己计算四边形顶点，实时更新，解决了。心里顿时觉得自己牛逼哄哄。如图，两边都做成公告板：



后来熟悉引擎之后，发现TMD人家早就一整套都有了，这个东西就叫billboardchain。万念俱灰。从那以后，做什么东西，都先查一查别人的思路，别人的方案，有没有做过。

这个东西，其实就是billboard的变种，一般的计算方式，都是直接计算顶点，每一帧，重新计算六个顶点的位置。不能直接用旋转，从矩阵层面解决问题。因为这是同一个mesh，稍稍想一下就知道，不大可能通过直接的旋转解决问题。如果分成两个Mesh，倒是可以通过旋转来实现，但是中间交叠部分，效果会有影响。除非是比较远的地方，不易穿帮，不然还是不要了。

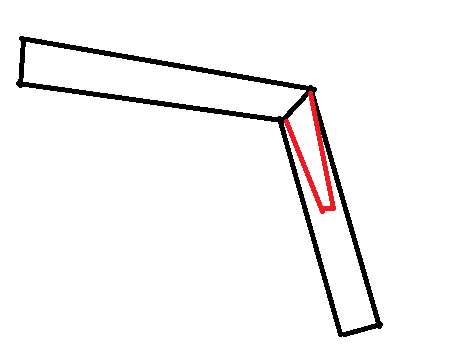
这里，由于计算比较复杂，就不写源码了，有兴趣的，自己随便找个引擎看看即可，看BillBoardChain关键字。

1. Trail。这个叫做拖尾。应该在粒子系统编辑器里面随处可见的关键字。大概是这类效果：



图片来自网络，侵删。

这个，其实可以是BillBoardChain的变种，区别在于，把后面那一截，变成实时计算的尾巴，包括尾巴的长度调节，颜色调节，缩小调节等等，如图：



红色部分是尾巴即可。这个Trail有什么不一样的地方呢？除了动态计算甩尾这个顶点，还可能多节甩尾，也可能改变颜色。颜色的改变，实现方案很多，例如可以通过顶点颜色，shader里面加入顶点颜色的计算。也可以shader里面做颜色渐变，例如N个颜色分段，不同段，根据U坐标做插值，取中间的渐变颜色。我不知道现在的引擎一般怎么搞的，按道理这些方案都是可行的。当然了，你也可以直接通过图片改变颜色，不通过shader或者顶点。

好了，言归正传，先讲一讲一般粒子系统的大概流程：

1. 有一个东西，一般叫做粒子发射器，专门发射粒子。这个就好比吹泡泡。你拿个泡泡，一吹，漫天泡泡乱飞，这个，就叫做泡泡发射器。英文一般叫做Emitter。先定义一个发射器，带方向。
2. 泡泡可能大小不一样，颜色不一样，贴图不一样。这些，一般都是随机的。例如大小有一个随机范围，例如颜色有一个随机范围，例如泡泡的贴图就五种，五种之间随机。甚至，泡泡贴图还可以用序列帧。即使泡泡不用序列帧，如果是其他东西呢？例如一群蝴蝶，不断飞出，当然就需要序列帧了。
3. 泡泡发射出来，飞在空中，有个原始速度，方向，但是可以设置受到其他因素的影响，例如风力。泡泡可能渐隐渐现，可能改变颜色，甚至改变形状等等……
4. 泡泡在空中飞的时候，一般都是公告板形态，意思是：任何方向看过来，都是正对着镜头的。所以，每一个泡泡，每帧，都要计算方向，计算量挺大的。
5. 泡泡一般都有生命周期，这个大多数是有一个随机范围的，不能太短，不能太长，每个泡泡出生的时候就随机好，到点了就破灭了，完蛋了。

以上就是大概流程，现在先不讲优化，先实现。实现都实现不了，谈何优化。

3d游戏里，火焰，烟花，浓烟，一群游鱼，一群蝴蝶，一群乌鸦等等，都可以用序列帧。只要距离比较远的，不容易穿帮。

看图，这样看似复杂的一个火焰效果，其实完全可以差不多的方式实现：



这种，可以用N个粒子发射器实现，N可以根据实际效果，在编辑器里美术编辑。图片来自网络，侵删。

火焰的实现，多种多样。除了上面的粒子系统实现，还有更上面的序列帧公告板实现，还有一个比较有意思的实现，叫做boise实现。这个不属于粒子系统的范畴，这里稍稍提一下。

找不到更好的图了，随便找一张，理解大意即可。

一张大概这样的图，原形的也差不多，大概是从里到外有渐变效果，带少量序列帧。

生成一张noise图，大概这样：

，然后这个noise是不断变化的，用不断变化的nosie，去跟这个原图做混合，alpha掉外面的内容，做成billboard，就可以得到一张看似火苗不断闪烁的火焰图了。

初级粒子系统的实现，就是如此简单。那什么才是高级实现呢？难点其实在于其他地方，例如物理模拟，例如优化。

先讲物理实现。主要就是碰撞，风力。早期，大概十多年前，是物理引擎的巅峰时期，一大堆物理引擎，例如牛顿，havok，physx……我还用过opcode这类非常小众的做碰撞检测。但是，物理引擎一个重要特点，就是非常庞大的计算量。以一个简单的点选射线碰撞检测为例，需要这么做：

1. 根据点选的方向，先得到一个射线。
2. 射线跟镜头范围内的所有Mesh，做一个AABB的碰撞检测。
3. 根据AABB的碰撞检测距离，做一个排序，从近到远，
4. 从近到远，开始进行三角形与射线的相交计算。这个过程，需要遍历Mesh的所有三角形。

运气差的时候，连续遍历几个Mesh，一个Mesh几千三角形，这个计算量得多大？这还是最简单的射线检测。例如典型点选物体，就用到这个简单的射线检测。

这个点选检测，还是点一次才算一次。如果做碰撞，岂不是要做无数次计算？每帧计算？CPu扛不住的。

所以，这里有很多的优化方式。这部分并非粒子系统的范畴，这里略略提一下，例如典型的就是胶囊优化，把人物模型优化成胶囊，就做胶囊碰撞。例如用GPU来计算。除了胶囊，其实还有其他很多东西，例如圆柱，圆锥，球，长方体等，把一个Mesh，尽量模拟成最接近的形状计算。

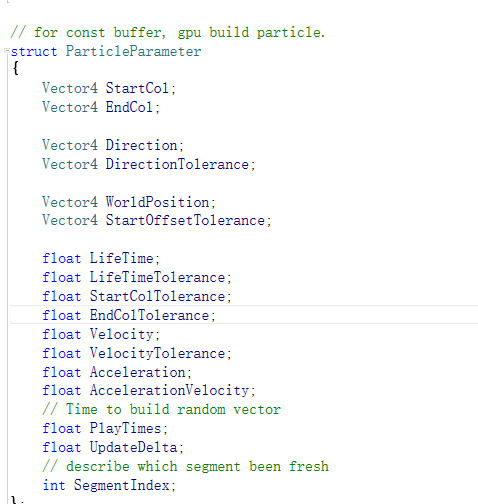
因为物理引擎计算量过大，必须进行硬件优化，当初一大堆没有被硬件厂商收购的引擎，慢慢的，就没有了活路，最终湮灭。现在，混最好的物理引擎，Physx无疑。有NV做后台，混得好是必然的。

这里，我无意大篇幅开讲物理引擎如何对粒子系统进行优化。一来，我并没有亲自写过这块，讲起来除非抄书，这不是我的风格。二来，现在进入了引擎时代，这块已经变成了接口调用了。理解了一般的原理，引擎直接用就完事了，好像已经没有了太多的操作空间。

下面，我从头讲一下一个简单的粒子系统的实现和大概优化。为了讲好这个章节，我直接花时间自己从头撸了一遍代码。代码在github上，包括软件粒子，硬件粒子都有。网址：https://github.com/lygyue/Bloom。这个本来是我给我家小朋友写的小游戏，本身是希望激活小朋友对编程的兴趣，所有代码全是自己手写，除了抄了一下数学部分。然而发现没什么用，小朋友依然没什么兴趣。

粒子系统第一个，一般是要一个粒子发生器，英文叫做Emitter。随便一个引擎，你搜一下particle emitter，几乎都是粒子发射器的封装。功能主要是发射粒子。这就好比放烟花，烟花都是从一个地方往上发射的，一朵一朵的烟花就是粒子组成，烟花本身就是发射器。

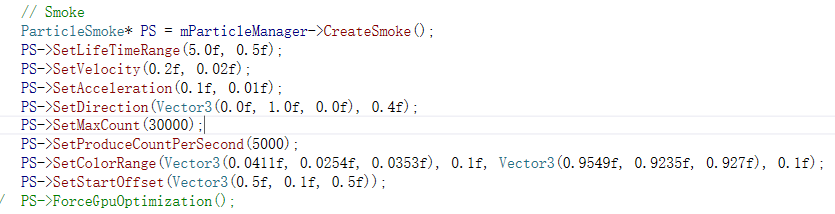
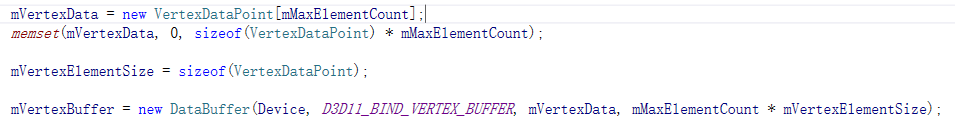
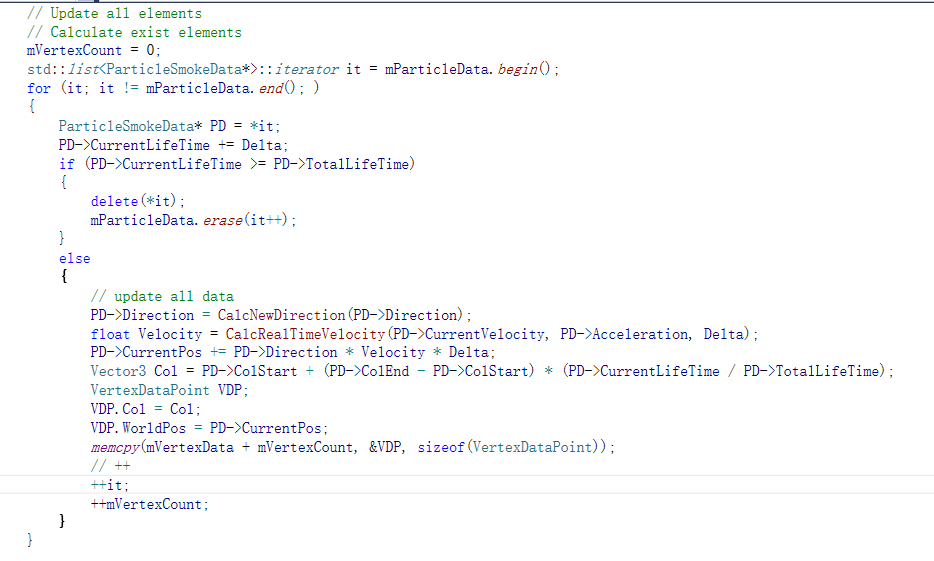
为了简单，我做了一个大量烟雾一样的粒子系统，粒子同时大概有30000个，涉及计算的因素有：粒子的颜色，粒子的方向，粒子的速度，粒子的加速度，粒子的存活时间。大概这样定义一个结构体即可：



这些变量都很好看懂，名字带Tolerance的，是误差范围。例如你一个粒子，生命周期是5秒，误差是0.5秒，那么这个粒子的存活时间就是4.5-5.5秒，随机。

这里还需要定义：每秒钟发射多少个粒子。我这里定义是5000。定义5000是有讲究的。例如最大粒子数是30000，最长存活时间是5.5秒，那么，计算粒子的时候，可以用一个Vertex Buffer，最大定义30000，足够了。5.5 \* 5000，最大粒子数，不可能超过30000个。分配一块这么大的显存，足够。

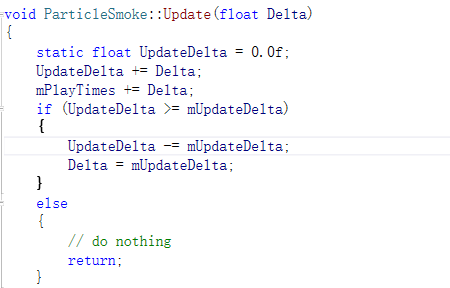
这个，CPU计算粒子的流程大概是怎么样的呢？我这里列一下：

1. 先定义一些粒子的参数，例如速度，加速度，颜色，旋转等等……。我这里大概是这样设置的：
2. 申请一块显存，作为Vertext Buffer，最大值30000 \* sizeof(element)，显存类型必须是dynamic的，然后cpu标志设置为cpu可写。代码大概这样：
3. 自己再申请一块一样大小的内存，内存中计算，自己可以用vector或者list，也可以自己做一个内存池，做一个简单的内存优化，不需要每次申请删除。每个粒子保存一些参数，例如速度，加速度，当前位置等等，每帧计算一次更新。我的代码大概如下：为了简单起见，我用的list，并没有做内存池优化，事实上，正经的还是要做一下内存池优化，自己用自己的容器比较合理，list毕竟只是个通用容器。上面部分代码写在Update里面即可，每帧更新一次。
4. 渲染粒子。D3D的渲染，没什么好说的，代码也有，懒得截图了。

我的PC配置是：I7的CPU，32G内存，P4000显卡。本身跑一个极小的游戏场景，x64、debug下，跑2000+的帧率。然而，开了这个粒子之后，立马降到40帧！

所以说，垃圾的代码或者架构，不是电脑硬件能弥补的。这种毫无优化的方式，CPU根本顶不住。给你I9 也没用。

我们开始第一种优化。

我们可以想见，这里面，大量的计算是无效计算。为什么呢？这就好比视频的刷新率，人眼每秒钟最多只能看到24帧的刷新率，所以一般视频都是25帧、30帧。60帧的视频是凤毛麟角，更高的只在一些高速摄像头，例如光学动作捕捉的摄像头里，获取数据用于计算的时候才见到了，人眼是识别不了的。这里粒子的Update也是一样的，我们的粒子，根本不需要每帧都刷新，按照科学的计算，一秒钟刷新30次就可以了。所以，在update里面做一个时间判断即可，我的代码是这样的：

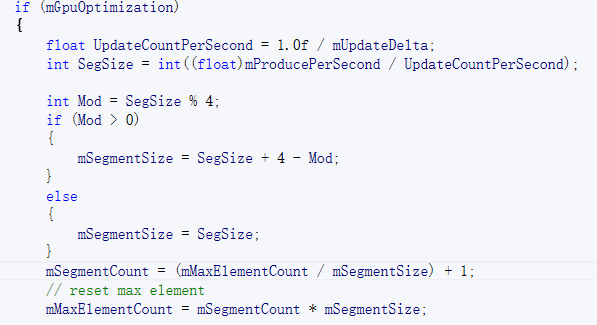
如果不满0.0333s的刷新时间，什么也不用做，不做无效的CPU计算。这样的话，渲染帧率瞬间提升N倍，一般能有300-500帧。大概这样：这里是671帧，算还可以。

而实际显示效果，反正我眼睛是看不出来什么问题，还是就这样。所以说，理解了这些东西，一些简简单单的优化，效率说不定轻易提升十几倍，比你上什么牛逼硬件有效得多。

但是，这个毕竟不是最优方案，最优方案是GPU粒子，用GPU来算。我这里，同样把这个粒子用Compute Shader实现了一遍。实现之后，发现帧率压根没影响，帧率妥妥的2000+。因此，什么SIMD优化，什么多线程计算粒子……弱爆了，对于大量并行的计算，什么优化不如GPU优化……之万一！

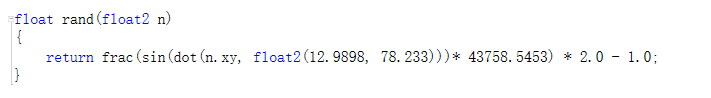
不过，GPU实现起来，需要自己做一下系统的设计，不好做通用的实现。这块我并没有借鉴任何的引擎实现。例如谁没有UE源码啊，要实现的话，去抄一抄不也是好的吗？我是抄都懒得抄了，全根据自己的理解自己写的。所以，我这个方案，不一定是最优方案，毕竟我不是专门研究这块的，只是根据自己的图形学知识自己实现的，路子比较野，有更好方案的，可喷。

说一下我的实现思路：

CS(Compute shader的简称，下同)分两部分：第一部分，属于产生粒子。由于我固定每秒钟刷新30次，粒子总数是已知的，那么，每次Update刷新的粒子数，是固定的，可算的。大概是这样算即可：

这里有一个小小要注意的，我每块计算的时候，我是用4对齐的。事实上，我在CS章节里面讲过，我记得一个GPU的线程簇，是64个线程。所以实际处理的时候，是不是64对齐是更合理的？这块我没有深入研究和测试过，我用的4对齐。我这里每秒钟5000个粒子，算下来一次产生168个粒子。然后总共是179块

第二部分是更新计算所有粒子的状态。我这里简单模拟一下冒烟粒子，只需要根据方向，速度，加速度稍稍计算一下即可。这块需要所有的粒子进行计算，所以，一次执行的线程数是168，dispath是179，刚好三万多点的线程。

这里面有一些细节是要注意的，例如rand函数，在CPU里面实现是很简单的，直接函数调用即可。而在GPU里面，就没这么简单的。实现方式一般是两种，一种就是类似于perline noise这种，用noise函数。或者自己封装一个差不多的做法，我这里是网上抄的一个：网上有各种版本的，大同小异。这个函数的做法，是类似于水波那种，传入的参数如果是有序的，其实返回值也是局部有序的，所以其实不大适合于用来直接替代rand。还有一种做法，是生成一张random的图，图里面的颜色都是随机的，不同的uv取出来的，不就是随机的了吗？这个做法，多用于一些阴影边缘模糊的时候。我这里为了简单起见，直接用了网上抄的一个。效果一般吧，不喜欢的轻喷，毕竟我就随手写个简单sample，主要是为了说明GPU如何优化粒子计算。如果觉得这样做随机数不好，可以const buffer传入cpu的随机数，也不会耗费什么资源。GPU优化后的效果大概这样，帧率完全没变化，还是2000+。有兴趣的可以自己看看代码，只需要这行代码去掉注释，就可以了：



估计有人会问：能不能设计一种通用的GPU优化方案？我认为比较有难度。因为对于Compute Shader而言，每次执行的线程数、必须是固定的，已知的，这样编译的时候就能优化。另外，很多粒子的属性都是不同的，计算方式也不同。我不知道别的知名引擎，例如UE，是如何实现的。我能想到的方案，是根据编辑器输入的各种参数，重新生成代码，再编译。我不知道UE是不是这样做的，我也没看过，毕竟我并不熟专门搞这一块的，有兴趣的话，可以自己去看。

好了，粒子系统，大概就这样了。我也并没有自己设计过非常牛逼复杂的粒子系统，带编辑器那种，所以并不一定很专业。但是我认为图形学这块的原理，我基本上是理解了的，这个优化流程也不会有什么毛病。有讲到不到位的地方，可喷。