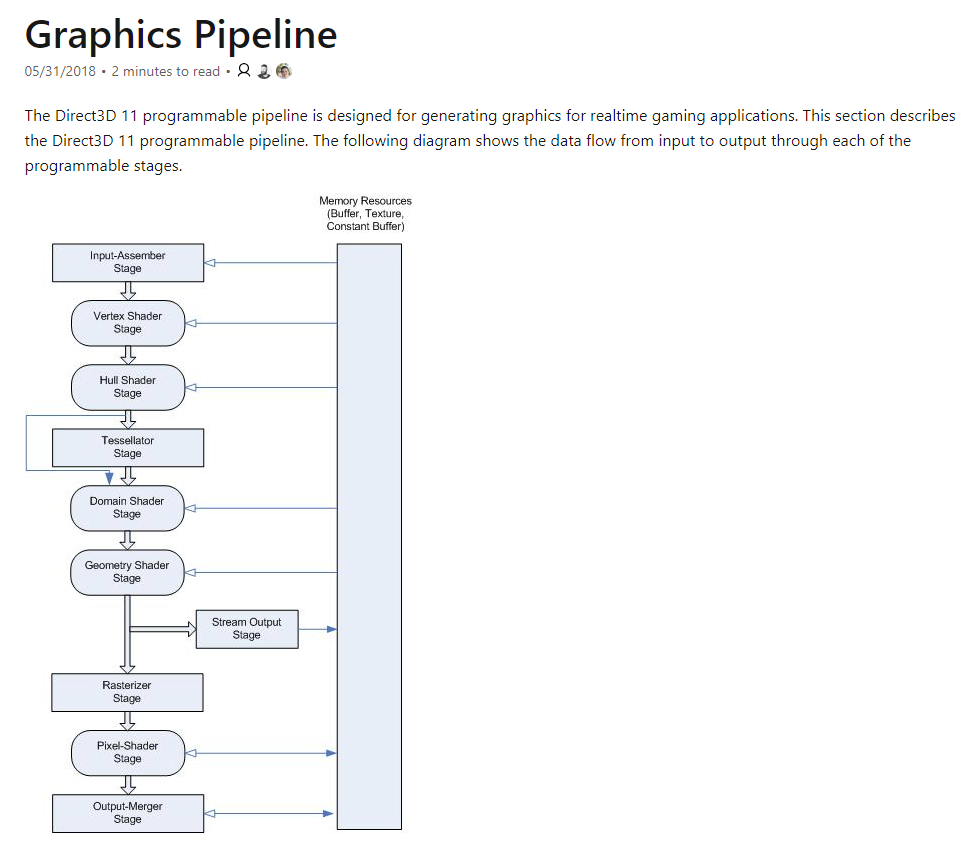
Compute Shader

CS主要是干什么的？为什么我们需要CS？其实，早期的着色器，只有VS跟PS。在更古老的年代，连VS跟PS都没有，就是固定渲染管线。这个就等于直接调接口，传参数，流程，算法，全是固定的，你不能自己弄。

后来，着色器越来越多，增加了GS，CS，还有DS，HS。详细的可以看看下去，下图直接从MS上截取下来的，这种事我很少干，不过这里好像找不到更好的描述方式了。



有兴趣的，可以自己直接上MS官网看详细的。

<https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline>

这里贴一下链接算了。

为什么需要那么多着色器？其实还是跟需求有关。例如GS，主要是GPU粒子的优化用，而DS，HS这类，主要是那个Tessellation的时候用到。Tessellation是个什么东西？我其实有计划有单独的章节。这里稍稍粗略解释一下。首先，我们抛出来一条结论：

一切曲线可以分解成连续的折线/线段！

这个很好理解。同理，我们可以知道：

一切的模型，可以分解成N个三角形。

这些我应该讲过，没讲过再讲一次，这都理解不了的，洗洗睡吧，搞图形学绝对没前途。

那么，三角形越多，自然也就是效果越好，这就是通常说的精模/高模。但是，三角形当然不能无限多，太多的话，计算量太大，显卡负载不了。而且太多的话，由于光栅化的渲染原理，当模型很远的时候，容易造成重面闪烁，也就是所谓的z-fighting，反而效果不好，所以，我们需要做的是LOD，越远的话，模型越粗，越近越精细。Tessellation其实就是传统LOD的延申，传统LOD（Level Of Detail），有很多的局限性，这个以后我写优化章节再仔细讲。反正结论就是：Tessellation是补充了LOD的不足，让近处模型更精细，效果更好。所以新增加了DS，HS。

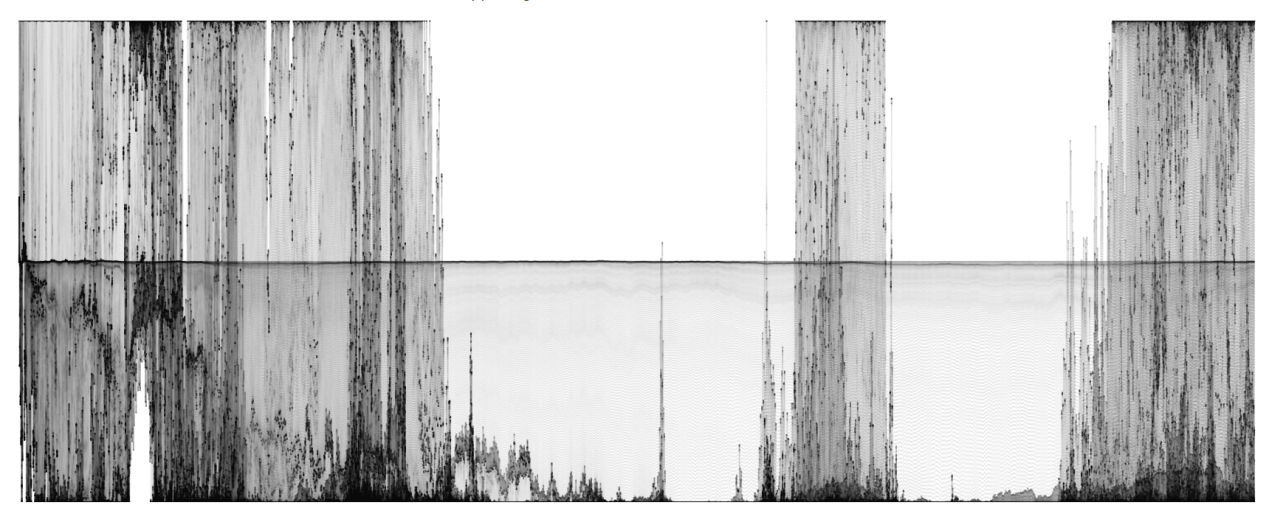
回到正题，CS有什么用呢？我们为什么需要CS？我认为，满足两个条件，那么你可以使用CS。

1. 当你有大量相似/相同的计算的时候。例如，你现在需要做1000000个矩阵乘法。
2. 当大量相似/相同的计算频率非常高的时候。例如，你每帧需要做一次1000000个矩阵乘法。

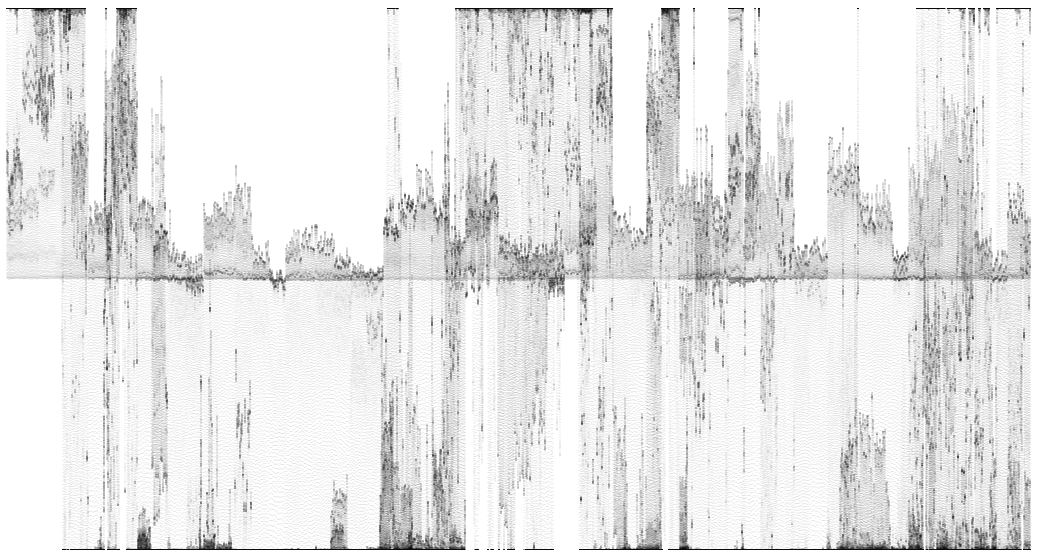
通俗易懂的说法就是：高频的重复计算。假设你有需要做1000000次矩阵乘法，但是你只需要算一次，那么你其实完全不需要用到CS，毕竟只算一次，游戏/程序启动的时候算了就完事了，谁也不会太在乎加载慢了那么一点点。反过来，如果你每次只需要做一个矩阵乘法，但是每帧算一次，那么你也没必要用CS，CPU就够了。

所以，满足以上两个需求的不会太多，我看貌似MS提供的CS的sample里，好像是用CS做了一个排序。这其实完全就是sample了。

最常见的CS用处，就是forward+渲染。老实说，渲染架构从forward到deferred到forward+，有点让人眼花缭乱。不过老实说，还是有本身的意义的。等我讲到deferred的时候，再顺带讲讲forward+，这里只是简单讲讲CS的应用。我以我自己做过的一个CS例子作为案例，介绍一下CS。



这是网上看的一个泼墨的效果，大佬做的。我一时技痒。也用CS做了一个。



这个是我自己做的，用的DX11跟CS。

老实说，我做的跟大佬做的还有差距，我后来想了一下，觉得需要加入Perline Noise才能有更好的效果。不过代码还没写完，就太忙了，没空搞了。就一直放着了。

首先要理解一点，CS是独立于渲染管线之外。上面有个渲染管线的图，你找不到CS的位置。也就是说，CS是个完全独立的东西，你任何时候，不做渲染的时候，你都可以通过CS来实现大量的计算。当然了，大量并行的计算，用CS不说好不好，GPU有更好的支持，例如说CUDA，例如说OpenCL。所以CS我猜测主要还是在渲染的时候用。

由于CS是大量并行的计算，所以就涉及一个东东：线程，线程组。

废话不多说，直接上代码，然后大概解释一下。

[numthreads(1024, 1, 1)]

void ComputeVertexBufferHistoryPixel(uint3 Gid : SV\_GroupID,

uint3 DTid : SV\_DispatchThreadID,

uint3 GTid : SV\_GroupThreadID,

uint GI : SV\_GroupIndex)

{

// refresh velocity and position now

if (RefreshVelocity == 1)

{

float2 uv = float2(Data[GI].Position.x, 0.5f);

float y = rand(float2(uv.x, Time));//noise2d(float2(uv.x, Time));

Data[GI].CurrentVelocity += y;

//Data[GI].CurrentVelocity = y;

}

//VertexBuffer[GI \* 2] = Data[GI].Position;

float RealVelocity = Data[GI].CurrentVelocity \* VELOCITY\_SCALE;

Data[GI].Position += float3(0, TimeDelta \* RealVelocity, 0);

Data[GI].Position.y = clamp(Data[GI].Position.y, -FAR\_DISTANCE, FAR\_DISTANCE);

float3 Original = Data[GI].Position;

// VertexBuffer[GI] = Original;

int VertexPos = 6 \* GI;

float3 QuadVertex[4];

float xOffset = 2.0f / 1920.0f;

float yOffset = 2.0f / 1080.0f;

if (RealVelocity < 0)

{

yOffset = -yOffset;

}

QuadVertex[0] = float3(Original.x - xOffset, Original.y, Original.z);

QuadVertex[1] = float3(Original.x - xOffset, Original.y + yOffset, Original.z);

QuadVertex[2] = float3(Original.x + xOffset, Original.y + yOffset, Original.z);

QuadVertex[3] = float3(Original.x + xOffset, Original.y, Original.z);

// 逆时针

VertexBuffer[VertexPos++] = QuadVertex[0];

VertexBuffer[VertexPos++] = QuadVertex[3];

VertexBuffer[VertexPos++] = QuadVertex[2];

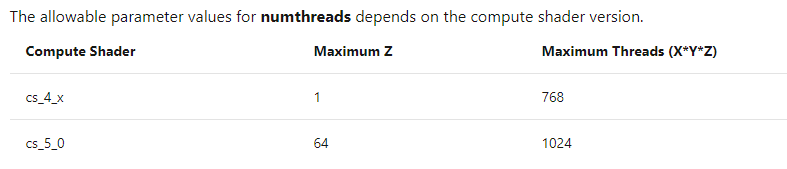
VertexBuffer[VertexPos++] = QuadVertex[0];

VertexBuffer[VertexPos++] = QuadVertex[2];

VertexBuffer[VertexPos++] = QuadVertex[1];

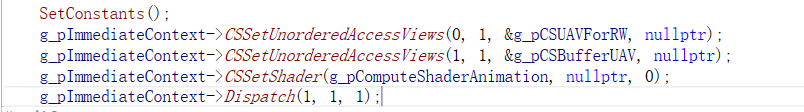
}

第一个，线程数。这里用了一个三维的，定义线程数。这个在CS 5里面，最大是1024.也就是说，一个线程组，最大只支持1024个线程数（CS 5.0），有图为证：



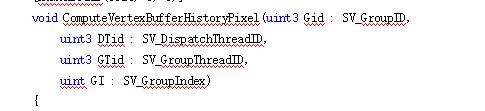
这意味着什么呢？例如，[numthreads(1024, 1, 1)]，这个定义，已经到了极限；当然，你也可以二维定义，例如[numthreads(32, 32, 1)]，这也是到了极限。[numthreads(32, 32, 2)]，这个定义就是错的。什么时候应该一维定义，什么时候应该二维定义？这个，完全取决于你的应用。例如你处理一张二维图片，估计用二维的写法，会简单好理解一点。这就好比写代码，理论上一维数组跟二维数组，都是占那么多的空间，那你什么时候该用一维数组，什么时候该用二维数组呢？难道用一维数组完全替代二维数组，自己算下标不是一样可以的嘛？

Dispatch又是什么用呢？这个，就涉及到大量的线程了。以forward+为例，当你屏幕分割的时候，一个格子大小是16 \* 16，但是你屏幕分辨率是1280 &\* 720，那么，你的线程数就是1280 / 16 = 80，720 / 16 = 45.你现在的线程数是80 \* 45 > 1024。意思是，一个线程组已经不够用了。这个时候，你需要多个线程组。Dispatch的意思，其实就是一次执行多个线程组。下面是我的CPP代码，DX11里面的：



这里面，有一个需要注意的技巧：线程组尽量不要浪费。还是以forward+那个为例。你现在有80 \* 45 = 3600个线程，怎么划分线程组才是最合理的？你当然可以每个线程组用1024个线程，那么你现在需要dispatch的次数是：3600 / 1024 = 3.515625。那么最低就是4.这样一来，其实有接近512个线程浪费掉了。这个应该很好理解吧？浪费太多，不可取。因此，这里其实也是一些细节活。

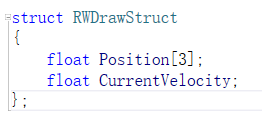
现在理解了线程组，线程，Dispatch的概念，上面几个参数其实很好理解了。你执行一个CS代码的时候，你可能需要知道：当前正在执行的线程组ID，正在执行的线程ID，总的线程分类，线程组分类。所以CS代码里面，这些都是默认就传进来的。也就是这几个参数：



这些其实到处可以查文档的东西，忘记了的时候，随手查查就有了，我也懒得去查一次验证。理解大概的原理基本上写起来都很顺手了。

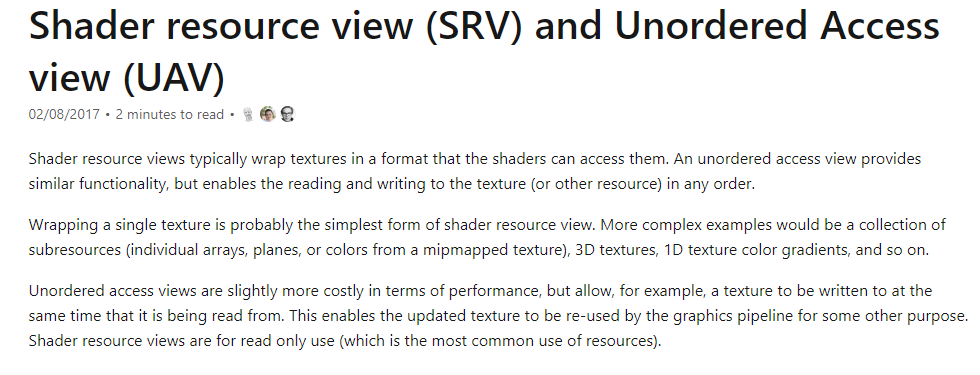
CS还有一个问题，有可能需要把大量的数据，从CPU传入到GPU。这里面，就有一个新概念，叫做UAV（Unordered Access View）。为什么要新增加这个东西？老实说，这东西本质上也就是一块显存，跟传一张图片进去有什么区别吗？还是有区别的。首先，图片的话，就太需要规格了一些，例如一般都是16 \* 16，32 \* 32这样，UAV比较随意一些。你非要抬杠说贴图也可以随意，（首先是贴图真的不能随意，不信自己mapped一下就知道，默认给你对齐。）那么还有另外一个很重要的不同，UAV可以写入自定义的数组数据！

说穿了，你可以自己定义自己的结构体数据，申请显存，传入进去。也可以申请显存，在CS阶段写入，这应该是没有问题的。



这是我定义的结构体。

另外，SRV跟UAV的不同，MS官网也有自己的解释，虽然这个解释我看着不大顺眼，不过确实不该错过。



看不懂的话，可以自己翻译。翻译了实在看不懂，搞图形学其实真的没有什么希望。

关于UAV，最后讲一点题外的。由于GPU是一个完全独立的硬件，显存的属性相当重要，写过DX12的其实深有体会。这个可以通过CPP程序这样理解，一块独立的CPU，一块独立的内存，操作的时候是很简单速度可以很快的，因为内存只有你可以存取。但是，一旦一块内存/显存，需要考虑到别人也要存取的时候，可以预见必须要做互斥。互斥无论是轮询还是回调，都是一个消耗操作。我记得我在哪个文档看过，强烈建议显存，除非非常必要，不然不要用CPU\_ACESS属性，我深以为然。所以，使用UAV的时候也要注意。

总结一下：

1. CS是用于高频的大量计算的着色器，非高频跟大量计算的，不要用。
2. CS主要还是辅助渲染的，因为非渲染的话，应该有更好的方案，例如OpenCL，CUDA……。反正我一般做B端，用CUDA比较多，而 OpenCL没用过。
3. CS的核心概念，就是线程、线程组、Dispatch，UAV。理解了这些概念，写CS还是很容易的。
4. 线程、线程组的分配尽量科学，减少浪费。

好了，关于CS就写到这里。长时间不用，不知道有没有错误，应该没有？有的话欢迎开喷，不介意。