备忘录_计算着色器

笔记本:	DirectX 12		
创建时间:	2022/9/3 10:07	**更新时间:**	2022/9/4 16:26
作者:	handsome小赞		
URL:	https://zhidao.baidu.com/question/368182824185803372.html		

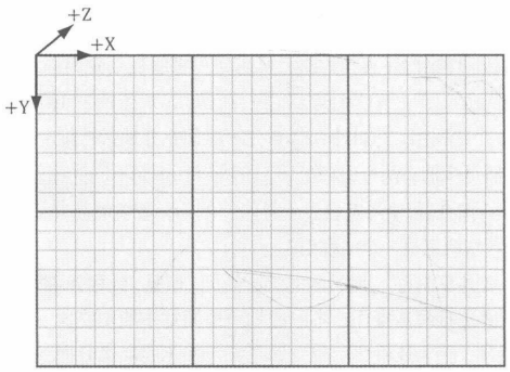
备忘录_计算着色器* 计算着色器并非渲染流水线的组成部分，但是却可以读写 GPU 资源。而且计算着色器也可以参与图形的渲染或单独用于 GPGPU 编程

- 每个线程组都有一块共享内存。
- 一个线程组含有 n 个线程，硬件实际上会将这线程分为多个warp（NVIDIA 公司生产的图形硬件所用的 warp 单位共有 **32** 个线程，未来可能改变），而且多处理器会以SIMD32的方式（即32个线程同时执行相同的指令序列）来处理warp。

  > 在d3d中，我们能以非32的倍数来指定线程组的大小，但是出于性能的原因，我们应当总是将线程组的大小设为warp尺寸的整数倍。

- 对于各种型号的图形硬件来说，线程数为 **256** 的线程组是一种普遍适于工作的初始设置。
  注意 组线程的数量于分派函数参数的设置配合很有讲究，直接关乎计算着色器的工作效率
- **启动线程组**

```
void ID3D12GraphicsCommandList::Dispatch (
  UINT ThreadGroupCountX,
  UINT ThreadGroupCountY
  UINT ThreadGroupCountZ
);
```

> **2D网格**
>
> ```
> cmdList->Dispatch(3, 2, 1);
> ```
>
> 
>
>
> 图 13.3　分派一个规模为 3×2 的线程组。此例假设每个线程组都有 8×8 条线程

- 计算着色器关键组成要素

1. 通过常量缓冲区访问的全局变量
2. 输入与输出资源
3. [numthread (X, Y, Z)]属性，指定 3D线程网格中的线程数量
4. 每个线程都要执行的着色器指令
5. 线程ID系统值参数
   `int3 dispatchThreadID : SV_DispatchThreadID`

- 计算流水线状态对象

  - 不同于渲染（图形）流水线
  - `D3D12_COMPUTE_PIPELINE_STATE_DESC`
  -
    `pRootSignature` 根签名定义了什么参数（CBV，SRV等）才是着色器所期望的输入
    `cs` 该字段指定了计算着色器（通过d3dUtil::CompileShader 编译）

- 数据的输入与输出资源

  - **纹理输入**
    只读资源

    在前一节的计算着色器示例中，我们定义了两个输入纹理资源：

    ```
    Texture2D gInputA;
    Texture2D gInputB;
    ```

    通过给输入纹理 gInputA 与 gInputB 分别创建 SRV（着色器资源视图），再将它们作为参数传入根参数，我们就能令这两个纹理都绑定为着色器的输入资源。例如：

    ```
    cmdList->SetComputeRootDescriptorTable(1, mSrvA);
    cmdList->SetComputeRootDescriptorTable(2, mSrvB);
    ```

    这个过程其实与着色器资源视图绑定到像素着色器的方法相同。但还需要注意的是，SRV 都是只读资源。

  - **纹理输出与无序访问视图**
  - 示例：
    `RWTexture2D<float4> gOutput;`
    输出资源的类型都有个前缀 “RW”意为读与写
    通过尖括号模板语法来指定输出资源的类型与维数
  - 如输出 DXGI_FORMAT_R8G8_SINT 类型的2D整型资源，则在 HLSL 文件里应这样写：
    `RWTexture2D<int2> gOutput;`
  - **无序访问视图**
    为了绑定在计算器着色器中要执行写操作的资源，我们需要将其与称为**无序访问视图**的新型视图关联在一起。
    在代码中，我们用描述符句柄来表示无序访问视图，且通过结构体 D3D12_UNORDERED_ACCESS_VIEW_DESC 来对它进行描述。

    > 为纹理资源创建UAV示例：

    - 通常，可能会将计算着色器输出的纹理用于对几何体进行贴图，因此需要为纹理分别绑定一个UAV和SRV（两者不能同时生效）

```
D3D12_RESOURCE_DESC texDesc;
ZeroMemory(&texDesc, sizeof(D3D12_RESOURCE_DESC));
texDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
texDesc.Alignment = 0;
texDesc.Width = mWidth;
texDesc.Height = mHeight;
texDesc.DepthOrArraySize = 1;

texDesc.MipLevels = 1;
texDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
texDesc.SampleDesc.Count = 1;
texDesc.SampleDesc.Quality = 0;
texDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
texDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS;

ThrowIfFailed(md3dDevice->CreateCommittedResource(
  &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
  D3D12_HEAP_FLAG_NONE,
  &texDesc,
  D3D12_RESOURCE_STATE_COMMON,
  nullptr,
  IID_PPV_ARGS(&mBlurMap0)));

D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
srvDesc.Format = mFormat;
srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
srvDesc.Texture2D.MostDetailedMip = 0;
srvDesc.Texture2D.MipLevels = 1;

D3D12_UNORDERED_ACCESS_VIEW_DESC uavDesc = {};

uavDesc.Format = mFormat;
uavDesc.ViewDimension = D3D12_UAV_DIMENSION_TEXTURE2D;
uavDesc.Texture2D.MipSlice = 0;

md3dDevice->CreateShaderResourceView(mBlurMap0.Get(),
  &srvDesc, mBlur0CpuSrv);
md3dDevice->CreateUnorderedAccessView(mBlurMap0.Get(),
  nullptr, &uavDesc, mBlur0CpuUav);
```

- **利用索引对纹理进行采样**
  - 线程ID SV_DispatchThreadID
  - **注意** 系统对计算着色器中的索引越界行为有着明确定义。**越界** 的读操作总返回 **0**，而向 **越界** 处写入数据时不会实际执行任何操作。
  - 采样
    计算着色器不可直接参与渲染，而 Sample 方法会自行根据屏幕上纹理所覆盖的像素数量而自动选择最佳的 mipmap 层级，所以应当使用 SampleLevel 方法来显示指定 mipmap层级。

    > 注意 若 mipmap层级参数为小数，则该小数用于在开启 mipmap线性过滤的两个 mipmap 层级之间进行插值

  - 归一化
    纹理坐标范围为 [0,1]，所以我们需要对整数索引进行取归一化纹理坐标的操作。得知道索引和纹理大小。
- **结构化缓冲区资源（structured buffer)**

> StructuredBuffer<>
>
> ```
> struct Data
> {
>     float3 v1;
>     float2 v2;
> };
>
> StructuredBuffer<Data> gInputA : register(t0);
> StructuredBuffer<Data> gInputB : register(t1);
> RWStructuredBuffer<Data> gOutput : register(u0);
> ```
>
> 结构化缓冲区是一种由相同类型元素所构成的简单缓冲区 —— 其本质上是一种数组。
> 结构化缓冲区可以像纹理那样与流水线相绑定。我们为它们创建SRV或UAV的描述符，再将这些描述符作为参数传入需要获取描述符表的根参数。或者，我们还能定义以描述符为参数的根签名，由此便可以将资源的虚拟地址作为根参数直接进行传递，而无须涉及描述符堆（这种方式仅限于创建缓冲区资源的 SRV 或 UAV，并不适用于纹理）

- **原始缓冲区**

  > 自行查看参考 SDK文档

- **将计算着色器的执行结果复制到系统内存**
  - 缓冲区其实位于显存中
  1. 所以，首先要以堆属性 D3D12_HEAP_TYPE_READBACK 来创建系统内存
  2. 再通过 ID3D12GraphicsCommandList::CopyResource 方法将 GPU 资源复制到系统内存资源中（系统内存资源必须与待复制的资源有着相同的类型和大小）。
  3. 最后利用 Map 映射API函数堆系统内存缓冲区进行映射，使CPU可以顺利读取其中的数据。

- 线程标识的系统值

  - **线程组ID**

    > SV_GroupID
    > 分配线程组个数 Gx * Gy * Gz，则 ID 范围 (0,0,0) ~ (Gx-1, Gy-1, Gz-1)

  - **组内线程ID**

    > SV_GroupThreadID
    > 线程组的个数 X * Y * Z，则 ID 范围 (0,0,0) ~ (X-1, Y-1, Z-1)
    > 唯一标识 **（局部的）**

  - **调度线程ID**

> SV_DispatchThreadID
> xyz = groupID.xyz * ThreadGroupSize.xyz * groupThreadID.xyz
> 唯一标识 **（全局的）**

- **组内线程ID的线性索引**

> 一维
> SV_GroupIndex

- 追加缓冲区与消费缓冲区

> - **消费结构化缓冲区**：一种输入缓冲区
> **追加结构化缓冲区**：一种输出缓冲区
>
> ```
> ConsumeStructuredBuffer<Particle> gInput;
> AppendStructuredBuffer<Particle> gOutput;
> ```
>
> ```
> // 对输入缓冲区中的数据元素之一进行处理（即“消费”，从缓冲区中移除一个元素）
> Particle p = gInput.Consume();
> ```
>
> ```
> // 将规范化向量追加到输出缓冲区
> gOutput.Append( p );
> ```
>
> 数据元素一旦经过处理（即消费），其他线程就不能再对它进行任何操作了（事实上也就是从消费缓冲区中移除掉了）。而且一个线程也只能处理一个数据元素。
> **注意** 追加结构化缓冲区的空间是不能动态扩展的。但是，它们一定有足够的空间来容纳我们要向其追加的所有元素。

- 共享内存与线程同步

> - 每个线程组都有一块称为共享内存或线程本地存储器的内存空间。
> - 共享内存的声明如下
>
> ```
> groupshared float4 gCache[256];
> ```
> 数组大小自定，但是线程组共享内存的上限为 **32kb**
>
> - 使用 SV_GroupThreadID 语义对共享内存进行索引
> - **使用过多的共享内存会引发性能问题**
>
> 使用过多的共享内存会引发性能问题[Fung10]，下面给出例子对此进行详解。假设现有一款最多支持 32 KB[1]共享内存的多处理器，而用户的计算着色器则需要共享内存 20 KB。这意味着只有为每个多处理器设置一个线程组才能满足此限制，因为 20 KB + 20 KB = 40 KB > 32 KB，所以没有足够的共享内存供另一个线程组使用[Fung10]。这样一来就限制了 GPU 的并发性，因为多处理器将无法在多个线程组之间进行切换而屏蔽处理过程中的延迟（13.1 节中曾提到，建议每个多处理器至少设有两个线程组）。因此，即使这款硬件在技术上仅支持 32 KB 的共享内存，但是通过缩减内存的使用量却能令其性能得到优化。
>
> - **共享内存常见的应用场景是存储纹理数据**
>
> 如**模糊图像**这种工作，需要对同一个纹素进行**多次拾取。**纹理采样受限于内存带宽和内存延迟，是一种速度较慢的GPU操作。所以为避免密集的纹理拾取操作，需要先将纹理样本全部预加载至共享内存块。
>
> ```
> Texture2D gInput;
> ```

```
RWTexture2D<float4> gOutput;

groupshared float4 gCache[256];

[numthreads(256, 1, 1)]
void CS(int3 groupThreadID : SV_GroupThreadID,
     int3 dispatchThreadID : SV_DispatchThreadID)
{
  // 每个线程都对纹理进行采样，再将采集数据存储在共享内存中
  gCache[groupThreadID.x] = gInput[dispatchThreadID.xy];

  // 等待组内的所有线程都完成各自的任务
  GroupMemoryBarrierWithGroupSync();

  // 此时，读取共享内存的任意元素并执行计算任务都是安全的
  float4 left = gCache[groupThreadID.x - 1];
  float4 right = gCache[groupThreadID.x + 1];

  ...
}
```

注意 需要使用`GroupMemoryBarrierWithGroupSync()` 进行线程同步

- 图像模糊

  - **图像模糊理论**

    针对源图像中的每一个像素 Pij，计算以它为中心的 m x n 矩阵的加权平均值。

    - m = 2a+1，n = 2b+1，强制m、n为奇数，确保具有中心
    - a 垂直模糊半径
      b 水平模糊半径
    - m x n 模糊核
    - G 权值之和

      > 改变不同的 σ 实现高斯模糊
      >
      > $$高斯函数\ G(x) = \exp\left(-\frac{x^2}{2\sigma^2}\right)$$
      >
      > 取不同的 σ 时 G 不同

    - **高斯模糊的分离性**
      1. 先通过 1D 横向模糊（horizontal blur）将输入的图像 I 进行模糊处理
      2. 对上一步输出的结果再次进行 1D 纵向模糊（vertical blur）处理

  - **渲染到纹理技术**
    后台缓冲区其实就是一种位于交换链中的纹理资源
    - 渲染到离屏纹理（渲染到纹理）的操作：
      1. 与常规的操作一样，通过 `ID3D12GraphicsCommandList::OMSetRenderTargets` 将后台缓冲区绑定到渲染流水线，确保场景绘制到后台缓冲区中
      2. 然后利用 `ID3D12GraphicsCommandList::CopyResource` 将后台缓冲区的资源复制到离屏纹理（默认堆 `D3D12_HEAP_TYPE_DEFAULT` ）
      3. 在离屏纹理上进行计算

4. 再利用 CopyResource复制回后台缓冲区
5. 使用 `IDXGISwapChain::Present` 方法呈现后台缓冲区到屏幕上。D3D12_RESOURCE_STATE_RENDER_TARGET

> **注意** 在执行copyResource前需要转换资源
> 源资源状态： `D3D12_RESOURCE_STATE_COPY_SOURCE`
> 目标资源状态：`D3D12_RESOURCE_STATE_COPY_DEST`
> **注意** 我们应当尽量减少流水线切换的交替执行。

- **图像模糊的实现概论**
  - 模糊算法的处理过程：

> 利用高斯模糊的分离性，准备两个可读写的纹理缓冲区：A，B

1. 给纹理 A绑定 SRV，作为计算着色器的输入
2. 给纹理 B绑定 UAV，作为计算着色器的输出
3. 分派线程组执行横向模糊操作
4. 给纹理 B绑定 SRV，作为计算着色器的输入
5. 给纹理 A绑定 UAV，作为计算着色器的输出
6. 分派线程组执行纵向模糊操作

> 其中纹理 A和纹理 B在某些时刻分别充当了计算着色器的输入与输出，但无法同时担任两种角色。