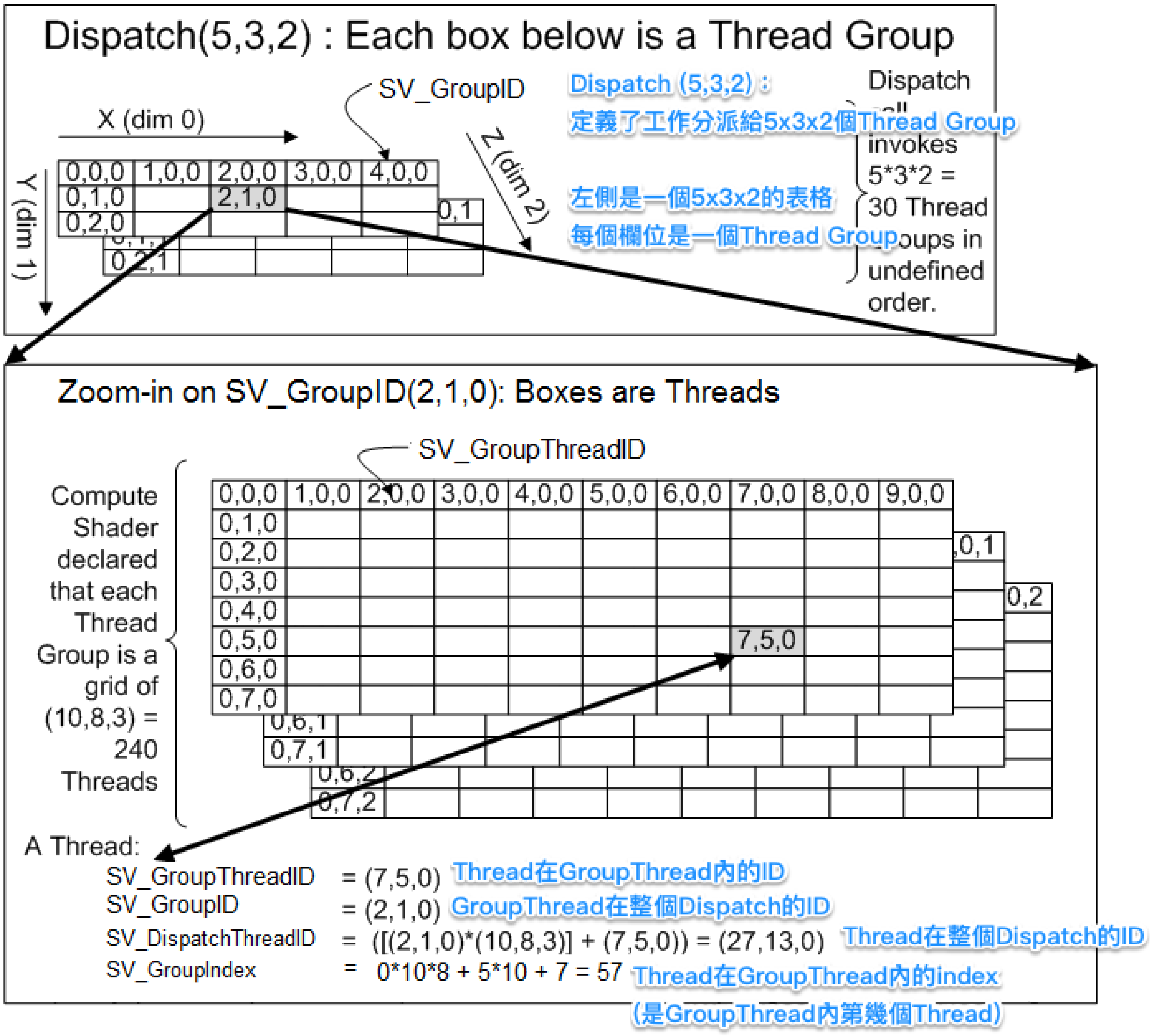
使用ComputeShader来输出计算结果

shader代码分析

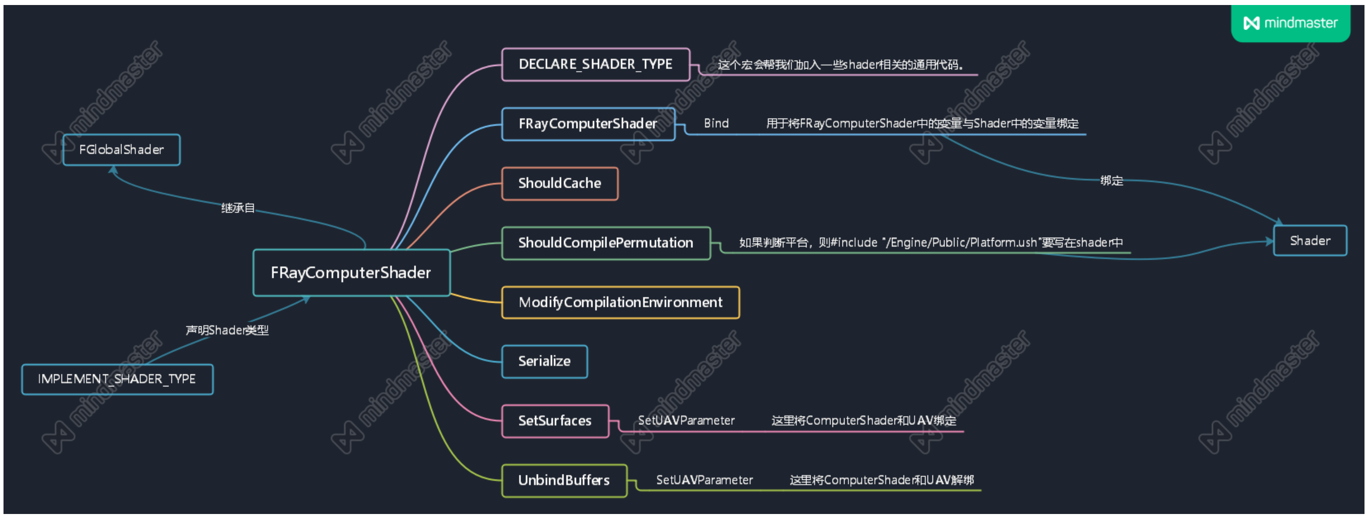
```
#include "/Engine/Public/Platform.ush" //(1)
RWTexture2D<float4> OutputSurface;  //(2)

[numthreads(32, 32, 1)] //(3)
void MainCS(uint3 ThreadId : SV_DispatchThreadID) //(4)
{
    float sizeX, sizeY;
    OutputSurface.GetDimensions(sizeX, sizeY);
    float2 iResolution = float2(sizeX, sizeY);
    float2 uv = (ThreadId.xy / iResolution.xy);

    OutputSurface[ThreadId.xy] = float4(uv, 0, 1);
}
```

- (1) 如果在ComputerShader的类内的成员函数ShouldCompilePermutation有判断平台则需要加这个include
- (2) 定义了一个Computer Shader要存取的贴图资源，可以在ComputerShader的类内进行绑定
- (3) 意思是创建了一个Thread Group，在执行时会把运算分派给每个Thread去执行。在这里表示一个Thread Group内有32 * 32 * 1 个Thread。
- (4) 这里声明了ThreadID表示SV_DispatchThreadID。具体什么是DispatchThreadID可以参考下图。简单来说就是DispatchThreadID是根据GroupID和GroupThreadID计算出来的某个线程在所有线程中的坐标


Dispatch(5,3,2) : Each box below is a Thread Group
SV_GroupID
X (dim 0)
Y (dim 1)
Z (dim 2)
0,0,0
1,0,0
2,0,0
3,0,0
4,0,0
0,1,0
2,1,0
0,2,0
0,1
0,2,1
Dispatch call invokes 5*3*2 = 30 Thread Groups in undefined order.
Dispatch (5,3,2)：
定義了工作分派給5x3x2個Thread Group
左側是一個5x3x2的表格
每個欄位是一個Thread Group
Zoom-in on SV_GroupID(2,1,0): Boxes are Threads
SV_GroupThreadID
0,0,0
1,0,0
2,0,0
3,0,0
4,0,0
5,0,0
6,0,0
7,0,0
8,0,0
9,0,0
0,1,0
0,2,0
0,3,0
0,4,0
0,5,0
0,6,0
0,7,0
7,5,0
0,1
0,2
0,6,1
0,7,1
0,6,2
0,7,2
Compute Shader declared that each Thread Group is a grid of (10,8,3) = 240 Threads
A Thread:
SV_GroupThreadID = (7,5,0)
Thread在GroupThread內的ID
SV_GroupID = (2,1,0)
GroupThread在整個Dispatch的ID
SV_DispatchThreadID = ([(2,1,0)*(10,8,3)] + (7,5,0)) = (27,13,0)
Thread在整個Dispatch的ID
SV_GroupIndex = 0*10*8 + 5*10 + 7 = 57
Thread在GroupThread內的index
(是GroupThread內第幾個Thread)

FRayComputerShader代码分析

- 这个在Shader篇的ComputerShader中分析过了。这里用脑图简要分析一下ComputerShader的结构


mindmaster
FGlobalShader
继承自
FRayComputerShader
声明Shader类型
IMPLEMENT_SHADER_TYPE
DECLARE_SHADER_TYPE
这个宏会帮我们加入一些shader相关的通用代码。
FRayComputerShader
Bind
用于将FRayComputerShader中的变量与Shader中的变量绑定
绑定
Shader
ShouldCache
ShouldCompilePermutation
如果判断平台，则#include "/Engine/Public/Platform.ush"要写在shader中
ModifyCompilationEnvironment
Serialize
SetSurfaces
SetUAVParameter
这里将ComputerShader和UAV绑定
UnbindBuffers
SetUAVParameter
这里将ComputerShader和UAV解绑

渲染线程

```
static void RayTracing_RenderThread(
    FRHICommandListImmediate& RHICmdList,
    ERHIFeatureLevel::Type FeatureLevel
)
{
    check(IsInRenderingThread());
    TArray<FVector4> Bitmap;

    // --- (1) begin ---

TShaderMapRef<FRayComputerShader>ComputerShader(GetGlobalShaderMap(FeatureLevel));
    RHICmdList.SetComputeShader(ComputerShader->GetComputeShader());
    // --- (1) end ---

    int32 SizeX = 256;
    int32 SizeY = 256;

    FRHIResourceCreateInfo CreateInfo;
    FTexture2DRHIRef Texture = RHICreateTexture2D(SizeX, SizeY, PF_A32B32G32R32F,
1, 1, TexCreate_ShaderResource | TexCreate_UAV, CreateInfo); //(2)
    FUnorderedAccessViewRHIRef TextureUAV = RHICreateUnorderedAccessView(Texture);
    ComputerShader->SetSurfaces(RHICmdList, TextureUAV); //(3)
    DispatchComputeShader(RHICmdList, *ComputerShader, SizeX / 32, SizeY / 32, 1);
//(4)
    ComputerShader->UnbindBuffers(RHICmdList);

    Bitmap.Init(FVector4(1.f, 0.f, 0.f, 1.f), SizeX * SizeY);

    uint32 LolStride = 0;
    uint8* TextureDataPtr = (uint8*)RHICmdList.LockTexture2D(Texture, 0,
EResourceLockMode::RLM_ReadOnly, LolStride, false);
    uint8* ArraryData = (uint8*)Bitmap.GetData();
    FMemory::Memcpy(ArraryData, TextureDataPtr,
GPixelFormats[PF_A32B32G32R32F].BlockBytes * SizeX * SizeY);
    RHICmdList.UnlockTexture2D(Texture, 0, false);

    SaveArrayToTexture(&Bitmap, SizeX, SizeY);
}
```

- (1) ComputerShader的创建以及将其设置为当前的ComputerShader
- (2) 创建一个2D纹理，大小是SizeX * SizeY ,格式是ABGR,且每位为32位float数, TexCreate_ShaderResource 是纹理可以用作着色器资源,TexCreate_UAV是纹理可以用作UAV资源

> UAV:UAV事实上是一个可以多线程随机读写的缓冲区

- (3) 将UAV视图和ComputerShader绑定,这样OutputSurface的计算结果存储到了UAV的缓冲区
- (4) DispatchComputeShader的操作是用来划分Thread Group,这里相当于创建了 8 * 8 * 1的线程组，在shader代码里[numthreads(32, 32, 1)]相当于每个线程组里有32 * 32 * 1个线程。这里应该同时也运行了Shader