第一次面试CMGE中手游U3D游戏引擎工程师

我的缺点：

1. 自我介绍不到位，不知道介绍什么
2. GI算法存在细节不清楚的问题，可能是很紧张一时之间宕机了
   - 如如何遍历Volume
   - World pos存在和Volume的转换问题
   - 对于DX12项目的介绍成问题，没有整体逻辑思路
   - 不清楚每个纹理的具体类型
   - 对于VXGI的各向异性过滤仍存在不理解的方向
     面试官问的问题：
     1.clipmap和3Dmipmap的优缺点各是什么
     2.LPV和VXGI的主要区别是什么
     3.RSM做了什么改进没有，比如Raymatching
     4.RSM为什么可以收获到bounce的内容
     5.存储球谐函数的纹理的uniform是什么
     6.有没有了解过后效
     7.有没有加入motion vector
     8.关于Raytracing

DX12基础架构

> *一些我认为的注意事项*
> *因为是U3D引擎工程师，然后在招聘简历上写的有OpenGL或者DX编程经验最好*
> *所以我并没有特别的去强调DX12的特性以及该如何写，只是把我认为和书上不一样的地方，可以作为闪光点的地方写了下来*

首先是Winmain函数，注册窗口，显示窗口，然后while游戏主循环

> *我认为这些都不太要紧，所以没有特别的去强调一些技巧，下面贴出部分我认为是技巧的东西*

```language-cpp
//传递一个指针，以便于我们可以在消息处理函数的时候使用实例（对象）里的内容
WinMain:
```

```
SetWindowLongPtr(hwnd, GWLP_USERDATA, reinterpret_cast<LONG_PTR>
(gSample.get()));
```

> *然后在窗口过程函数里面有一句：*
> *这个巧思的原因是什么呢，原因是因为有一定的抽象因素Ht*

```
>auto sample = reinterpret_cast<DXRSExampleGIScene*>                language-cpp
(GetWindowLongPtr(hWnd, GWLP_USERDATA));
>```

然后介绍**主循环，是有消息就处理消息，没有消息就运行渲染器实例**

然后**运行Timber**，传递一个回调函数Timber
根据选项选择渲染是同步还是异步
这个项由ImGui控制
GBuffer，Shadowmapping
首先还是看看同步渲染
：**首先渲染GBuffer和ShadowMapping**

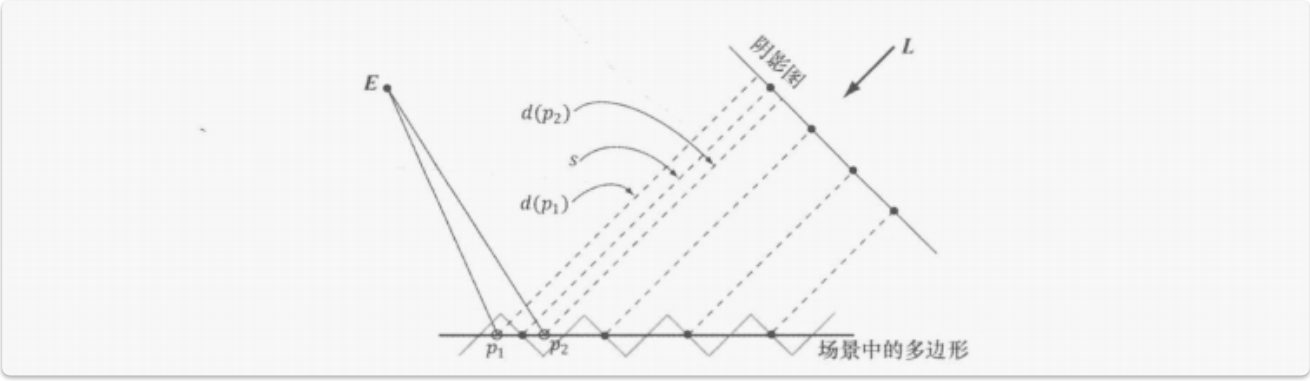
GBuffer	uniform
float4 color : SV_Target0;	DXGI_FORMAT_R8G8B8A8_UNORM
float4 normal : SV_Target1;	DXGI_FORMAT_R16G16B16A16_SNORM
float4 worldpos : SV_Target2;	DXGI_FORMAT_R32G32B32A32_FLOAT

UNORM 代表 Unsigned Normalized,即无符号规格化。
SNORM 代表 Signed Normalized,即有符号规格化。
shadowmapping :DXGI_FORMAT_D32_FLOAT

对于shadowmapping有很重要的一点需要讨论：
我们的projection到底是什么？
首先我们给出关于shadowMapping的Vertex Shader代码
```cpp
float4 VSOnlyMain(VSInput input) : SV_Position
{
    float4 result;
    result = mul(World, float4(input.position.xyz, 1));
    result = mul(LightViewProj, result);
    result.z *= result.w;
//ht
    return result;
}
```

下面给出一个关于shadow mapping流水线对象设置的小插曲
```

```cpp
mRasterizerStateShadow.FillMode = D3D12_FILL_MODE_SOLID;
        mRasterizerStateShadow.CullMode = D3D12_CULL_MODE_BACK;
        mRasterizerStateShadow.FrontCounterClockwise = FALSE;
        mRasterizerStateShadow.SlopeScaledDepthBias = 10.0f;
        mRasterizerStateShadow.DepthBias = 0.05f;
        mRasterizerStateShadow.DepthClipEnable = FALSE;
        mRasterizerStateShadow.DepthBiasClamp =
D3D12_DEFAULT_DEPTH_BIAS_CLAMP;
        mRasterizerStateShadow.MultisampleEnable = FALSE;
        mRasterizerStateShadow.AntialiasedLineEnable = FALSE;
        mRasterizerStateShadow.ForcedSampleCount = 0;
        mRasterizerStateShadow.ConservativeRaster =
D3D12_CONSERVATIVE_RASTERIZATION_MODE_OFF;
```



这些是Direct3D 12中创建阴影贴图时的典型光栅化状态配置,各项作用如下:

1. *FillMode: 使用默认的填充模式,加速绘制速度。*
2. *CullMode: 背面剔除,减少绘制图形量。*
3. *FrontCounterClockwise: 设置默认的顶点顺序语义。*
4. *SlopeScaledDepthBias: 减轻阴影贴图近面误差。*
5. *DepthBias: 结合上项修正近面不必要的自遮挡。*
6. *DepthClipEnable: 关闭深度裁剪,需要渲染整个视锥。*
7. *DepthBiasClamp: 限制偏移量防止变形。*
8. *MultisampleEnable: 关闭多采样节省运算量。*
9. *AntialiasedLineEnable: 关闭线条抗锯齿。*
10. *ForcedSampleCount: 使用默认采样次数。*
11. *ConservativeRaster: 关闭保守栅格化加速绘制。*
    如果不这样配置,可能导致的问题包括:
12. *生成的阴影贴图存在自遮挡问题。*
13. *远处阴影细节被裁剪丢失。*
14. *速度变慢,质量不提升。*
15. *经典近面Peter Panning问题。*

> 16. *因偏移过大导致的阴影形变。*
> 17. *采样数超出需要造成的资源浪费。*
>
> *综上,这些配置用于优化速度同时提升阴影贴图的质量。*

这里想讲一下透视除法，对于games101里面的

- **Projection transformation**
  - Orthographic projection (cuboid to “canonical” cube [-1, 1]^3)
  - Perspective projection (frustum to “canonical” cube)

是并不太严谨的
我们看一下real time rendering对于正交矩阵的描述

In OpenGL the axis-aligned cube has a minimum corner of $(-1,-1,-1)$ and a maximum corner of $(1,1,1)$; in DirectX the bounds are $(-1,-1,0)$ to $(1,1,1)$. This

$$\mathbf{P}_o = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{4.64}$$

which is a mirroring matrix. It is this mirroring that converts from the right-handed viewing coordinate system (looking down the negative z-axis) to left-handed normalized device coordinates.

DirectX maps the z-depths to the range $[0,1]$ instead of OpenGL's $[-1,1]$. This can be accomplished by applying a simple scaling and translation matrix applied after the orthographic matrix, that is,

$$\mathbf{M}_{st} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.65}$$

Reflective shadow mapping

然后就是重点，对于RSM的渲染，
还是老样子，给出关键部分

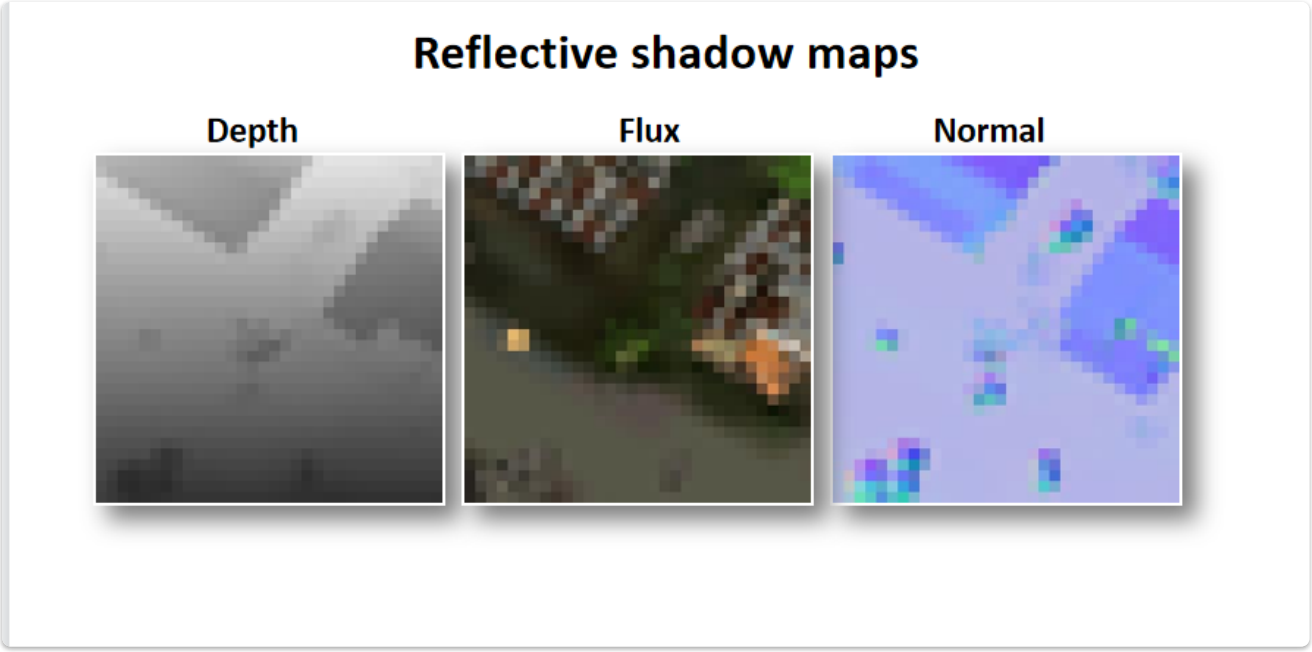
RSM	uniform
float4 worldPos : SV_Target0;	DXGI_FORMAT_R32G32B32A32_FLOAT
float4 normal : SV_Target1;	DXGI_FORMAT_R16G16B16A16_FLOAT
float4 flux : SV_Target2;	DXGI_FORMAT_R8G8B8A8_UNORM

顺便贴出PS的代码

```cpp
PSOutput PSRSM(VSOutput input)
{
PSOutput output;

output.worldPos = float4(input.worldPos, 1.0);
output.normal = normalize(float4(reflect(input.normal,LightDir.rgb),
0.0f));
output.flux = DiffuseColor * LightColor;

    return output;
}
```



图源自Cascaded Light Propagation Volumes for Indirect Illumination
ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games
21 February, 2010, Washington, USA

可以看见他是Depth我们用的是world pos

然后我们给出计算真正的RSM的代码，因为我们不是整个纹理对最终的像素进行贡献光照，而是对于先把最终的worldpos转换到这张texture上，再利用值

首先RSM是三张texture，那我们最后计算光照的时候采样哪一张呢，采样那张的多少东西呢？这是个问题
所以我们通过预先计算，把三张合成一张可以直接用的数据

先贴出代码：

```cpp
float3 CalculateRSM(float3 pos, float3 normal)
{
    float4 texSpacePos = mul(ShadowViewProjection, float4(pos, 1.0f));
    //透视除法，一般来说硬件会帮我们做，但是这里我们是手动变换
    texSpacePos.rgb /= texSpacePos.w;
          //从NDC的Xy坐标[-1,1]变换成[0,1](x,y)
    texSpacePos.rg = texSpacePos.rg * float2(0.5f, -0.5f) +
float2(0.5f, 0.5f);

    float3 indirectIllumination = float3(0.0, 0.0, 0.0);

    uint width = 0;
    uint height = 0;
    uint nol = 0;
    normalLSBuffer.GetDimensions(0, width, height, nol);

    for (int i = 0; i < RSM_SAMPLES_COUNT; i++)
    {
        float2 coord = texSpacePos.rg + RSMRMax * float2(xi[i].x *
sin(2.0f * PI * xi[i].y), xi[i].x * cos(2.0f * PI * xi[i].y));

        float2 texcoord = coord * float2(width, height);
        float3 vplPosWS = worldPosLSBuffer.Load(uint3(uint2(texcoord),
0)).rgb;
        float3 vplNormalWS = normalLSBuffer.Sample(RSMSampler,
coord).rgb;
        float3 flux = fluxLSBuffer.Load(uint3(uint2(texcoord),
0)).rgb;

        float3 vplPosDir = (pos - vplPosWS);

        float3 res = flux * ((max(0.0, dot(vplNormalWS, vplPosDir)) *
max(0.0, dot(normal, -vplPosDir))) / (dot(vplPosDir, vplPosDir) *
dot(vplPosDir, vplPosDir)));
        res *= xi[i].x * xi[i].x;

        indirectIllumination += res;
    }

    return indirectIllumination;
}
```

$$(s + r_{max}\xi_1 sin(2\pi\xi_2), t + r_{max}\xi_1 cos(2\pi\xi_2)) \quad (4)$$

(s,t) 是当前着色点的屏幕空间的坐标（UV坐标），我们可以通过生成多组（这个一组就代表多计算一次间接光照，有几个就相当于用多少个虚拟点光源），ξ1，ξ2为随机数（一组采样就需要一对随机数）代入上面的公式就可以得到一个采样坐标，然后再根据上面的光照计算公式来计算这个VPL对着色点的光照公式。注意：这种方法会有很大的误差，为了进行一些弥补，在每计算完一个虚拟点光源对着色点的光照之后应该再乘以权重 ξ_1^2 。

图来自[https://zhuanlan.zhihu.com/p/357259069]

这里有一个非常难以理解的点，我给大家理清一下思路：
我要最终的结果，我最终是延迟渲染，一个一个像素的去渲染那张四边形的quad，这张图的根本是GBuffer，我们也是根据坐标去索引GBuffer，所以这张图是跟GBuffer同一个视角的
这个时候是camera视角，对于一个像素，我肯定可以通过GBuffer知道它的Worldpos，既然知道了world pos那么我们就对这个我们要进行处理的像素进行变换，变换到光源空间的位置，变换到光源空间的位置以后，我们再对于这一点获取它在光源视角看来，周围的像素的光照贡献值。

既然是在光源视角看这一个坐标，那么我们就可以根据这个坐标算出来光源针对这一点坐标的flux, normal,pos

$$E_p(x,\vec{n}) = \frac{\max\{0, \langle \overrightarrow{n_p} | (x - x_p) \rangle\} \max\{0, \langle \vec{n} | (x_p - x) \rangle\}}{||x - x_p||^4},$$

这是在得到另一点对于它自己的贡献，得到夹角的平方然后再根据距离平方衰减
底下的四次方两次方是距离的平方衰减，两次是对于法线的归一化
最后结果还要乘以的权重为xi1^2

随机数的生成：

```cpp
RSMCBDataRandomValues rsmPassData2 = {};
                for (int i = 0; i < RSM_SAMPLES_COUNT; i++)
                {
                        rsmPassData2.xi[i].x = RandomFloat(0.0f,
1.0f);
                        rsmPassData2.xi[i].y = RandomFloat(0.0f,
1.0f);
                        rsmPassData2.xi[i].z = 0.0f;
                        rsmPassData2.xi[i].w = 0.0f;
                }
```

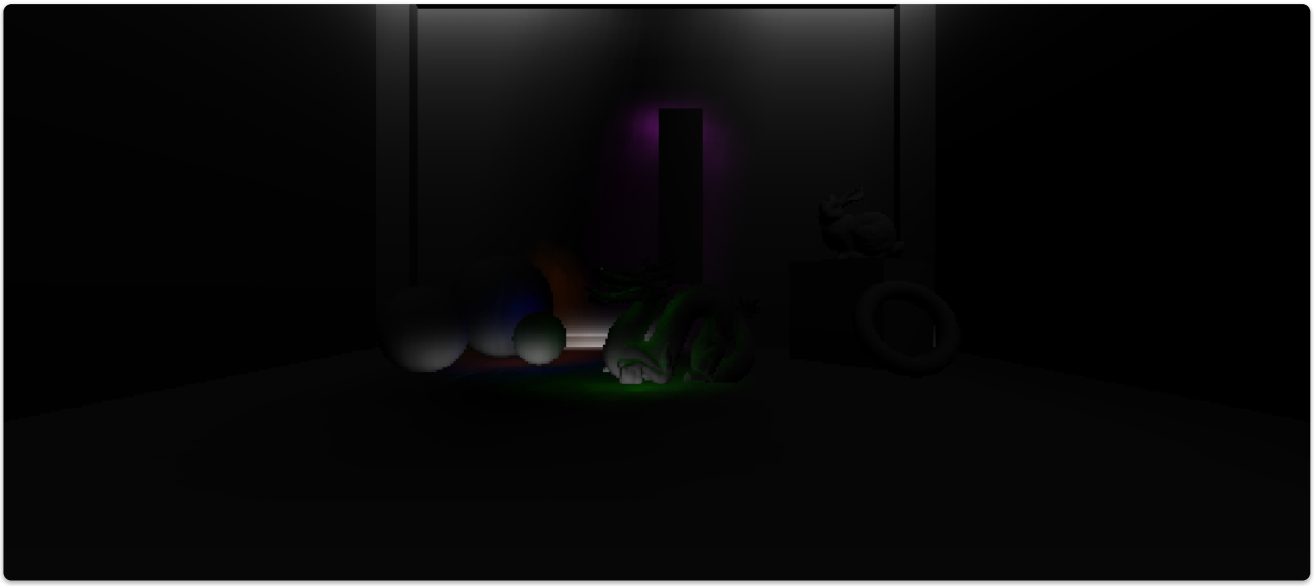
生成的结果记得saturate一下

然后就看看Lighting阶段如何去使用RSM

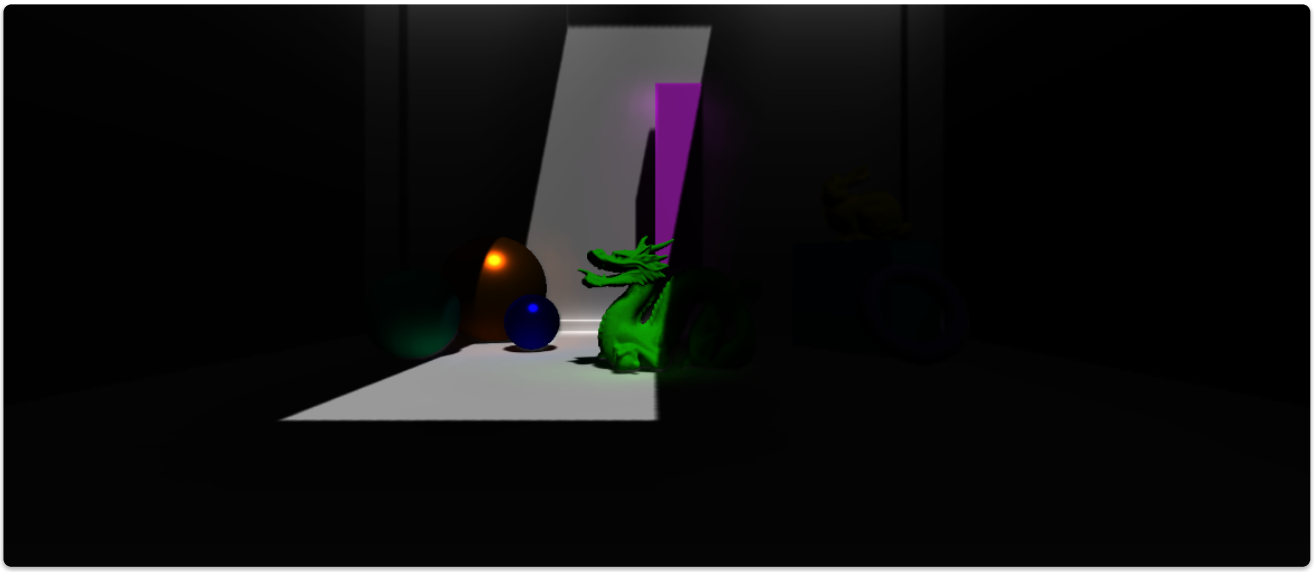
```cpp
// RSM
    if (useRSM)
    {
        uint gWidth = 0;
        uint gHeight = 0;
        albedoBuffer.GetDimensions(gWidth, gHeight);
        float3 rsm = rsmBuffer.Sample(BilinearSampler, inPos *
float2(1.0f / gWidth, 1.0f / gHeight)).rgb;

        indirectLighting += rsm * albedo.rgb * rsmGIPower;
    }
```

albedo源自于GBuffer，worldpos也源自于Gbuffer，rsmBuffer也是同GBuffer一视角的，所以根据同一input.pos得出RSM是非常合理的



RSM处理后的最终可利用值↑



成图

II. Reflective Shadow Mapping

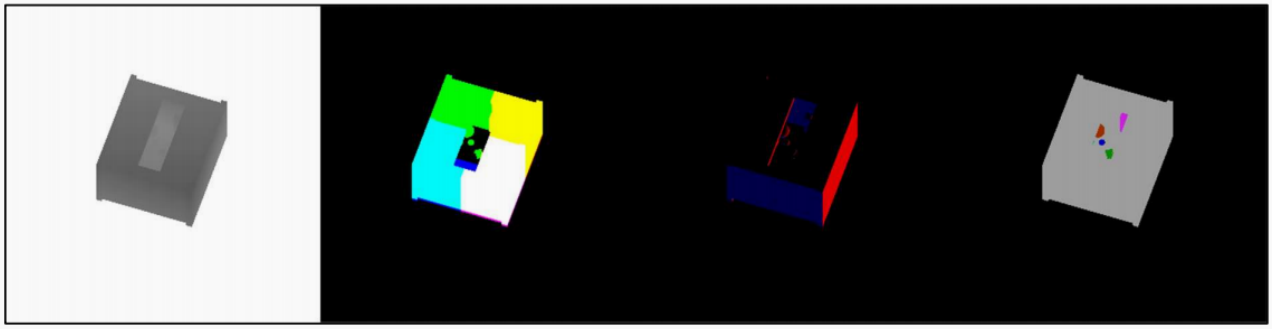


Figure 4. RSM textures of the scene from the directional light view: depth, world positions, normals, flux (in order)

源于原项目的论文