

D3D12 게임 프로젝트 로딩 성능 개선

유영천

<https://megayuchi.com>

<https://youtube.com/megayuchi>

Loading의 의미

- 통상적인 의미로는 저장소에 있는 데이터를 메모리에 Loading.
 - 말 그대로 저장소에서 데이터를 읽어서 메모리(일반적으로 System Memory)로 Load한다(fread(), ReadFile() 등 사용)

Loading의 의미

- 실제로는 더 많은 작업들이 포함됨
 - 필요한 메모리 미리 할당(Memory Pool 생성 등)
 - Shader Compile
 - 저장소에 cache데이터 설정
 - 자료구조 생성(공간 분할 Tree생성 등)
 - 정적 데이터를 로드해서 공간 자료구조에 삽입
 - 12 bytes vector -> 16 bytes vector로 변환...
 - 저장소에서 로드한 데이터(System Memory)를 GPU 메모리로 전송
 - D3D리소스 할당
 - Copy System Memory -> GPU Memory

어떤 작업이 제일 시간을 많이 잡아먹나?

- 데이터 로드 <- 광디스크 매체를 사용하던 시기에는 이게 가장 치명적이었음
- SSD가 일반화된 이후로는 데이터 로드는 그다지 치명적이지 않음.
- 경험적으로는 '데이터 가공'과 이 과정에서 메모리 할당/해제가 가장 치명적이었음.
- D3D12로 가면서 생각지도 못했던 복병이 생김
 - GPU 리소스 할당
 - GPU메모리로의 전송

OS와 그래픽 API에 상관없는 최적화 Tip

- 파일 여러 개를 읽지 말고, 다수의 파일들을 하나의 파일에 packing. 한번만 open하고 offset으로 왔다갔다 하면서 읽으면 훨씬 빠름(경험상 최소 2배 이상).
- 메모리 할당/해제를 줄이면 무조건 빨라짐.
- 특히 STL사용을 줄이면 빨라짐(부지불식간에 메모리 할당/해제가 빈번하게 발생함)
- 고정 사이즈 Memory Pool을 사용할 것.
 - Tree빌드할때 node메모리는 new/malloc으로 할당하지 말고 자체 Memory Pool을 사용할 것. 대체로 4-5배는 차이 남.

D3D 게임 프로젝트에서의 로딩 시간

- D3D11까지는 그래픽 API에 따라서 로딩 시간이 달라지는 상황은 상상해본 적도 없었다.
- 동일 엔진과 게임 프로젝트를 D3D11 -> D3D12로 포팅하자 로딩 속도가 10배 이상 느려짐.
- D3D API를 사용하는 레이어만 빼고 모든 코드가 동일한데 로딩 시간이 10배 이상 더 걸림.

D3D11에서의 리소스 생성 및 데이터 전송

```
HRESULT ID3D11Device::CreateBuffer(  
    [in] const D3D11_BUFFER_DESC* pDesc,  
    [in, optional] const D3D11_SUBRESOURCE_DATA* pInitialData,  
    [out, optional] ID3D11Buffer** ppBuffer  
);
```

- 버퍼를 채울 데이터의 포인터를 전달.
- 생성과 동시에 데이터 전송
- 함수가 리턴하면 모든 작업이 완료되었다고 믿으면 됨.
- 추측 - 함수가 빨리 리턴하는 이유 - GPU메모리 할당 자체도 비동기로 처리하고 곧바로 ID3D11Buffer리턴...하지 않을까. 리턴 시점에서 완전히 작업이 끝난건 아니지만 해당 리소스를 직접 사용할 때 까지만 끝나면 됨.

D3D12에서의 리소스 생성 및 데이터 전송

```
HRESULT CreateCommittedResource(  
    [in] const D3D12_HEAP_PROPERTIES* pHeapProperties,  
    [in] D3D12_HEAP_FLAGS HeapFlags,  
    [in] const D3D12_RESOURCE_DESC* pDesc,  
    [in] D3D12_RESOURCE_STATES InitialResourceState,  
    [in, optional] const D3D12_CLEAR_VALUE* pOptimizedClearValue,  
    [in] REFIID riidResource, [out, optional] void** ppvResource  
);
```

1. CreateCommittedResource() -> GPU메모리 생성
 2. CreateCommittedResource() -> Upload Buffer(시스템메모리) 생성
 3. Upload Buffer에 데이터 카피
 4. Upload Buffer -> GPU 버퍼로 카피(비동기 작업).
 5. Fence
 6. Wait Fence
 7. Upload Buffer제거
- Wait를 제거하고 사용된 Upload Buffer를 자동으로 제거하는 요령 -이 건은 오래전에 이미 해결했고 오늘의 주제는 아님.
https://www.youtube.com/live/IHIAloRa-HI?si=cC90Gkz_1ld_XIQK
https://github.com/megayuchi/ppt/blob/main/docs/2021_0518_D3D12%20%EB%A6%AC%EC%86%8C%EC%8A%A4%20%EA%B4%80%EB%A6%AC%20%EC%A0%84%EB%9E%B5.pdf

D3D12사용시 로딩이 느린 이유

- 리소스 데이터 전송 과정이 너무 길다.
 - 정확히는 async작업인 것이 문제. 작업 완료를 wait하면 엄청 느려진다. wait하지 않는 방법을 찾으면 문제되지 않는다.
- 리소스 할당이 느림.
 - CreateCommittedResource()가 진짜 느림.
 - 이건 생각도 못했다.
 - 나중에 판단하기로는 D3D12가 느린게 아니고 D3D11이 빠른 것이다.
 - GPU에 일을 시키는 방법이
커널 -> 드라이버 -> GPU 작동 -> 드라이버 -> 커널 -> 유저모드 프로세스
임을 감안하면 느린게 당연하다.

CreateCommittedResource 성능 보완 계획

- CreateCommittedResource() 함수는 프로그래머가 고칠 수 없음.
- 호출 회수를 줄인다.
- 버텍스 12개, 버텍스 16개, 버텍스 32개 생성해야하는데 12+16+32개
분의 버텍스 버퍼 한 개만 CreateCommittedResource() 로 할당하고
offset:0, offset:12, offset:28으로 사용한다. 이것으로 생성 속도 자체는
3배 빨라진다.
- D3D Resource Buffer Heap을 구현한다.
 - 큰 덩어리의 GPU메모리를 잡아놓고 그 안에서 malloc/free할 수 있다면
CreateCommittedResource() 호출을 극단적으로 줄일 수 있다.
 - 실제 물리 메모리를 점유하지 않고 주소만 리턴. 어드레스 범위는 0 - XXXXXX
 - HEAP에서 할당한 받은 주소를 GPU메모리의 offset으로 사용

D3D12 API의 특성

- 대부분의 draw/dispatch 및 SRV/CBV/UAV 생성 함수들은
파라미터로
ID3D12Resource*가 아닌 `D3D12_GPU_VIRTUAL_ADDRESS` 를 받음.
- ID3D12Resource*를 받는 경우도 FirstElement로 offset지정 가능.
- `D3D12_GPU_VIRTUAL_ADDRESS` p =
ID3D12Resource::GetGPUVirtualAddress() + offset

D3D12 API의 특성

```
typedef struct D3D12_VERTEX_BUFFER_VIEW
```

```
{  
    D3D12_GPU_VIRTUAL_ADDRESS BufferLocation;  
    UINT SizeInBytes;  
    UINT StrideInBytes;  
} D3D12_VERTEX_BUFFER_VIEW;
```

ID3D12Resource* pBuffer = ...
pBuffer->GetGPUVirtualAddress() + offset

```
void ID3D12GraphicsCommandList::IASetVertexBuffers(  
    [in] UINT StartSlot,  
    [in] UINT NumViews,  
    [in, optional] const D3D12_VERTEX_BUFFER_VIEW* pViews );
```

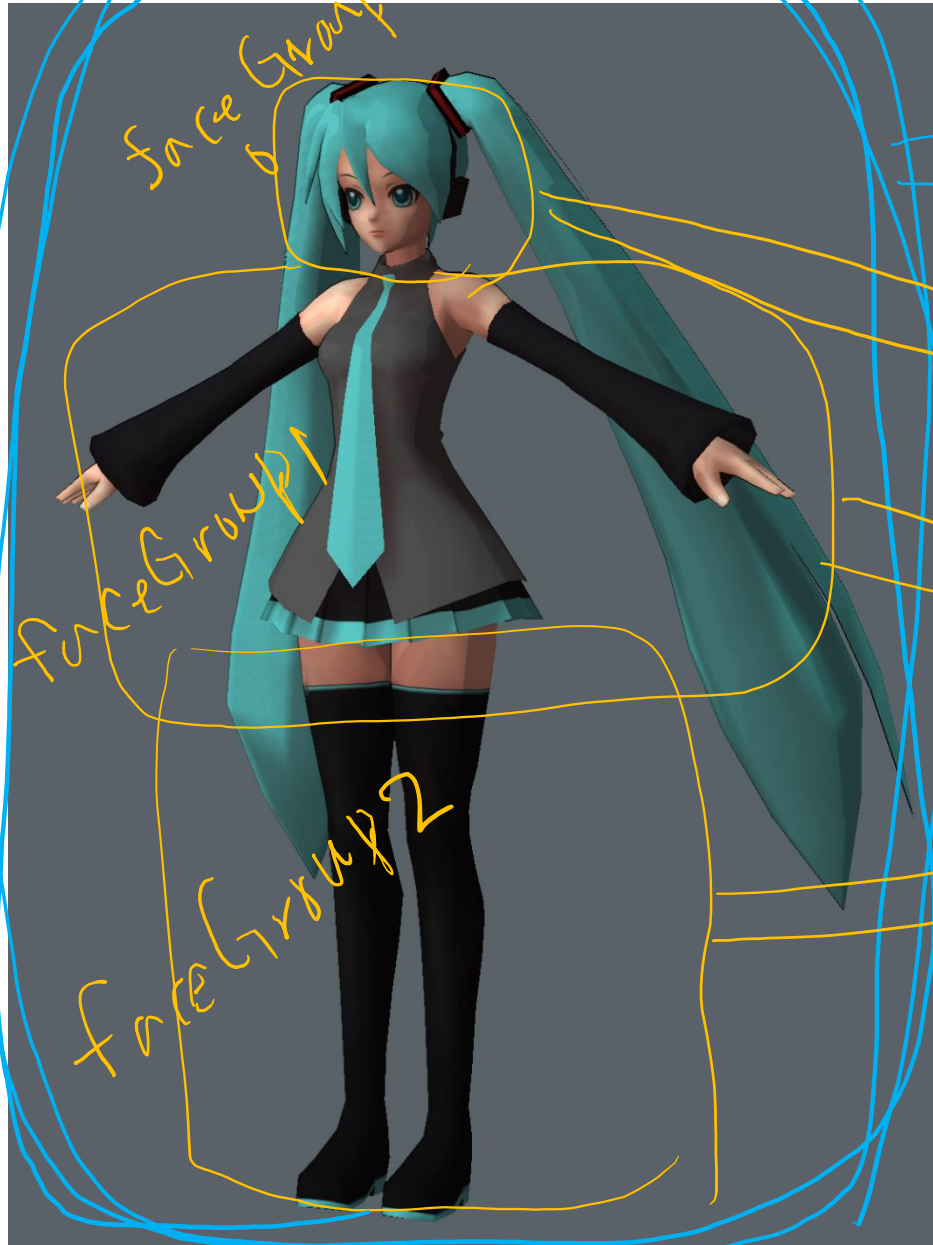
```
void ID3D12GraphicsCommandList::DrawIndexedInstanced(  
    [in] UINT IndexCountPerInstance,  
    [in] UINT InstanceCount,  
    [in] UINT StartIndexLocation,  
    [in] INT BaseVertexLocation,  
    [in] UINT StartInstanceLocation );
```

Offset

동시에 생성되고 해제되는 매시들 처리

- 정적인 삼각형 기반 맵에선 다수의 매시들이 한번에 생성 됐다가 한번에 제거됨.
- 한번에 생성될 매시들은 Vertex Buffer/ Index Buffer 하나를 쪼개서 사용할 수 있다.
- 데이터 -> 시스템 메모리로 로딩
- 필요한 크기의 D3DResource 할당
- 시스템 메모리 -> D3DResource로 한 번에 전송

동시에 생성되고 해제되는 매시들 처리



IASetVertexBuffer(Vertex Buffer 0)

Vertex Buffer 0(ID3D12Resource) – 64KB aligned

IASetIndexBuffer(Index Buffer 0);

Draw((Index Buffer 0)->GetGPUVirtualAddress());

Index Buffer 0(D3D12Resource 0) – 64KB aligned

IASetIndexBuffer(Index Buffer 1);

Draw((Index Buffer 1)->GetGPUVirtualAddress());

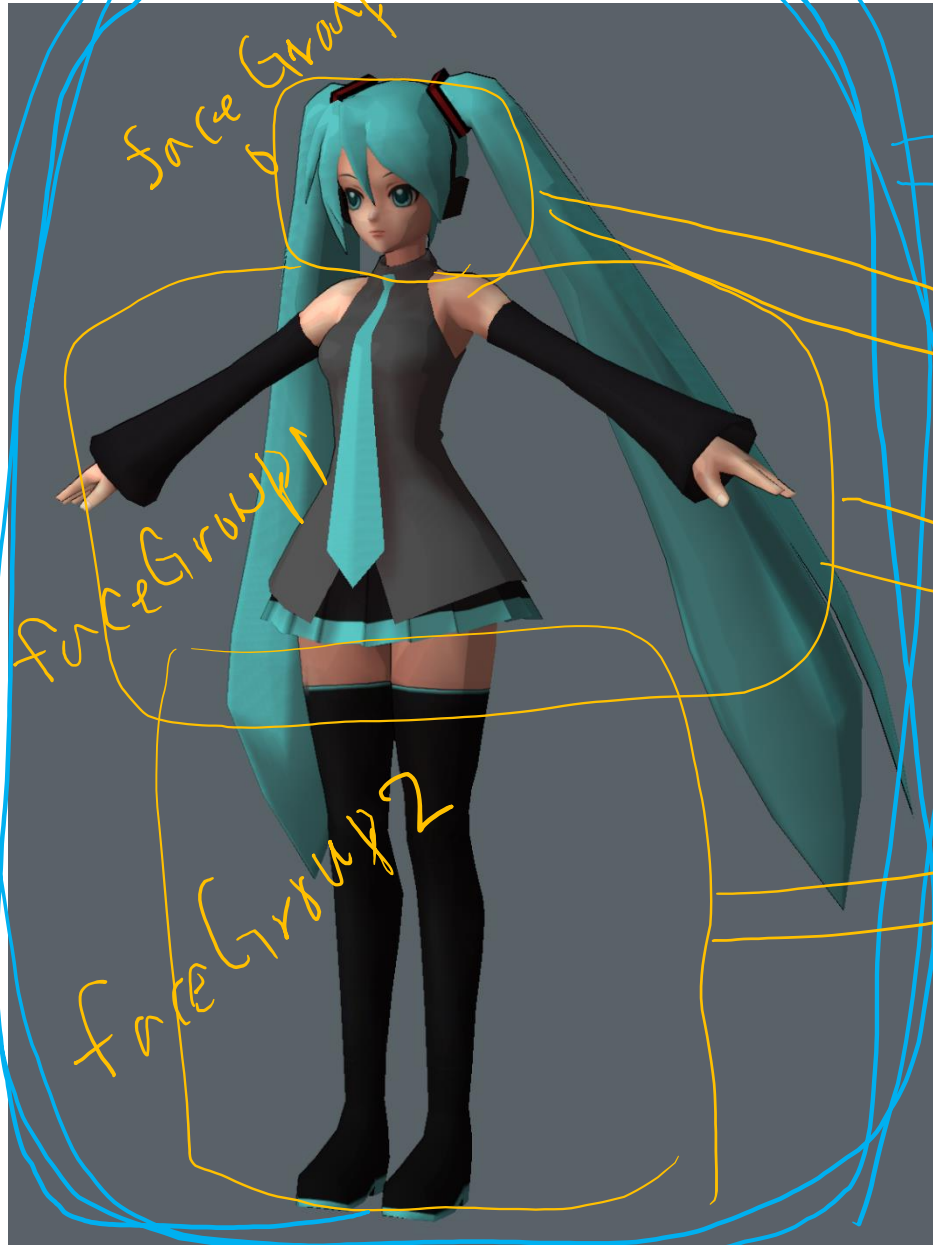
Index Buffer 1(D3D12Resource 1) – 64KB aligned

IASetIndexBuffer(Index Buffer 2);

Draw((Index Buffer 2)->GetGPUVirtualAddress());

Index Buffer 2(D3D12Resource 2) – 64KB aligned

동시에 생성되고 해제되는 매시들 처리



IASetVertexBuffer(Vertex Buffer 0)

IASetIndexBuffer(Index Buffer 0);

Vertex Buffer 0(ID3D12Resource) – 64KB aligned

Index Buffer 0(D3D12Resource 0) – 64KB aligned

Draw((Index Buffer 0)->GetGPUVirtualAddress() + Offset[0]);

Draw((Index Buffer 0)->GetGPUVirtualAddress() + Offset[1]);

Draw((Index Buffer 0)->GetGPUVirtualAddress() + Offset[2]);

1 Vertex Buffer per Object N Index Buffer per Object








VertexBuffer개수 : 3개
IndexBuffer 개수: 6개

- 오브젝트 #1
 - Vertex Buffer → VertexBuffer 0
 - FaceGroup #1(동일한 텍스처를 사용하는 삼각형 집합) → IndexBuffer
 - FaceGroup #2(동일한 텍스처를 사용하는 삼각형 집합) → IndexBuffer
- 오브젝트 #2
 - Vertex Buffer → VertexBuffer 1
 - FaceGroup #1(동일한 텍스처를 사용하는 삼각형 집합) → IndexBuffer
 - FaceGroup #2(동일한 텍스처를 사용하는 삼각형 집합) → IndexBuffer
 - FaceGroup #3(동일한 텍스처를 사용하는 삼각형 집합) → IndexBuffer
- 오브젝트 #3
 - Vertex Buffer → VertexBuffer 2
 - FaceGroup #1(동일한 텍스처를 사용하는 삼각형 집합) → IndexBuffer

1 Vertex Buffer per Object

1 Index Buffer per Object

VertexBuffer개수 : 3개
IndexBuffer 개수: 3개

- 오브젝트 #1 
 - FaceGroup #1(동일한 텍스처를 사용하는 삼각형 집합) 
 - FaceGroup #2(동일한 텍스처를 사용하는 삼각형 집합) 
- 오브젝트 #2 
 - FaceGroup #1(동일한 텍스처를 사용하는 삼각형 집합) 
 - FaceGroup #2(동일한 텍스처를 사용하는 삼각형 집합) 
 - FaceGroup #3(동일한 텍스처를 사용하는 삼각형 집합) 
- 오브젝트 #1 
 - FaceGroup #1(동일한 텍스처를 사용하는 삼각형 집합) 

1 Vertex Buffer per Map

1 Index Buffer per Map




VertexBuffer개수 :1개
IndexBuffer 개수: 1개



• 오브젝트 #1

- FaceGroup #1(동일한 텍스처를 사용하는 삼각형 집합) 
- FaceGroup #2(동일한 텍스처를 사용하는 삼각형 집합) 

• 오브젝트 #2

- FaceGroup #1(동일한 텍스처를 사용하는 삼각형 집합) 
- FaceGroup #2(동일한 텍스처를 사용하는 삼각형 집합) 
- FaceGroup #3(동일한 텍스처를 사용하는 삼각형 집합) 

• 오브젝트 #1

- FaceGroup #1(동일한 텍스처를 사용하는 삼각형 집합) 

동시에 생성되고 해제되는 매시들 처리

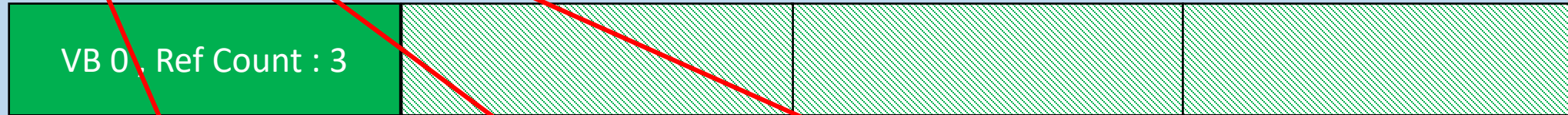
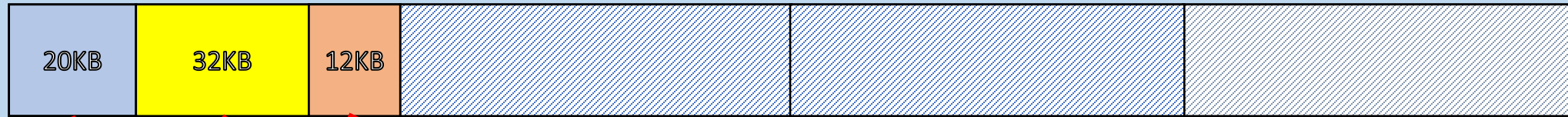
- 런타임에 생성/삭제 되는 매시에 대해서는 적용 불가.
- 범용적으로 사용할 수 없다.
- 우선 적용하면 좋지만 이것만으로는 부족하다.

D3D ResourceBufferHeap 구현

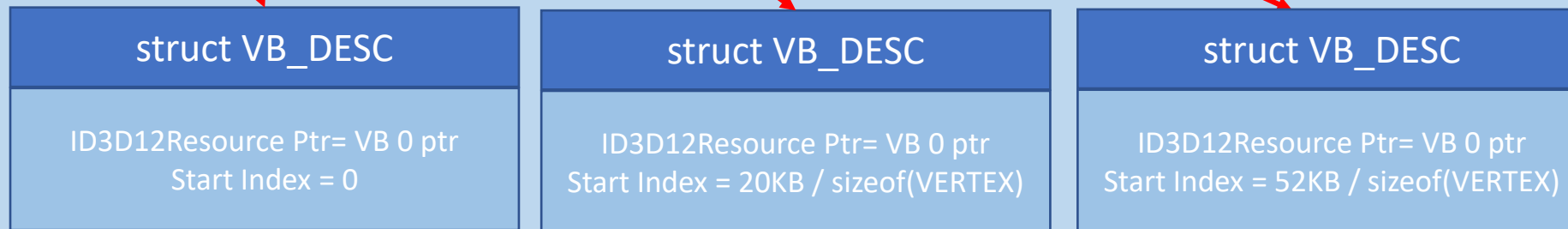
- CreateCommittedResource() 함수를 최소 호출하면서 D3D Resource를 malloc/free와 같은 방법으로 할당/해제하는 메모리 매니저를 만든다.
- 우선 실제 물리 메모리를 점유하지 않으면서 주소만 리턴하는 heap을 만든다.
- 일반적으로 heap구현시에는 할당되는 메모리 블록 안에서 앞뒤로 Head/tail 블록이 필요한데 GPU메모리에 head/tail을 써넣을 수 없으므로 블록의 정보를 기록한 구조체를 chain으로 연결한다.
- Heap에서 얻은 주소를 D3DResource 배열에 맵핑한다.

D3DResourceBufferHeap

Heap Address(Alloc/Free address only, addr range = $(0 - 65536 * N) - 1$ (ex: $N = 4$))



D3DBuffer(64KB)



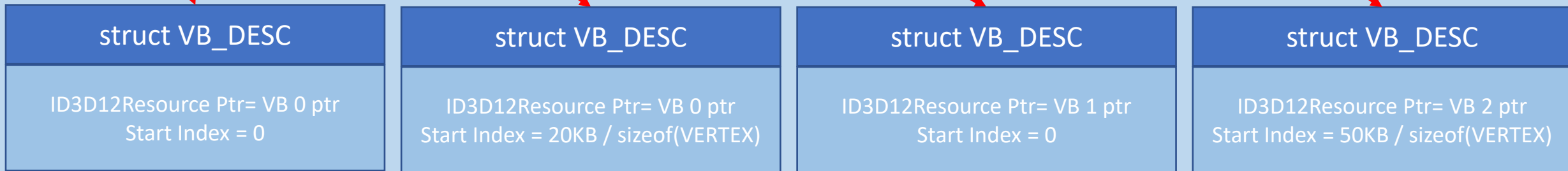
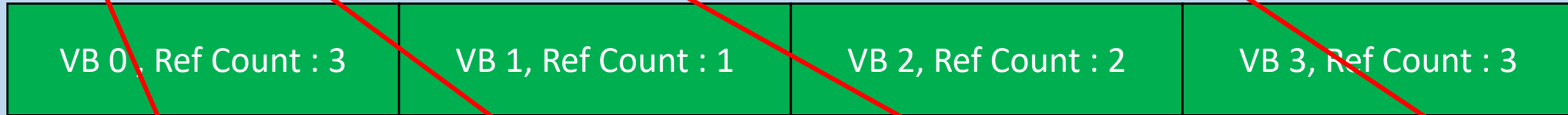
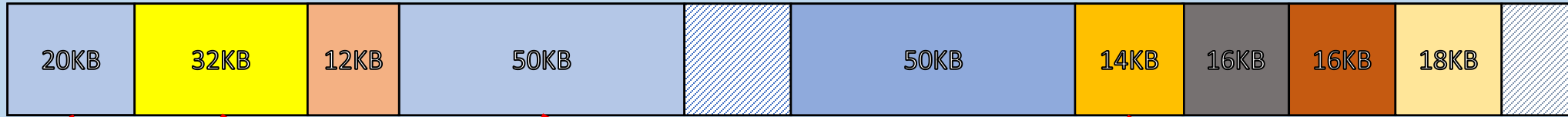
void* pHeapHandle

void* pHeapHandle

void* pHeapHandle

D3DResourceBufferHeap

Heap Address(Alloc/Free address only, addr range = $(0 - 65536 * N) - 1$ (ex: $N = 4$))



void* pHeapHandle

void* pHeapHandle

void* pHeapHandle

void* pHeapHandle

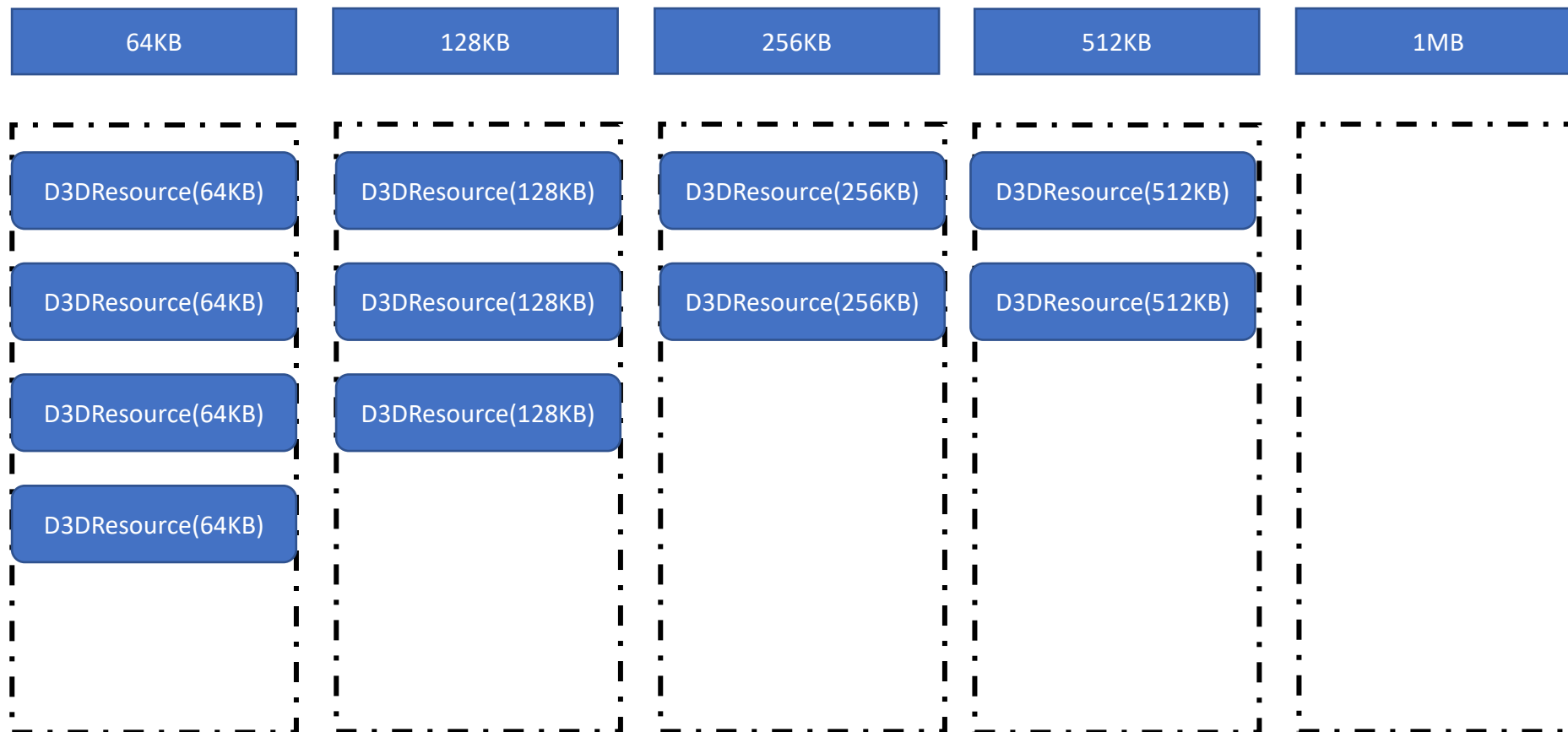
Heap대신 고정 사이즈 Pool구현

- GPU 메모리에 맵핑하는 Heap은 고사하고 그냥 Heap자체가 만들기 어렵다.
- 테스트 하기도 힘든데 엔진 코드에 적용 하려니 스스로의 간이 너무 작게 느껴진다.
- 정말 빨라질까? 정말 D3DResource 개수를 줄일 수 있을까? 효과가 있을지 먼저 검증부터 하고 싶다.

Heap대신 고정 사이즈 Pool구현

- 64KB, 128KB, 256KB...로 N개의 D3DResource를 할당. 사이즈 별로 Linked list로 연결해 둔다.
- 할당 요청이 들어오면 필요한 사이즈 이상의 버퍼를 리턴.
- 조건을 만족하는 버퍼가 없으면 새로 생성하고 리턴.
- 해제할 때는 Release()하는 대신 사이즈별 Link에 다시 연결해 둔다.
- 메모리 낭비가 좀 있지만 구현하기 쉽고 속도도 빠름.
- 잘 작동하면 인터페이스를 유지하고 내부구현을 가변 사이즈 Heap으로 바꾼다.

D3D Resource Manager-고정 사이즈 Pool구현

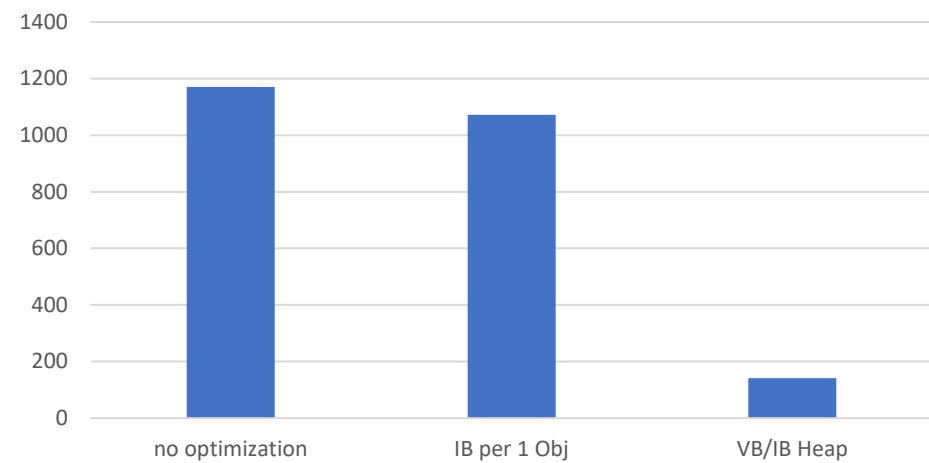


성능 테스트



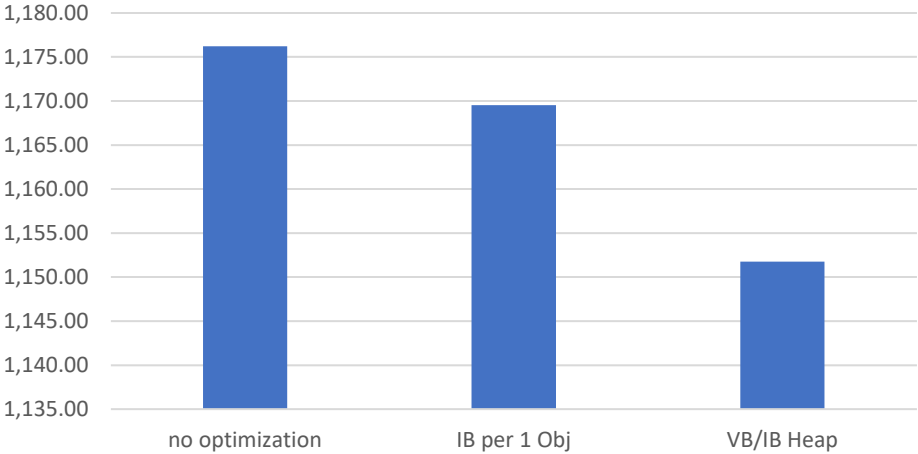
	no optimization	IB per 1 Obj	VB/IB Heap
로딩시간(ms)	1170.43	1072.34	141.22
GPU메모리 사용량(MiB)	1,176.21	1,169.53	1,151.77
Vertex Buffer 개수	4810	4666	12
Index Buffer 개수	3758	2333	10

로딩시간(ms)



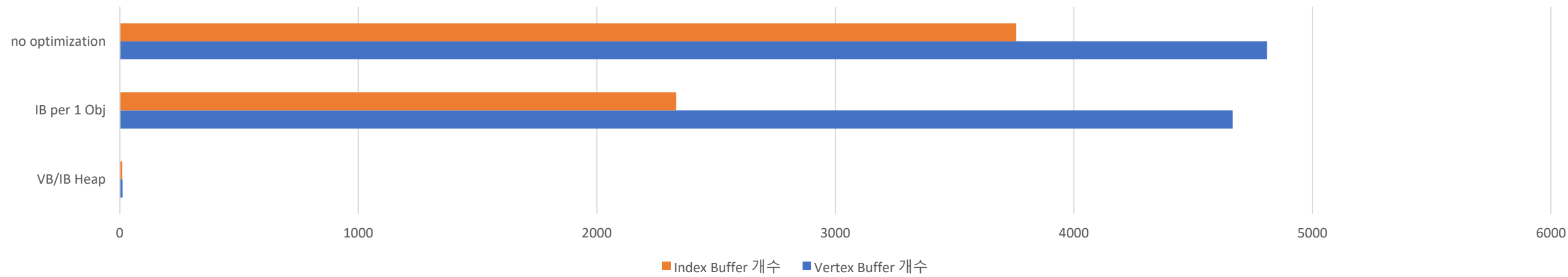
	no optimization	IB per 1 Obj	VB/IB Heap
로딩시간(ms)	1170.43 ms	1072.34 ms	141.22 ms

GPU메모리 사용량(MiB)



	no optimization	IB per 1 Obj	VB/IB Heap
GPU메모리 사용량	1176.21 MiB	1169.53 MiB	1151.77 MiB

Buffer 개수



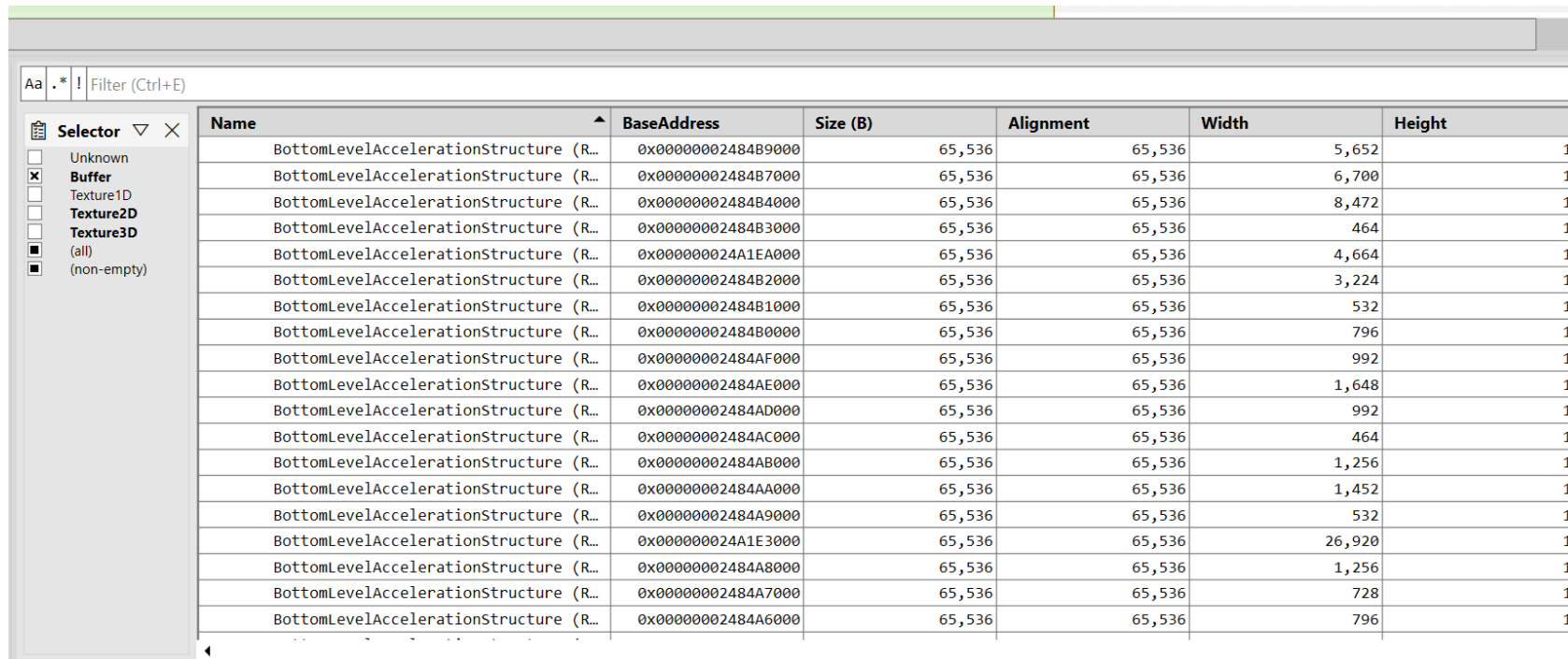
	no optimization	IB per 1 Obj	VB/IB Heap
Vertex Buffer 개수	4810	4666	12
Index Buffer 개수	3758	2333	10

D3D12만의 문제는 아니다!!!!

- VOXEL HORIZON iOS 빌드(Metal API)에서 문제 발생
 - 수만개의 복셀 오브젝트(buffer)와 오브젝트마다 붙어있는 라이트 텍스처(MTL::Texture)의 개수가 문제 됨.
 - 폰이 hang 됨.
 - 대량의 텍스처 -> 덩어리 텍스처를 Heap자료구조에 맵핑하는 방법(Texture Heap)
 - 텍스처 메모리 전체 사이즈가 절반 이하로 감소 -> 문제 해결!
- 어떤 API든 리소스 개수가 많으면 문제 될 수 있음.
- 어떤 API와 어떤 드라이버든 리소스 1개만 할당해도 실제 점유메모리와 요청 메모리 사이즈는 차이가 있음.
- 드라이버마다 처리 가능한 리소스 개수에 편차가 있음.

추후 작업과제

- DXR에서 사용하는 BLAS(Bottom Level Acceleration Structure)에는 아직 Heap을 적용하지 못함.
- BLAS도 heap으로 할당할 수 있다면 상당한 성능 향상을 기대함.



The screenshot shows a memory dump tool interface. On the left is a 'Selector' panel with checkboxes for 'Unknown', 'Buffer' (checked), 'Texture1D', 'Texture2D', 'Texture3D', '(all)', and '(non-empty)'. The main area is a table with the following columns: Name, BaseAddress, Size (B), Alignment, Width, and Height. The table contains 20 rows, all of which are 'BottomLevelAccelerationStructure' objects. Each row displays a unique base address, a size of 65,536 bytes, an alignment of 65,536, and varying width and height values.

Name	BaseAddress	Size (B)	Alignment	Width	Height
BottomLevelAccelerationStructure (R...	0x00000002484B9000	65,536	65,536	5,652	1
BottomLevelAccelerationStructure (R...	0x00000002484B7000	65,536	65,536	6,700	1
BottomLevelAccelerationStructure (R...	0x00000002484B4000	65,536	65,536	8,472	1
BottomLevelAccelerationStructure (R...	0x00000002484B3000	65,536	65,536	464	1
BottomLevelAccelerationStructure (R...	0x000000024A1EA000	65,536	65,536	4,664	1
BottomLevelAccelerationStructure (R...	0x00000002484B2000	65,536	65,536	3,224	1
BottomLevelAccelerationStructure (R...	0x00000002484B1000	65,536	65,536	532	1
BottomLevelAccelerationStructure (R...	0x00000002484B0000	65,536	65,536	796	1
BottomLevelAccelerationStructure (R...	0x00000002484AF000	65,536	65,536	992	1
BottomLevelAccelerationStructure (R...	0x00000002484AE000	65,536	65,536	1,648	1
BottomLevelAccelerationStructure (R...	0x00000002484AD000	65,536	65,536	992	1
BottomLevelAccelerationStructure (R...	0x00000002484AC000	65,536	65,536	464	1
BottomLevelAccelerationStructure (R...	0x00000002484AB000	65,536	65,536	1,256	1
BottomLevelAccelerationStructure (R...	0x00000002484AA000	65,536	65,536	1,452	1
BottomLevelAccelerationStructure (R...	0x00000002484A9000	65,536	65,536	532	1
BottomLevelAccelerationStructure (R...	0x000000024A1E3000	65,536	65,536	26,920	1
BottomLevelAccelerationStructure (R...	0x00000002484A8000	65,536	65,536	1,256	1
BottomLevelAccelerationStructure (R...	0x00000002484A7000	65,536	65,536	728	1
BottomLevelAccelerationStructure (R...	0x00000002484A6000	65,536	65,536	796	1