DirectX 12 프로그래밍에 대한 고찰

유영천

https://megayuchi.com

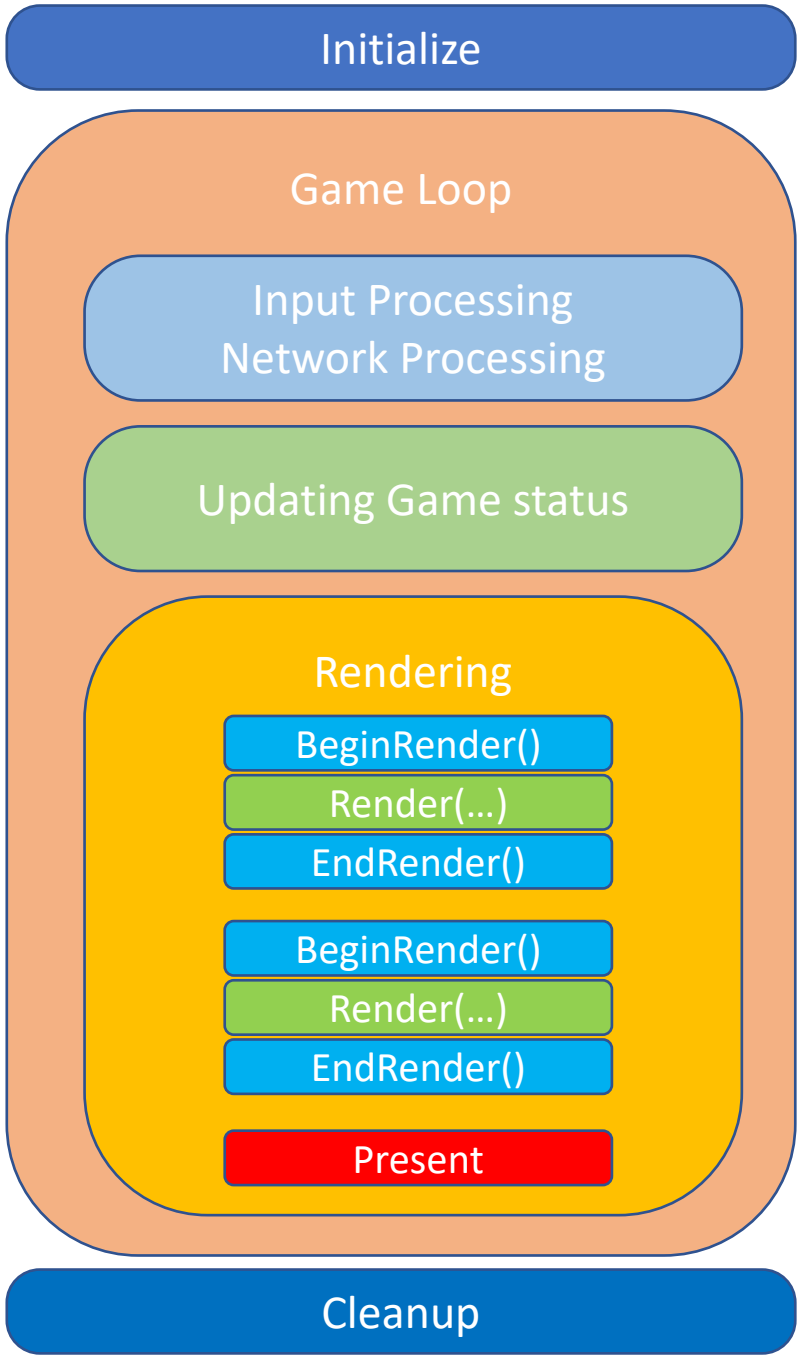
Tw: @dgtman

DirectX 12를 해야하는가?

- YES! YES! YES!
- 기존 rasterization만 사용할거라면 D3D11 -> D3D12로 넘어갈 필요가 없다.
- 하지만 Raytracing을 사용할거라면 선택의 여지가 없다.
- Raytracing으로 가기 위해서 D3D12를 해야한다.
- D3D11 -> D3D12/DXR로 바로 간다해도 D3D12의 rasterization 기술은 여전히 필요하다.

렌더링 스케쥴링

Game Loop

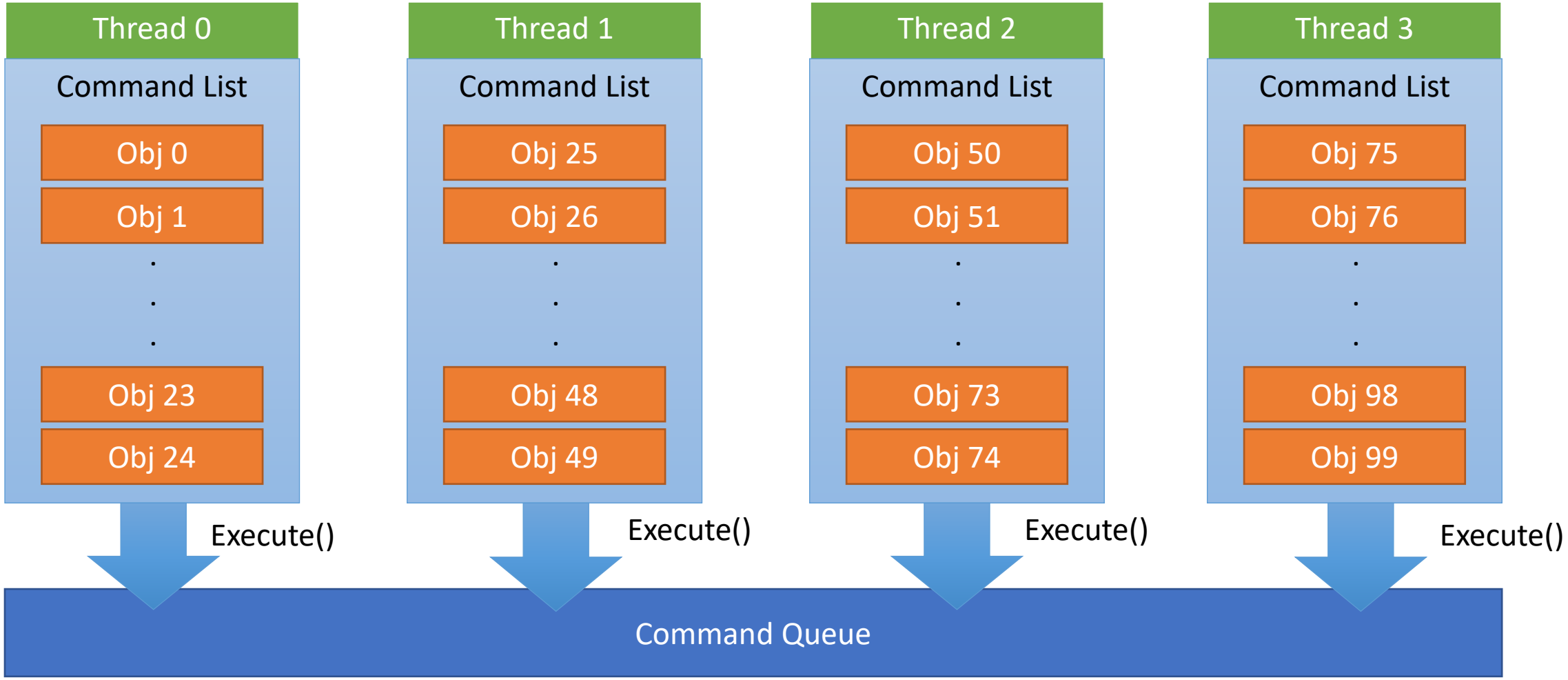


- BeginRender() 과 EndRender()사이에 draw/dispatch
- Present()호출 후 wait

멀티 스레드 렌더링

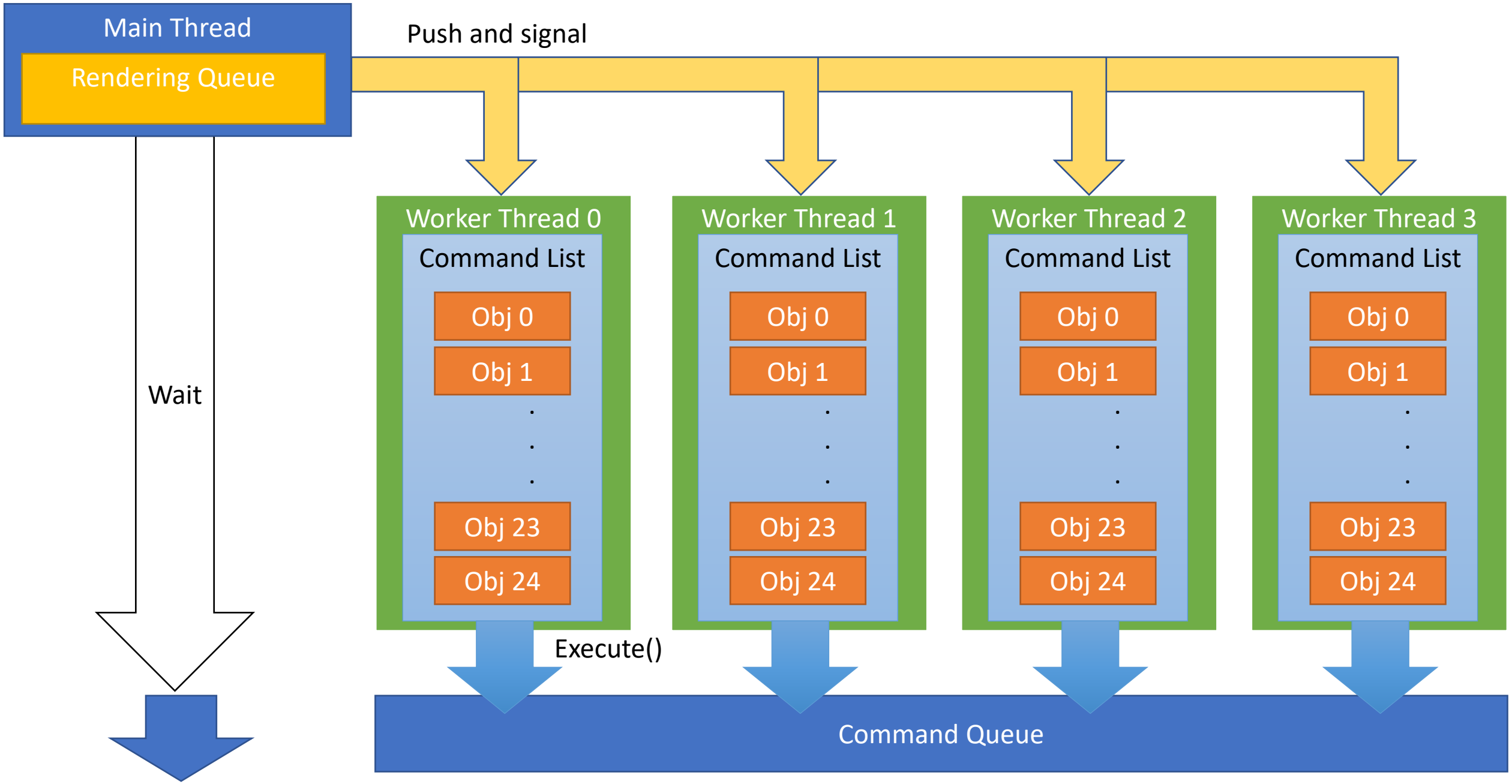
- D3D12에서 멀티 스레드 렌더링은 필수.
- 더 빠른 속도를 위해서 위해서 (x)
- 말이 되는 속도를 얻기 위해서(o)

Multi-Thread Rendering 구현



Blocking 방식

- 렌더링할 오브젝트를 Queue에 쌓아놨다가 일괄처리
- 단 시간 내에 GPU파워를 최대한 활용

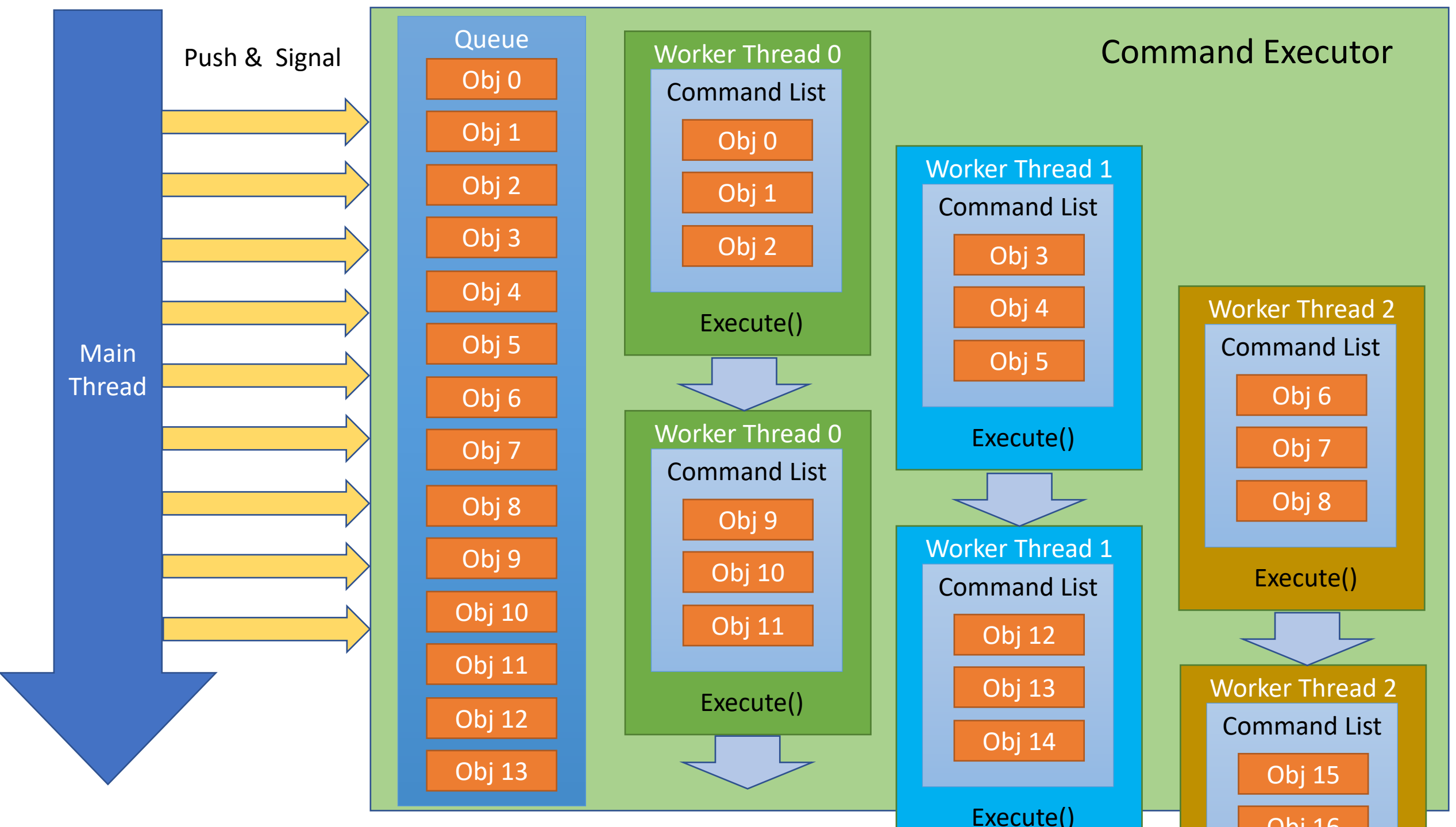

Main Thread
Rendering Queue
Push and signal
Wait
Worker Thread 0
Command List
Obj 0
Obj 1
Obj 23
Obj 24
Worker Thread 1
Command List
Obj 0
Obj 1
Obj 23
Obj 24
Worker Thread 2
Command List
Obj 0
Obj 1
Obj 23
Obj 24
Worker Thread 3
Command List
Obj 0
Obj 1
Obj 23
Obj 24
Execute()
Command Queue

CPU Timeline

GPU Timeline

Non-blocking 방식

- Queue를 사용하지 않고 메인스레드의 요청에 따라 그 즉시 렌더링
- 전체적으로 균등하게 GPU파워를 활용

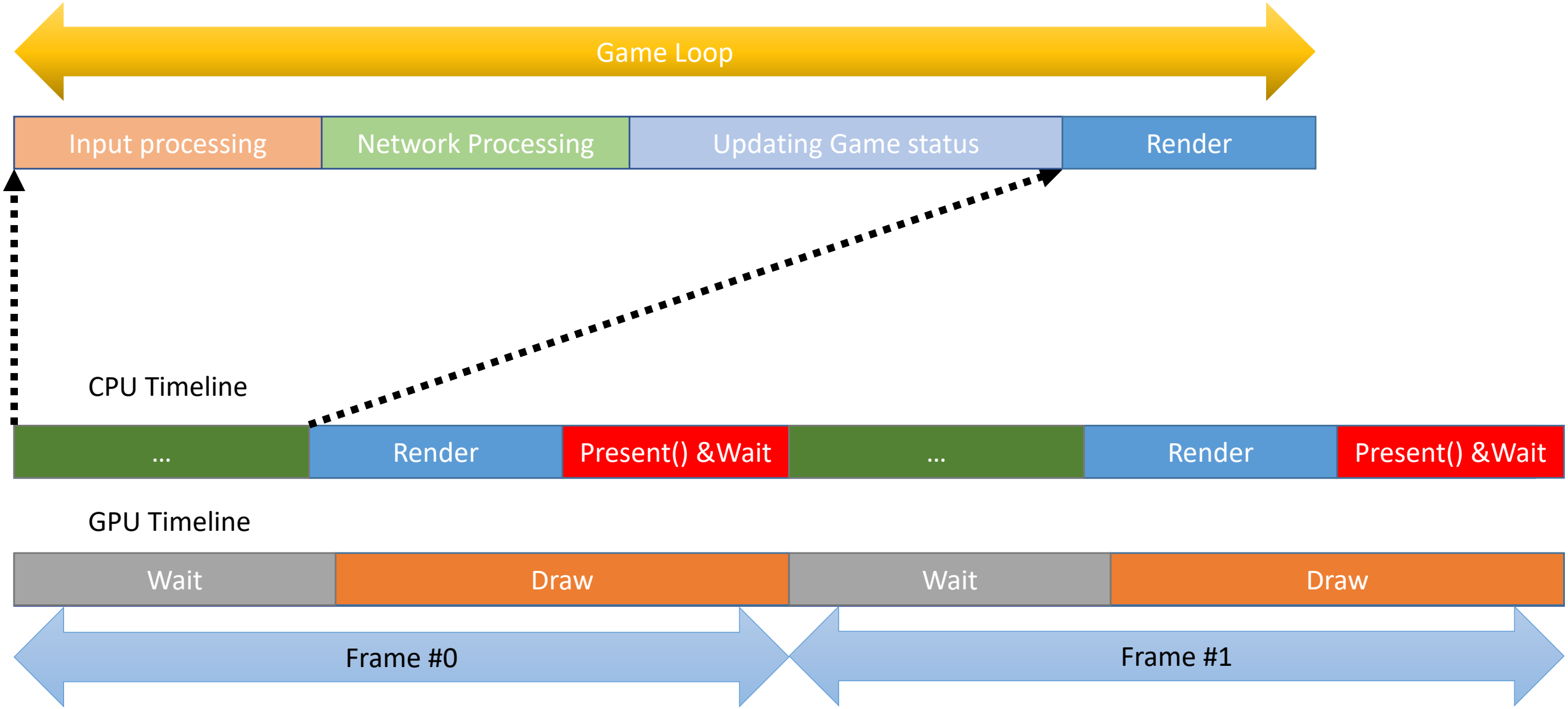

Main Thread
Push & Signal
Queue
Obj 0
Obj 1
Obj 2
Obj 3
Obj 4
Obj 5
Obj 6
Obj 7
Obj 8
Obj 9
Obj 10
Obj 11
Obj 12
Obj 13
Command Executor
Worker Thread 0
Command List
Obj 0
Obj 1
Obj 2
Execute()
Worker Thread 0
Command List
Obj 9
Obj 10
Obj 11
Execute()
Worker Thread 1
Command List
Obj 3
Obj 4
Obj 5
Execute()
Worker Thread 1
Command List
Obj 12
Obj 13
Obj 14
Execute()
Worker Thread 2
Command List
Obj 6
Obj 7
Obj 8
Execute()
Worker Thread 2
Command List
Obj 15

CPU Timeline

GPU Timeline

- 현재는 Non-blocking방식을 사용하지만 항상 이 방식이 빠르지는 않다.
- 여전히 만족스럽지 못하다.
- 한 프레임 내에서의 스케쥴링 방식의 변화는 한계가 있다.

게임 루프 내에서의 timeline



비동기식 중첩 렌더링

- 그야말로 비동기 렌더링
- CPU타임라인과 GPU타임라인을 아예 일치시키지 않는다.
- 현재프레임 – 1, 또는 현재 프레임 -2의 fence 값을 wait한다.
- 리소스를 제거하거나 변경할때 동기화가 필요하다. 이 시점에선 pending된 렌더링 커맨드에 대한 wait가 필요하다.
- 동시에 중첩시킬 프레임 수만큼의 Constant Buffer, Descriptor heap 등등의 리소스가 필요하다.

게임 루프 내에서의 timeline

CPU Timeline

Present & wait #0

Present & wait #1

Present & wait #2

GPU Timeline

비동기식 중첩 렌더링

- 멀티 스레드로 충분히 머리 아픔.
- 중첩 렌더링 하면 x2 , x4로 머리 아픔.
- 돌아가도 맘 편하게 못잔다. 동기화에 구멍이 없는걸까?
- GPU 성능 > CPU 성능인 경우의 테스트
- GPU 성능 < CPU 성능인 경우의 테스트

렌더링 스레드 개수

- 많을수록 좋다? (x)
- 처리량 때문에 응답성이 떨어지는 경우에만 스레드 개수를 늘릴 필요가 있다.
- 현대 CPU의 Turbo boost특성상 스레드가 많아지면 개별 클럭은 떨어짐 -> 성능저하의 주요요인
- 가볍든 무겁든 스레드간 동기화가 필요하다-> 대기 시간 발생
- 많다고 능사가 아님. 절대로.
- 경험상 물리 코어 개수를 절대 넘기지 않는 편이 좋다.

성능 테스트

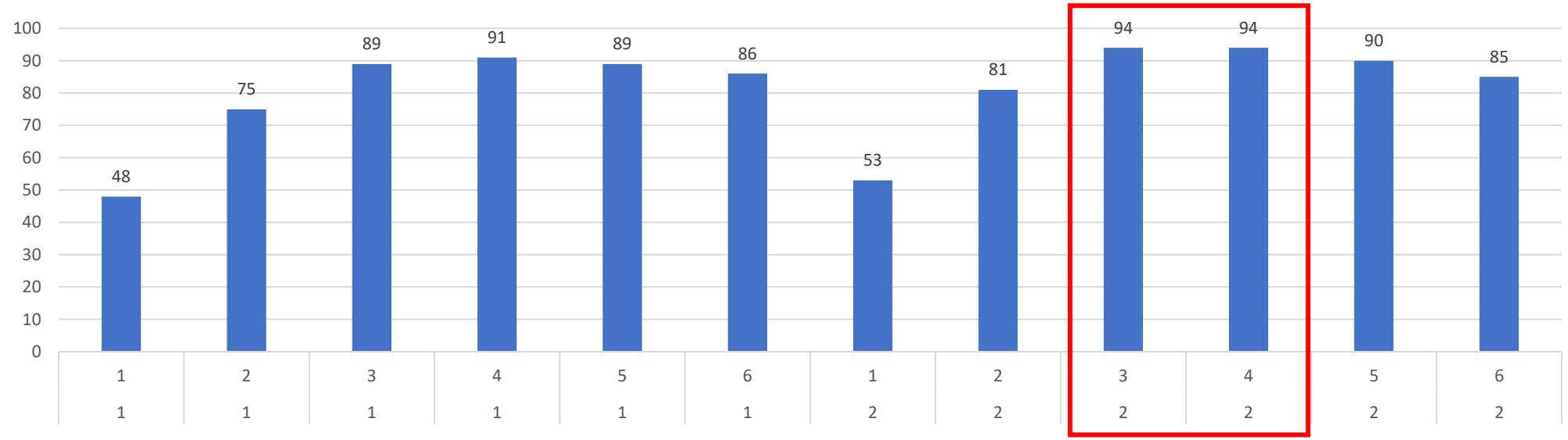
Rasterization – D3D11

- Triangle Map
- Res : 1200x800
- HW Occ off , SW Occ off

- Singled thread
- No nested frames
- GPU usage : 89%
- 479 FPS

Rasterization – D3D12

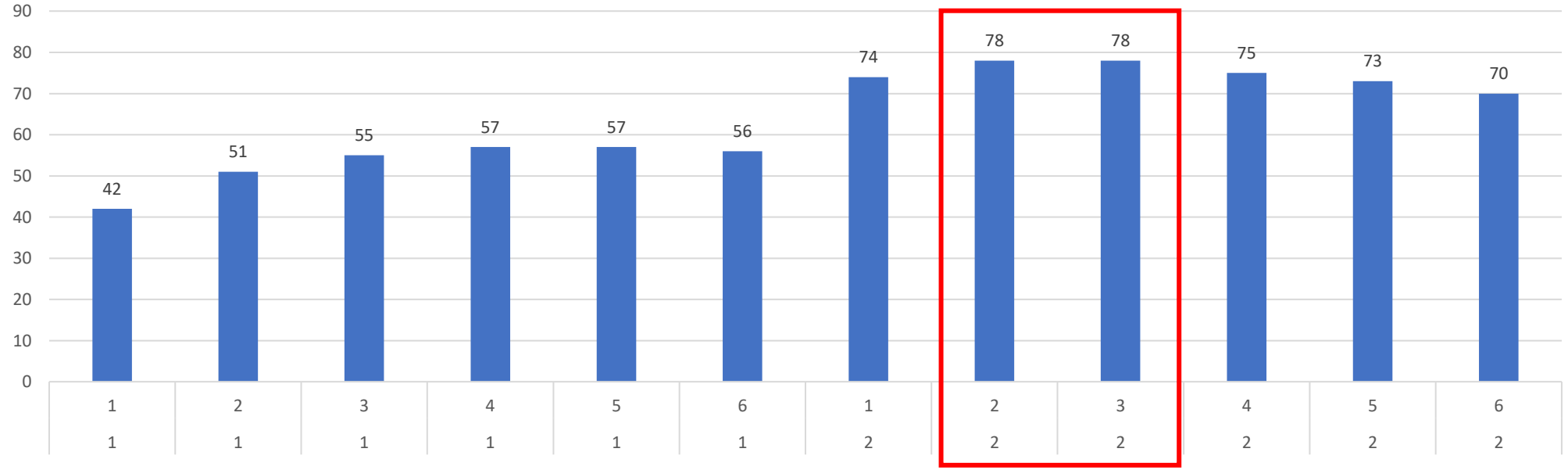
- Triangle Map
- Res : 1200x800
- HW Occ off , SW Occ off



D3D12 , Res : 1200x800 , Raster , HW Occ off , SW Occ off												
nested frames	1	1	1	1	1	1	2	2	2	2	2	2
threads	1	2	3	4	5	6	1	2	3	4	5	6
GPU usage	30%	38%	38%	39%	39%	43%	27%	31%	38%	39%	39%	39%
FPS	48	75	89	91	89	86	53	81	94	94	90	85

성능 테스트 - Raytracing

- Triangle Map
- D3D12/DXR
- Res : 1200x800
- HW Occ off , SW Occ off



D3D11 , Res : 1200x800 , DXR, No RTAO , HW Occ off , SW Occ off												
nested frames	1	1	1	1	1	1	2	2	2	2	2	2
threads	1	2	3	4	5	6	1	2	3	4	5	6
GPU usage	52%	64%	70%	72%	74%	74%	91%	96%	96%	96%	95%	94%
FPS	42	51	55	57	57	56	74	78	78	75	73	70

Fence & Wait전략

- 중첩 렌더링을 하더라도 언젠가는 한번 이상 wait를 해야한다.
- 리소스의 변경, 해제 시 필요하다.
- GPU작업만으로 변경이 일어나는 경우는 Resource Barrier만으로 동기화 가능.
- 최대한 적게, 최대한 늦게.

여러 개의 Command Queue

- 경험적으로는 성능 차이 거의 없었다.
- 순서보장 안됨.
- 코드 매우 복잡해짐.

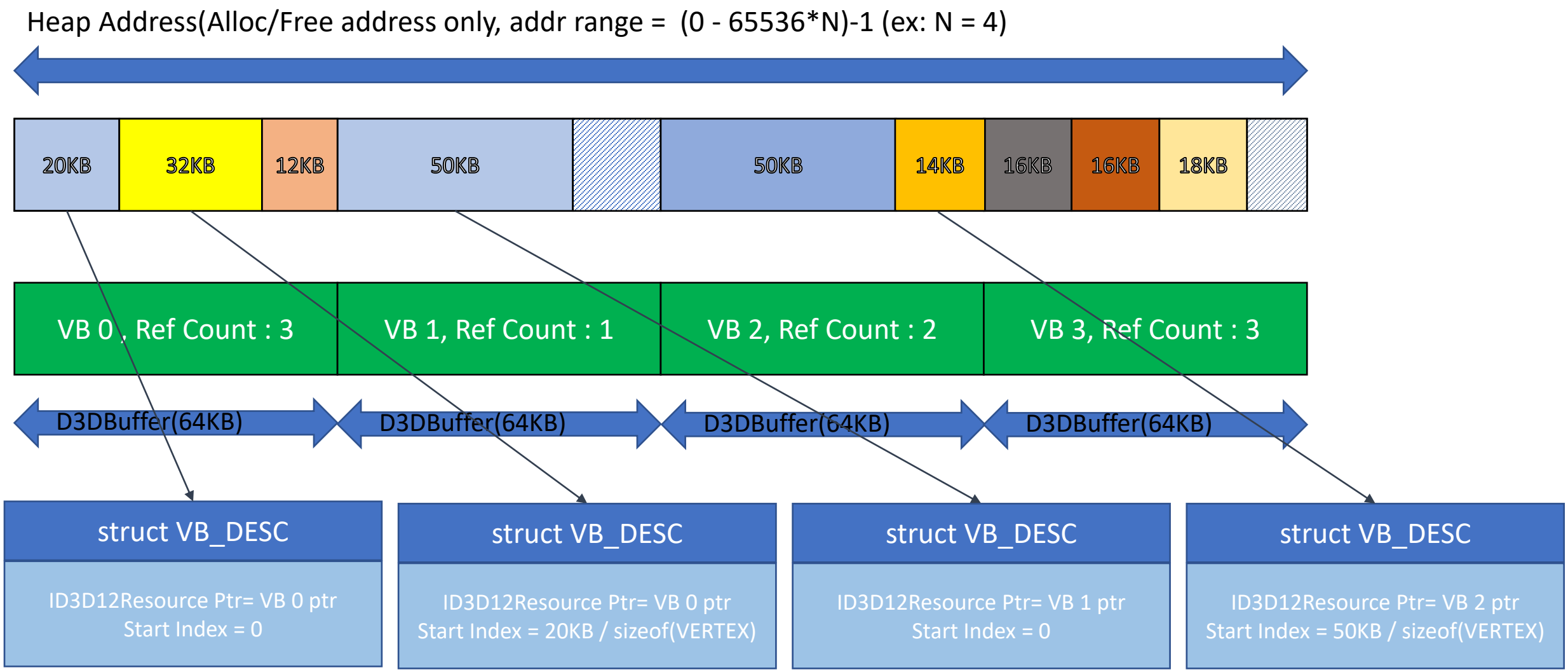
RTX GPU vs GXT GPU

- DXR에서의 성능차이

Voxel world / 1920x 1200	GPU Usage	FPS
CPU : i7 8700K (6 Cores) , GB5 score : 1169(single) / 6111(multi) GPU: GTX1660Ti (SMs : 25, Cuda Cores : 1536 , RT Cores : 0)	99%	27
CPU : AMD EPYC 7V12 (4 Cores) , GB5 score : 961(single) / 3448(multi) GPU : Tesla T4 (SMs : 40 , Cuda Cores : 2560, RT Cores: 40)	96%	94
Triangle map / 1920x 1200		
CPU : i7 8700K (6 Cores) , GB5 score : 1169(single) / 6111(multi) GPU : GTX1660Ti (SMs : 25, Cuda Cores : 1536 , RT Cores : 0)	99%	33
CPU : AMD EPYC 7V12 (4 Cores) , GB5 score : 961(single) / 3448(multi) GPU : Tesla T4 (SMs : 40 , Cuda Cores : 2560, RT Cores: 40)	81%	82

Resource align

- 32bytes짜리 Vertex buffer한개를 만들어도 64KB소모.
- 1x1짜리 Texture하나를 만들어도 64KB소모
- VertexBuffer와 IndexBuffer가 GPU메모리를 낭비하는 주원인이 된다.
- 라이트맵을 사용하거나 복셀지형을 만들게 되면 Texture의 64KB align도 상당히 문제가 된다.
- 버퍼 하나를 크게 잡아놓고 안에서 쪼개써야할 필요가 있다.


Heap Address(Alloc/Free address only, addr range = (0 - 65536*N)-1 (ex: N = 4)
20KB
32KB
12KB
50KB
50KB
14KB
16KB
16KB
18KB
VB 0 , Ref Count : 3
VB 1, Ref Count : 1
VB 2, Ref Count : 2
VB 3, Ref Count : 3
D3DBuffer(64KB)
D3DBuffer(64KB)
D3DBuffer(64KB)
D3DBuffer(64KB)
struct VB_DESC
ID3D12Resource Ptr= VB 0 ptr
Start Index = 0
struct VB_DESC
ID3D12Resource Ptr= VB 0 ptr
Start Index = 20KB / sizeof(VERTEX)
struct VB_DESC
ID3D12Resource Ptr= VB 1 ptr
Start Index = 0
struct VB_DESC
ID3D12Resource Ptr= VB 2 ptr
Start Index = 50KB / sizeof(VERTEX)

- Heap자료구조를 만든다.
  - Alloc(size N)을 호출하면 메모리를 할당해주는게 아니고 base address를 주면 그에 대한 상대주소를 리턴한다.
  - 다양한 사이즈를 할당할 수 있어야한다.
  - 해제시 인접 블록이 해제된 상태라면 병합할 수 있어야 한다.
- Heap의 어드레스 범위는 0 – D3DResource의 size * 버퍼의 최대 개수
  - 예를 들어 D3DResource(Buffer)를 64KB씩 할당해서 최대 1024개를 사용한다면 heap의 addres범위는 0 – 65536*1024이다.
- D3DResource의 최대개수만큼의 ptr 배열을 만든다.
- Vertex Buffer or Index Buffer를 할당할때 Heap->Alloc() 호출
- 이렇게 얻은 address – base addr(일반적으로 0)로 상대 address를 얻는다.상대 address로 맵핑될 D3DResource의 index를 결정한다.
- 선택된 D3DResource의 ptr배열이 null인지 체크. Null이면 D3D API를 사용해서 D3DResource(Buffer)를 만든다.
- Null이 아닌 경우 D3DResource의 ref count증가.
- VB_DESC에 D3DResource의 ptr을 설정.
- 상대 address를 이용해서 VB_DESC에 StartIndex도 설정.
- Draw…()호출할때 VB_DESC의 D3DResource의 ptr과 StartIndex를 파라미터로 넣어준다.