D3D12프로그래밍 소개

유영천

https://megayuchi.com

https://youtube.com/megayuchi

DirectX?

- MS의 Graphics API
- 초기에는 Windows 95에서 GDI를 대신해 그래픽 카드 메모리로의 직접 연결통로를 만들어주기 위한 API였음.
  - DirectDraw가 메인 구성요소, 이후에 Direct3D가 추가 됐음.
  - 입력/사운드/네트워크 기능 등이 추가
  - 현재 D3D를 제외한 다른 구성요소들은 업데이트가 중단된 상태.
  - 현재 실질적으로 남은건 D3D11/D3D12 (그래픽스 – D3D11/12 , 사운드 – XAudio2, 입력 XInput)
- 대부분의 Windows 게임과 100%의 XBOX게임이 DirectX를 사용
- 게임 수로 보면 DX9 >= DX11 >>> DX12

그래픽 API소개

- Direct3D in DirectX의 변화
  - Direct3D 3/5/6/7 - 다분히 OpenGL스러운 고전적인 디자인
  - Direct3D 8 - 최초의 Shader지원. XBOX Original의 기본 API.
  - Direct3D 9 - shader 지원 본격화. 이전까지 D3D의 완성형
  - Direct3D 10 - 다소HW친화적인 디자인. 바뀐 GPU드라이버 모델에 따라 device lost처리의 난해함/어드레스 공간 제약이 사라짐. Windows Vista 망하면서 같이 망함.
  - Direct3D 11 - Tessellation/Compute Shader지원. 아직까지 쌩쌩한 현역.
  - Direct3D 12 - Windows 진영의 차세대 API. 망하는듯 했으나 Raytracing지원으로 화려하게 부활.

DirectX 12 -> DirectX 12 Ultimate

- DirectX Raytracing지원
- Mesh shader지원
- Direct Storage 지원
- Agility SDK지원

D3D12의 문제점

- 거의 대부분의 내장GPU에선 정상 작동 안함.
  - 정상작동 하는듯 해도 한계상황에 가면 크래시하거나 화면이 깨지거나 하는 일이 다반사
- 이론상 GPU가 DX Feature Level 11을 지원하면 돌아는 가지만 실제로는 nvidia Maxwell이전 세대 GPU에서 크~게 느리거나 정상작동 하지 않는다.
- 코드 양이 적어도 4배 이상 많아진다. 체감 10배.
- 굉장히 신경을 쓰지 않으면 시스템 메모리와 GPU메모리 사용량이 50%이상 더 소모된다. 상당한 노력을 기울여야 겨우 D3D12엔진보다 조금 덜 먹는 수준까지 맞출 수 있다.

DX12성능 – 유명 게임 벤치마크

- Quantum Break – DX11 > DX12

  https://www.computerbase.de/2016-09/quantum-break-steam-benchmark/3/#diagramm-quantum-break-3840-2160-fx-8370

- Battle Field 1 – DX11 >= DX12

  http://www.gamestar.de/spiele/battlefield-1/artikel/battlefield_1_open_beta,53468,3301876.html

- F1 2020 – nvidia DX11 >= DX12 , AMD DX12 >=DX11

  https://www.guru3d.com/articles-pages/f1-2020-pc-graphics-performance-benchmark-review,4.html

DX12가 DX11보다 빠르지 않다.

그럼에도 D3D12를 학습할 이유가 있는가?

- D3D11에 대한 기능 업데이트는 오래 전에 끊겼다.
- Raytracing이 D3D12에서만 지원된다.
- 상용엔진(언리얼엔진)의 기본 API가 D3D11->D3D12로 바뀌었다.
- 개발환경(특히 디버깅)은 MS 플랫폼이 가장 편리하므로 차세대 API(Metal/Vulkan)로 이행하기 위해선 D3D12로 시작하는 편이 적응하기 쉽다.
  - metal은 D3D11의 특징과 D3D12의 특징을 모두 가지고 있다.

D3D12 12 Programming

성능향상을 위한 D3D11 -> D3D12 변화(~~빨라지지도 않았지만~~)

- CPU ->Draw Call -> GPU 처리의 과정을 줄임.
- 비동기 렌더링
- 완전한 멀티 스레드 렌더링(완전히 thread-safe)
- 각종 State들을 한방에 처리 ->
  - OMSet…RSSet…PSSet…VSSet… -> ID3D12PipelineState 한 개로.
- DirectX runtime과 드라이버에서 해주던 일들을 Application 레이어로 빼냄.~~(이 때문에 드라이버에서 최적화를 못해준다.)~~

API의 변화

- Resource Binding ->Root Signature / Descriptor Table
- Immediate Context -> Command List & Command Queue
- State변경(blend/depth/shader 설정 등) -> ID3D12PipelineState
- Shader 코드는 그대로 사용 가능.
- D3DX? 텍스쳐 파일 로딩함수도 직접 만들어야 함. 다만 이제는 DirectXTex가 D3D12도 지원한다.
- 자동으로 이루어졌던 Resource Transition은 ResourceBarrier를 사용해서 직접 처리해야한다.
- Dynamic Resource Renaming? 그딴거 없다.
  - 기존에는 draw작업을 위해 전달했던 리소스를 draw호출 직후 해제해도 상관없지만 d3d12에선 draw작업이 '실제로 gpu에서 완료 ' 될 때까지 리소스가 해제되거나 변경되어서는 안된다.

주요객체 대응 관계

D3D11	D3D12
ID3D11Device	ID3D12Device
ID3D11DeviceContext(Immediate)	ID3D12CommandQueue
ID3D11DeviceContext(Deferred)	ID3D12GraphicsCommndList
ID3D11Texture2D	D3D12_CPU_DESCRIPTOR_HANDLE
ID3D11Buffer	D3D12_VERTEX_BUFFER_VIEW
ID3D11VertexlShader/ID3D11PixelShader	ID3D12PipelineState
ID3D11ComputeShader	ID3D12PipelineState
ID3D11SamplerState	D3D12_STATIC_SAMPLER_DESC + ID3D12RootSignature
ID3D11DepthStencilState	ID3D12PipelineState

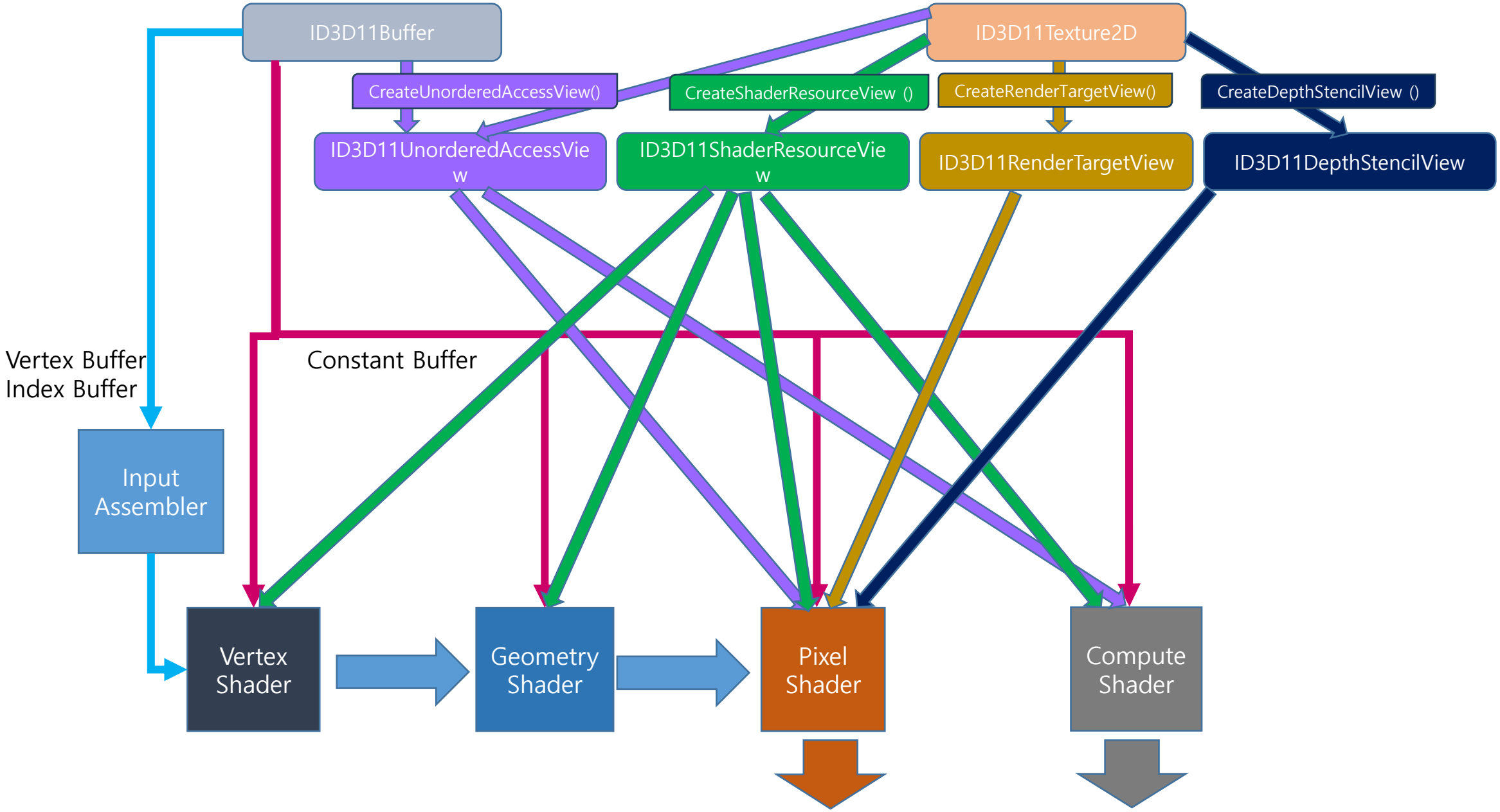
Resource Binding

Resource Binding

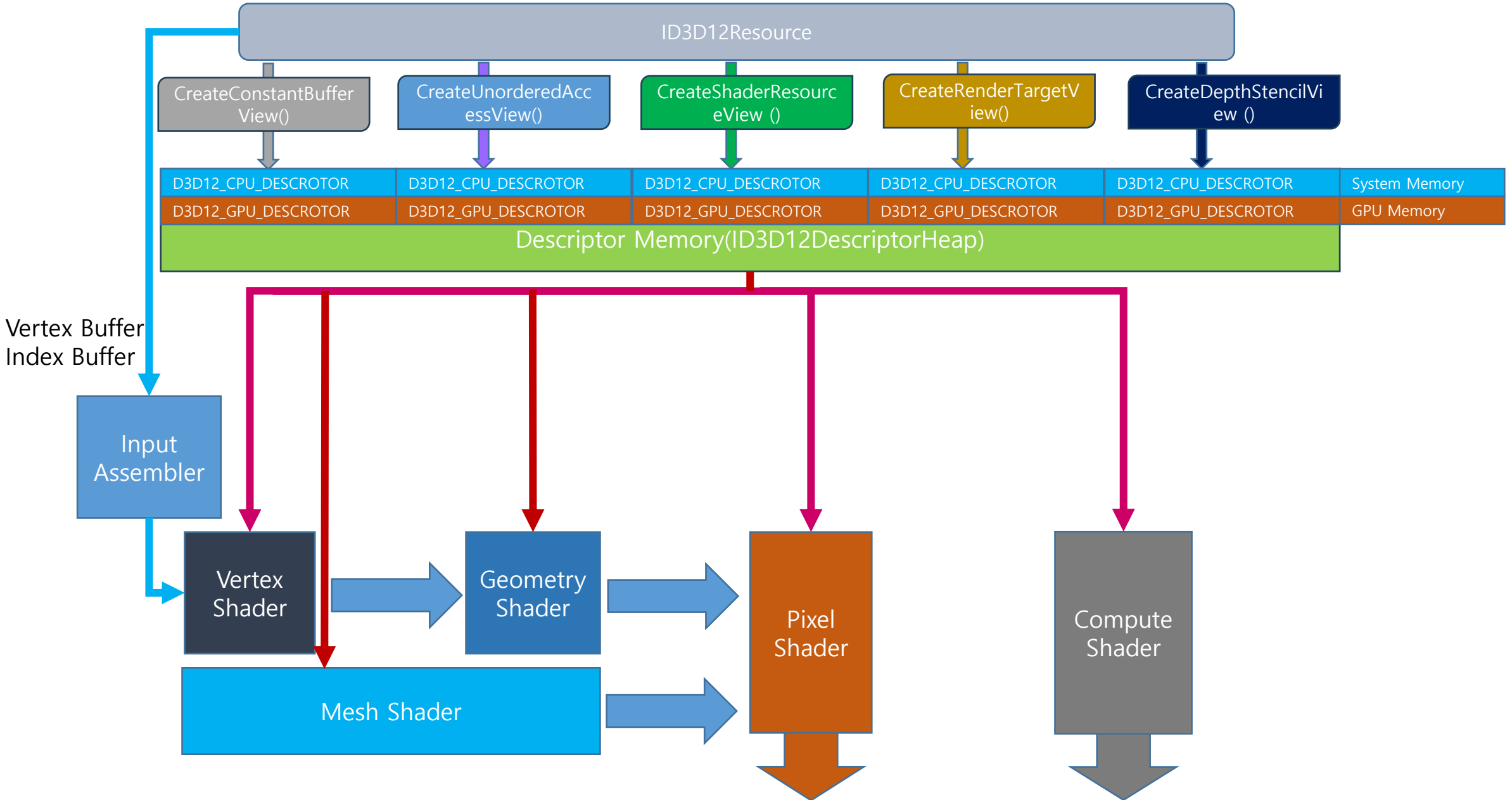
- Vertex Buffer
- Index Buffer
- Texture
- Constant Buffer
- Unordered Access Buffer (for Compute Shader)
- Sampler

렌더링을 위해 이러한 Resource들이 Graphics Pipeline에 bind되어야 한다.

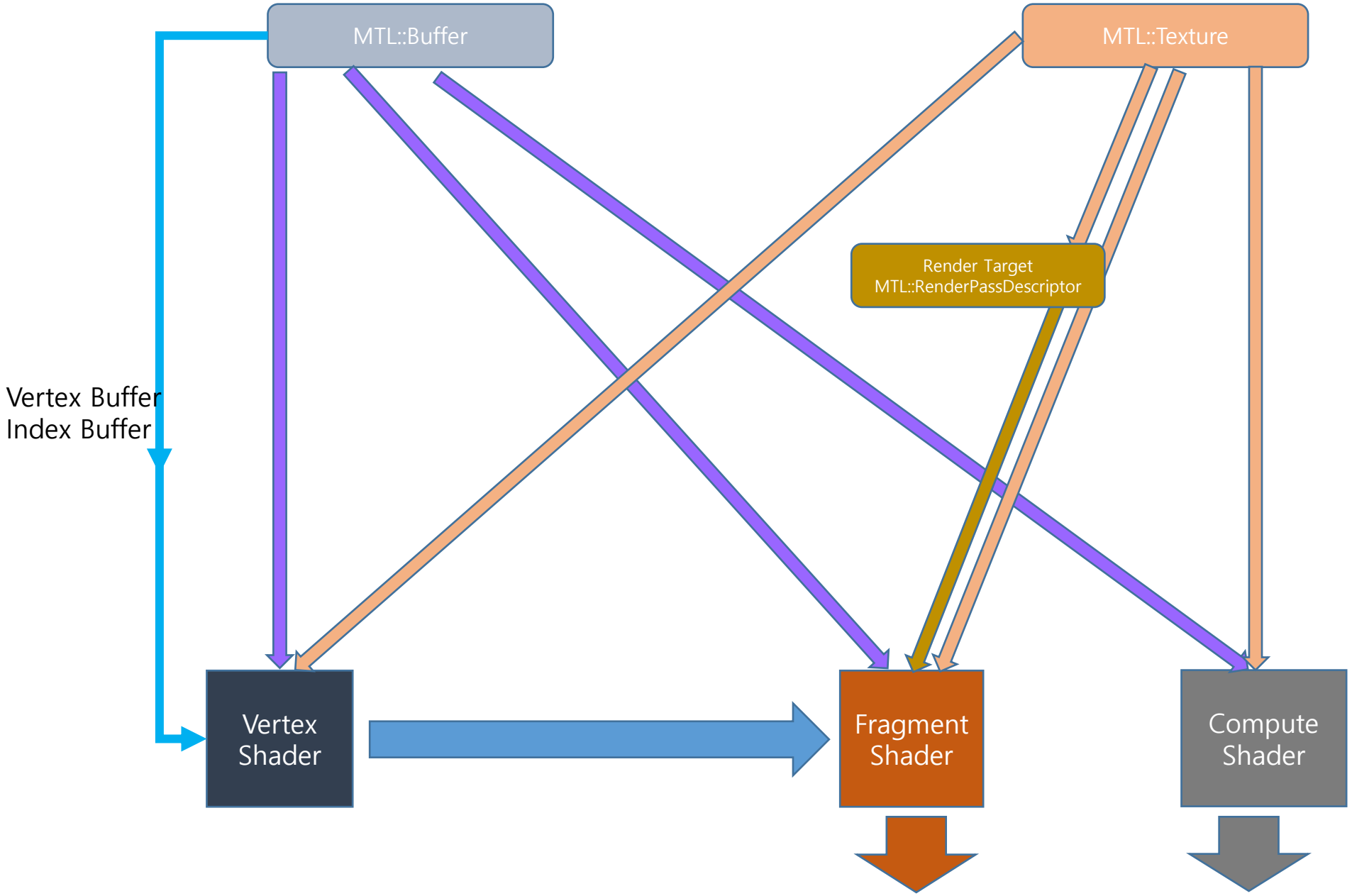
D3D11 Resource흐름



D3D12 Resource흐름



Metal Resource흐름

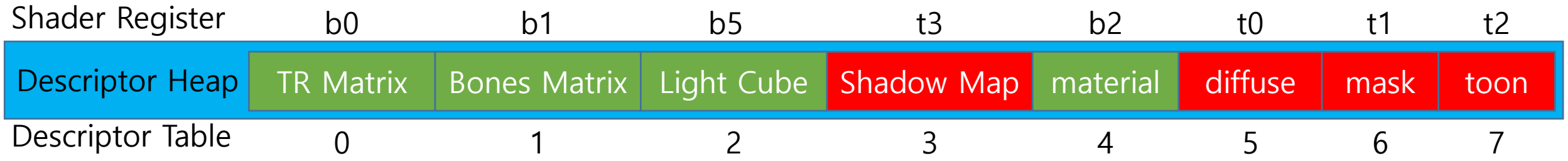


Descriptor

- Resource의 정보를 기술한 메모리 블록
- CBV,SRV,UAV,RTV,DSV를 생성하면 그 결과로 이 Descriptor를 얻는다.
- 32-64bytes 사이즈.(GPU마다 다름.)
- 객체가 아니다. 해제할 필요 없다.
- 내부적으로 GPU Memory, CPU Memory pair로 구성됨.
- D3D12_GPU_DESCRIPTOR_HANDLE, D3D12_CPU_DESCRIPTOR_HANDLE로 표현되며 사실상 포인터.

Descriptor Table

- Descriptor의 논리적 배열
- Draw/Compute 작업을 위해 RTV/DSV/CBV/SRV/UAV의 핸들이 Descriptor Table에 배치된 상태로 API에 전달된다.
- Descriptor Heap의 임의의 위치가 Descriptor Table에 맵핑된다.

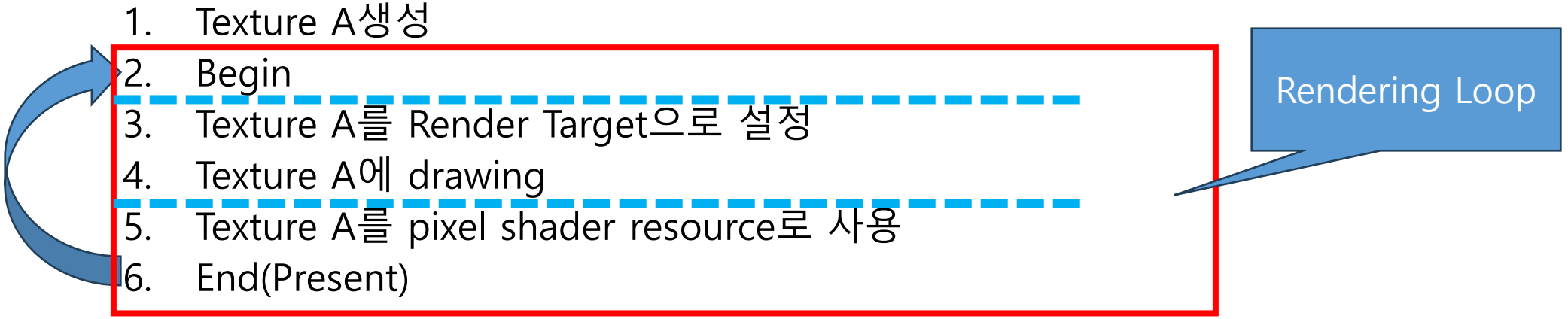


Descriptor Heap

- Descriptor Table로 사용할 선형 메모리
- Descriptor는 논리적인 단위, Descriptor Heap은 Descriptor Table이 실제로 구현될 물리메모리.
- ID3D12DescriptorHeap로 구현되어 있다.
- CPU측 메모리, GPU측 메모리 pair로 구성되어있다.
- RTV/DSV/CBV/SRV/UAV의 descriptor(기술자/일종의 포인터)는 이 Descriptor Heap의 CPU측 메모리에 기록(write)된다.
- CPU코드에서 Descriptor의 내용을 Descriptor Heap의 CPU측 메모리에 write, Shader에선 Descriptor Heap의 GPU측 메모리에서 read한다.

Resource Barrier

- GPU스레드가 GPU리소스를 억세스 할 때 동기화하기 위한 수단.
  - ex) Texture A를 GBuffer로 사용하는 시나리오



  - 4의 drawing이 끝나기 전에는 5의 pixel shader가 실행되어서는 안된다.
  - 5의 pixel shader작업이 끝나기 전에는 4의 drawing이 실행되어서는 안된다.
  - 즉 3과 5가 시작되기 전에 리소스의 상태를 보증(barrier를 둠으로서)하는 것으로 GPU스레드들을 동기화 시킨다.

Resource Barrier

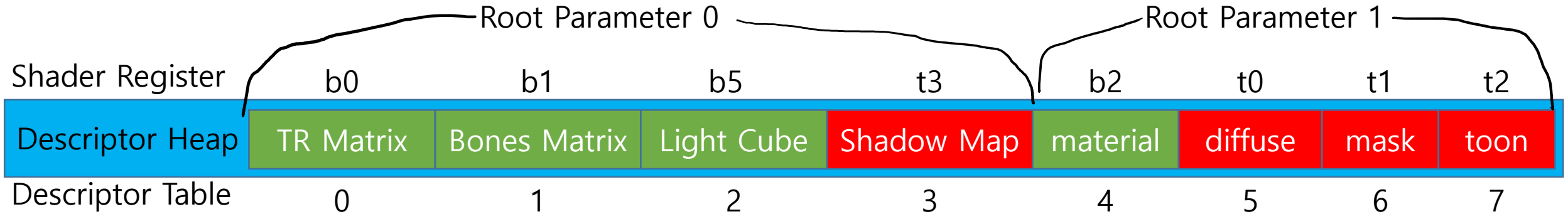
- RTV로 사용된 리소스는 SRV로 사용하기 전에 Transition되어야 함.
- SRV로 지정된 리소스는 RTV로 사용하기 전에 Transition되어야 함.
- DX11에선 자동 + implicit 이었으나 D3D12에선 수동 + explicit 으로 처리한다.

```
pCommandList->ResourceBarrier(1,
    &CD3DX12_RESOURCE_BARRIER::Transition(m_pRenderTargetDiffuse,
    D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE,
    D3D12_RESOURCE_STATE_RENDER_TARGET));

pCommandList->ResourceBarrier(1,
    &CD3DX12_RESOURCE_BARRIER:: Transition(m_pRenderTargetDiffuse,
    D3D12_RESOURCE_STATE_RENDER_TARGET,
    D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE));
```

Root Signature – ID3D12RootSignature

- 어떤(Texture, Constant Buffer, Sampler등) Resource가 어떻게 Pipeline에 bind 될지를 정의
- Resource binding 설정을 기술한 일종의 템플릿이다.
- Descriptor Table에서 어떤 리소스가 어느 레지스터로 맵핑되는지를 표현한다.

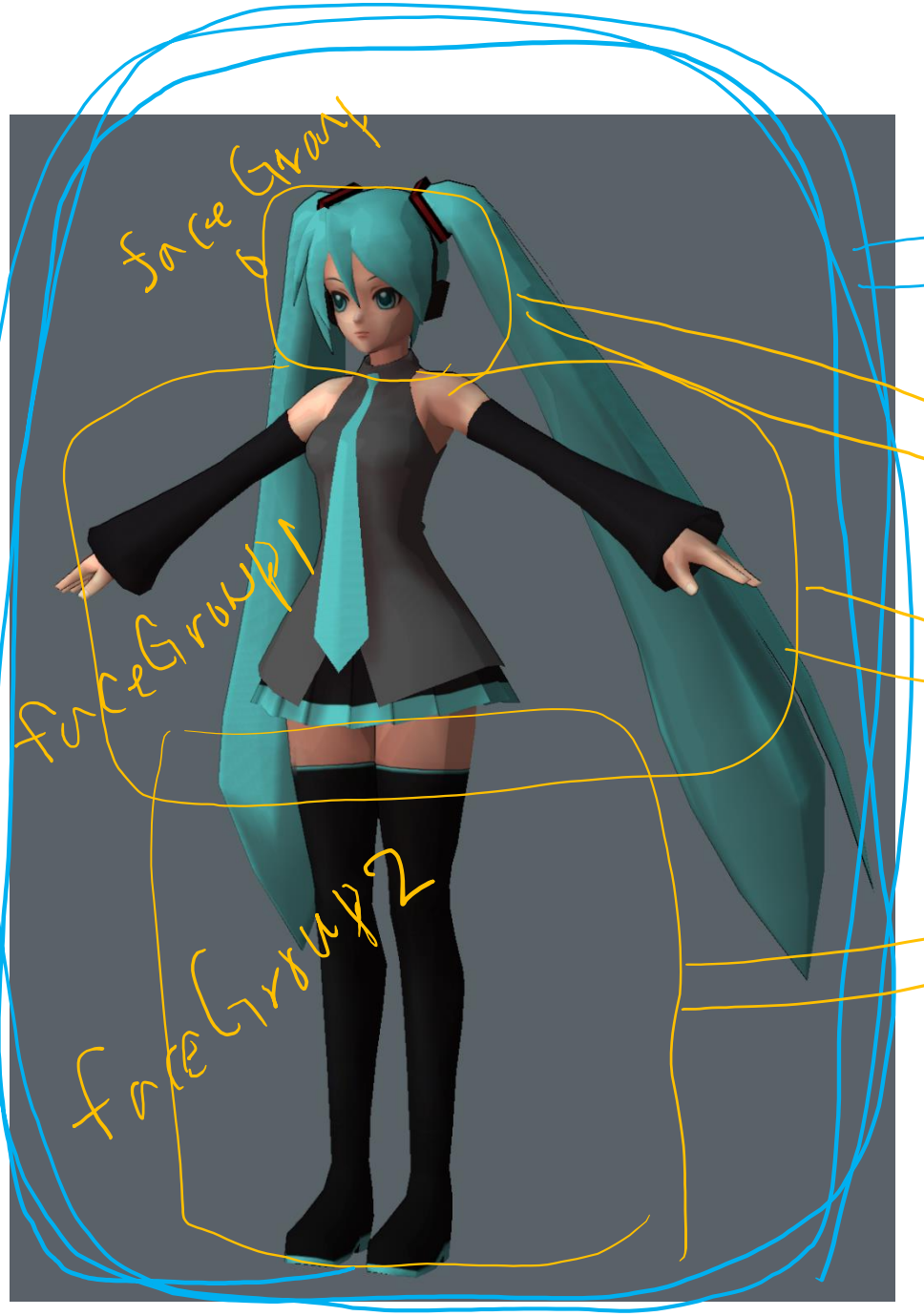
```
CD3DX12_DESCRIPTOR_RANGE rangesPerObj[3];
rangesPerObj[0].Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 2, 0);// b0 : default , b1 : bones
rangesPerObj[1].Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 1, 5);// b5 : Light Cube
rangesPerObj[2].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 1, 3);// t3 : shadow

CD3DX12_DESCRIPTOR_RANGE rangesPerFacegroup[2];
rangesPerFacegroup[0].Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 1, 2);// b2 : material
rangesPerFacegroup[1].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 3, 0);// t0 : diffuse , t1 : mask , t2 : toon ta
ble

CD3DX12_ROOT_PARAMETER rootParameters[2];
rootParameters[0].InitAsDescriptorTable(_countof(rangesPerObj), rangesPerObj, D3D12_SHADER_VISIBILITY_ALL);
rootParameters[1].InitAsDescriptorTable(_countof(rangesPerFacegroup), rangesPerFacegroup, D3D12_SHADER_VISIB
ILITY_ALL);
```



Root Signature와 Descriptor Table 사용

- 여러 개의 material그룹으로 쪼개진 캐릭터를 렌더링 한다고 할 때…

Root Signature와 Descriptor Table 사용



TR Matrix	Bones Matrix	Light Cube	Shadow Map

pCommandList->SetGraphicsRootDescriptorTable(0, gpuHeap);

FaceGroup 0	material	diffuse	mask	toon

pCommandList->SetGraphicsRootDescriptorTable(1, gpuHeap);

FaceGroup 1	material	diffuse	mask	toon

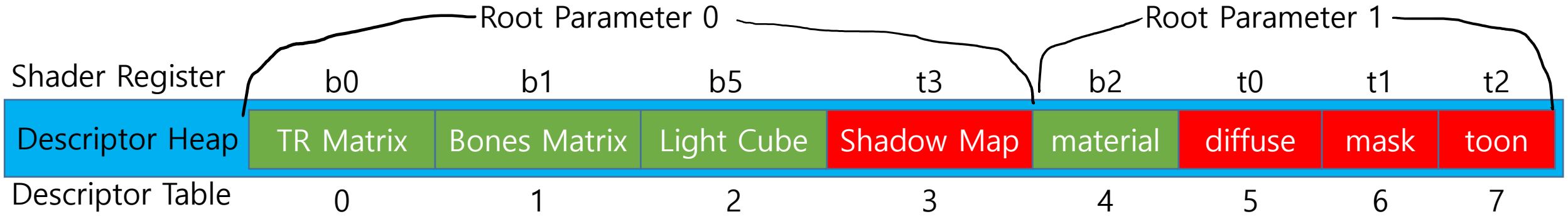
pCommandList->SetGraphicsRootDescriptorTable(1, gpuHeap);

FaceGroup 2	material	diffuse	mask	toon

pCommandList->SetGraphicsRootDescriptorTable(1, gpuHeap);

Root Signature와 Descriptor Table 사용

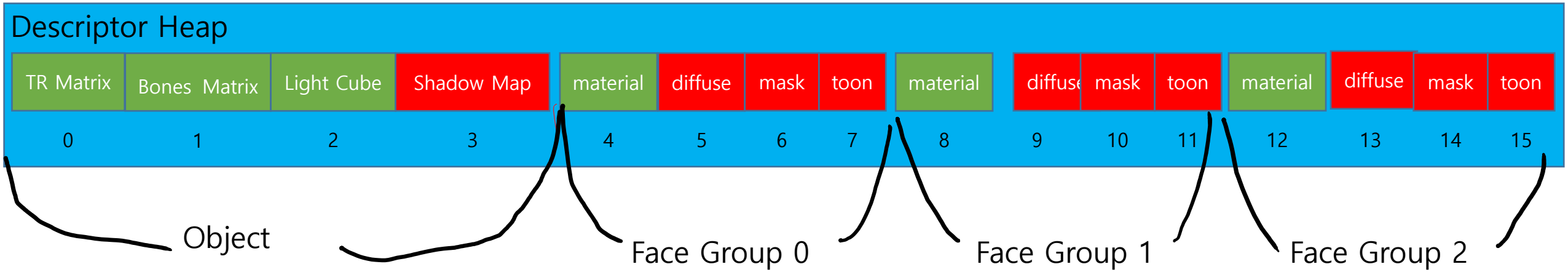
```
CD3DX12_DESCRIPTOR_RANGE rangesPerObj[3];
rangesPerObj[0].Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 2, 0);// b0 : default , b1 : bones
rangesPerObj[1].Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 1, 5);// b5 : Light Cube
rangesPerObj[2].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 1, 3);// t3 : shadow

CD3DX12_DESCRIPTOR_RANGE rangesPerFacegroup[2];
rangesPerFacegroup[0].Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 1, 2);// b2 : material
rangesPerFacegroup[1].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 3, 0);// t0 : diffuse , t1 : mask , t2 : toon ta
ble

CD3DX12_ROOT_PARAMETER rootParameters[2];
rootParameters[0].InitAsDescriptorTable(_countof(rangesPerObj), rangesPerObj, D3D12_SHADER_VISIBILITY_ALL);
rootParameters[1].InitAsDescriptorTable(_countof(rangesPerFacegroup), rangesPerFacegroup, D3D12_SHADER_VISIB
ILITY_ALL);
```



Root Signature와 Descriptor Table 사용

필요 Descriptor개수 : 16개

```
D3D12_GPU_DESCRIPTOR_HANDLE gpuHeap = pDescriptorHeap->GetGPUDescriptorHandleForHeapStart();
pCommandList->IASetVertexBuffer(...);
pCommandList->SetGraphicsRootDescriptorTable(0, gpuHeap);
for (i=0; i<3; i++)
{
        pCommandList->SetGraphicsRootDescriptorTable(1, gpuHeap);
        pCommandList->IASetIndexBuffer(...);
        pCommandList->DrawIndexedInstanced(...);
        gpuHeap.Offset(4, DescriptorSize);

}
```



Command List & Command Queue

- 비동기 렌더링과 멀티스레드 렌더링을 위한 디자인
- D3D11의 Immediate Context는 더 이상 존재하지 않는다.
- Graphics Command를 Command List 에 Recording해서
- Command Queue에 전송. (이 시점에 GPU Queue로 전송)
- 1개의 Command List만으로도 처리는 가능. -> 성능 안나옴
- 멀티스레드로 여러 개의 Command List를 동시에 recording하고 각각의 스레드가 독립적으로 Execute하는 것을 권장.

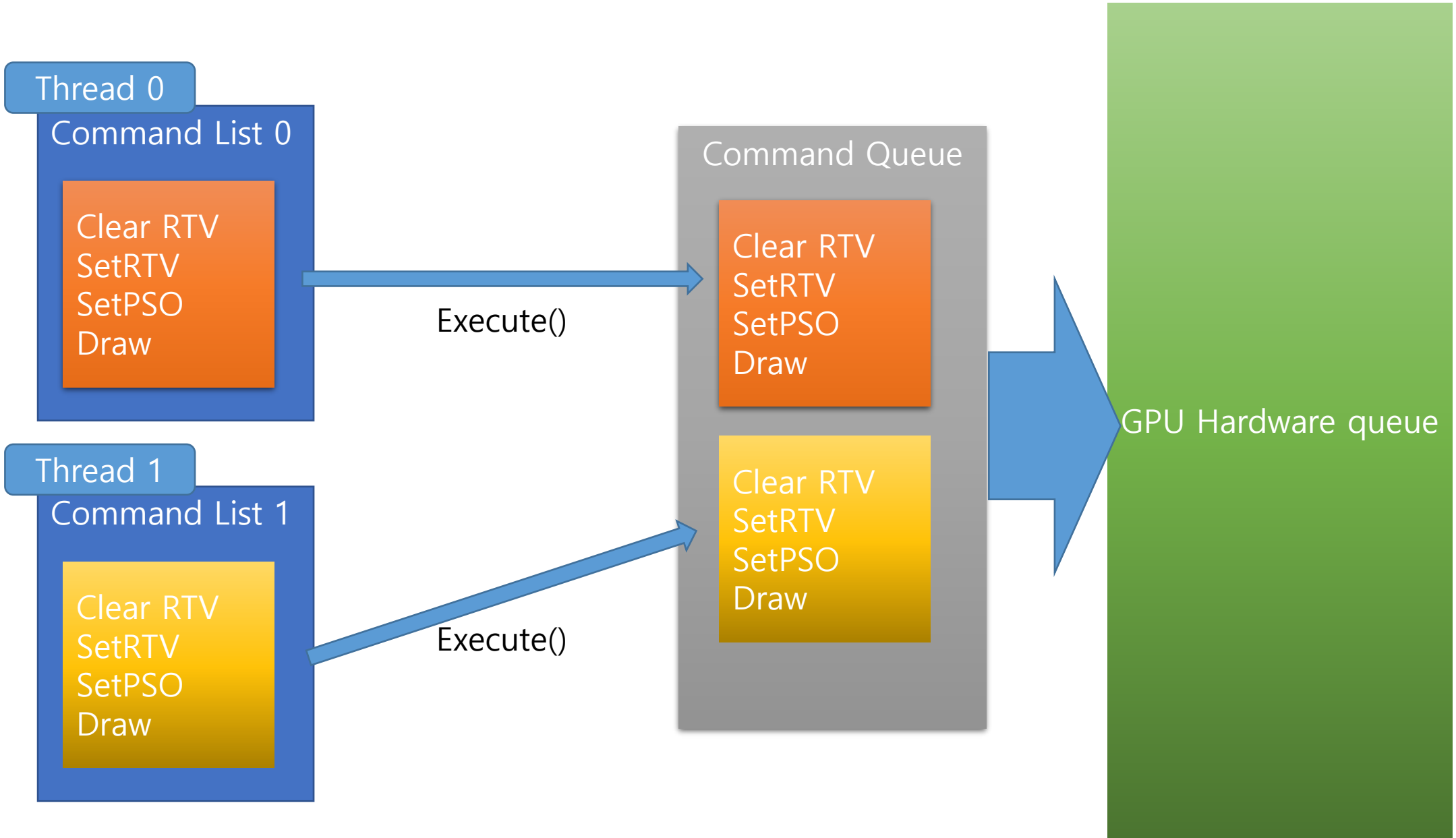
Command Queue – 작업 완료전까지 리소스 유지

- Command Queue에 작업을 제출했다고 하는 것이 렌더링 작업의 완료를 뜻하는 것이 아니다.
- Command Queue에 제출한 작업이 완료되기 전에 해당 작업에서 참조하는 D3D리소스들이 해제되어서는 안된다.

Command Queue – 완료 체크

- ID3D12Fence객체를 사용해서 작업이 완료되었는지를 표시한다.
- ID3D12CommandQueue::Signal(fence, completed value)를 호출, 전송한 작업이 완료되면 ID3D12Fence객체의 Completed value가 업데이트된다.
- ID3D12Fence::GetCompletedValue()로 작업이 완료되었는지를 체크.

Command Queue



Pipeline State – ID3D12PipelineState

- Blend State
- Depth State
- Render Target format
- Shaders…

이 모든 상태들을 하나로 묶어서 처리.

Shader 하나, Blend상태 하나 바꾸려고 해도 ID3D12PipelineState를 통째로 바꿔야함.

Shader 폭발에 이은 Pipeline State 폭발!

Pipeline State폭발의 예시

```
psoDesc.pRootSignature = m_pRootSignature;

D3D12_DEPTH_STENCIL_DESC    depthDescList[DEPTH_TYPE_NUM] = {};
SetDepthTypeDesc(depthDescList,_countof(depthDescList));

for (DWORD depth_type=0; depth_type<DEPTH_TYPE_NUM; depth_type++)
{
    psoDesc.DepthStencilState = depthDescList[depth_type];

    for (DWORD blend_index=0; blend_index<BLEND_TYPE_NUM; blend_index++)
    {
        psoDesc.BlendState = blendDesc[blend_index];

        for (DWORD shader_type=0; shader_type<VL_SHADER_TYPE_NUM; shader_type++)
        {
            for (DWORD param=0; param<SHADER_PARAMETER_COMBO_NUM; param++)
            {
                if (IsPhysique(param))
                {
                    psoDesc.InputLayout = { layoutVL_Physique, _countof(layoutVL_Physique) };
                }
                else
                {
                    psoDesc.InputLayout = { layoutVL, _countof(layoutVL) };
                }

                psoDesc.VS = CD3DX12_SHADER_BYTECODE(m_pVS[shader_type][param]->pCodeBuffer,m_pVS[shader_type][param]->dwCodeSize);
                psoDesc.PS = CD3DX12_SHADER_BYTECODE(m_pPS[shader_type][param]->pCodeBuffer,m_pPS[shader_type][param]->dwCodeSize);

                if (FAILED(pResourceManager->CreateGraphicsPipelineState(&psoDesc, &m_pPipelineState[depth_type][blend_index][shader_type][param])
                    __debugbreak();
            }
        }
    }
}
```

- 5차원, 6차원, 7차원 ..N차원 배열을 쓰던가...
- MS에서 권장하는 방법은 PipelineState cache를 만들어서 사용하라는 것.
- 다양한 렌더링 옵션을 설정하는 상황을 아예 피하게 됨. -> 결과적으로 툴에서 사용할 API로는 D3D12보다 D3D11을 선호하게 됨.

Write to GPU Resources

D3D11 Create – Update - Draw

생성과 동시에 값을 써넣기

- CreateBuffer(const D3D11_SUBRESOURCE_DATA *pInitialData)
- Draw()

or

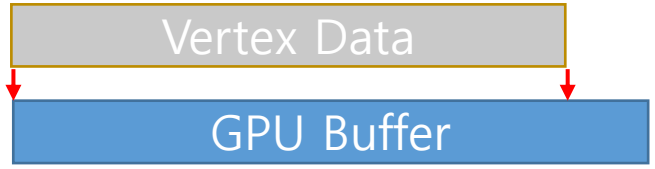
생성 후 값을 써넣기

- CreateBuffer()
- UpdateSubResource()
- Draw()

D3D11 Create – Update - Draw

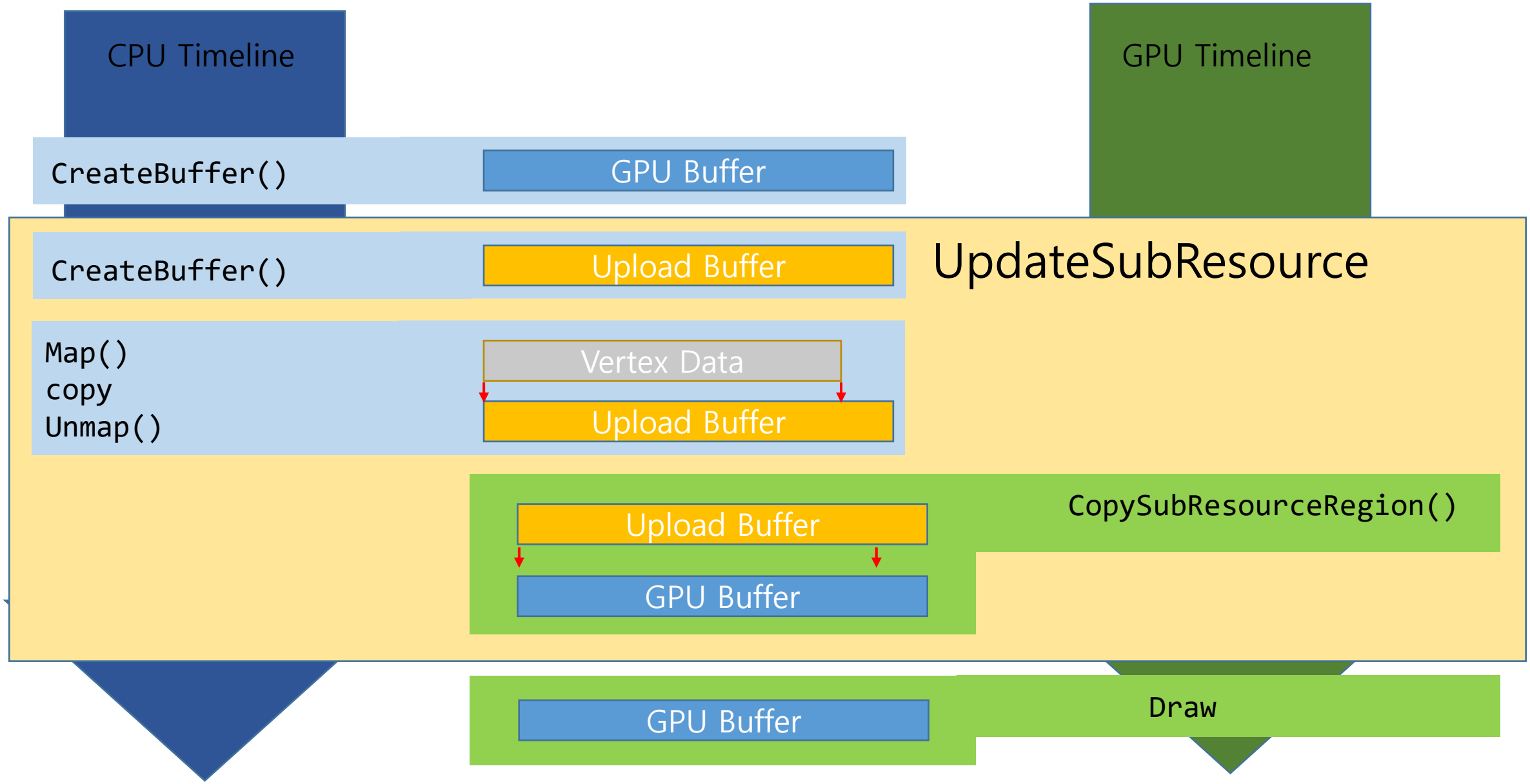
CreateBuffer()

UpdateSubResource



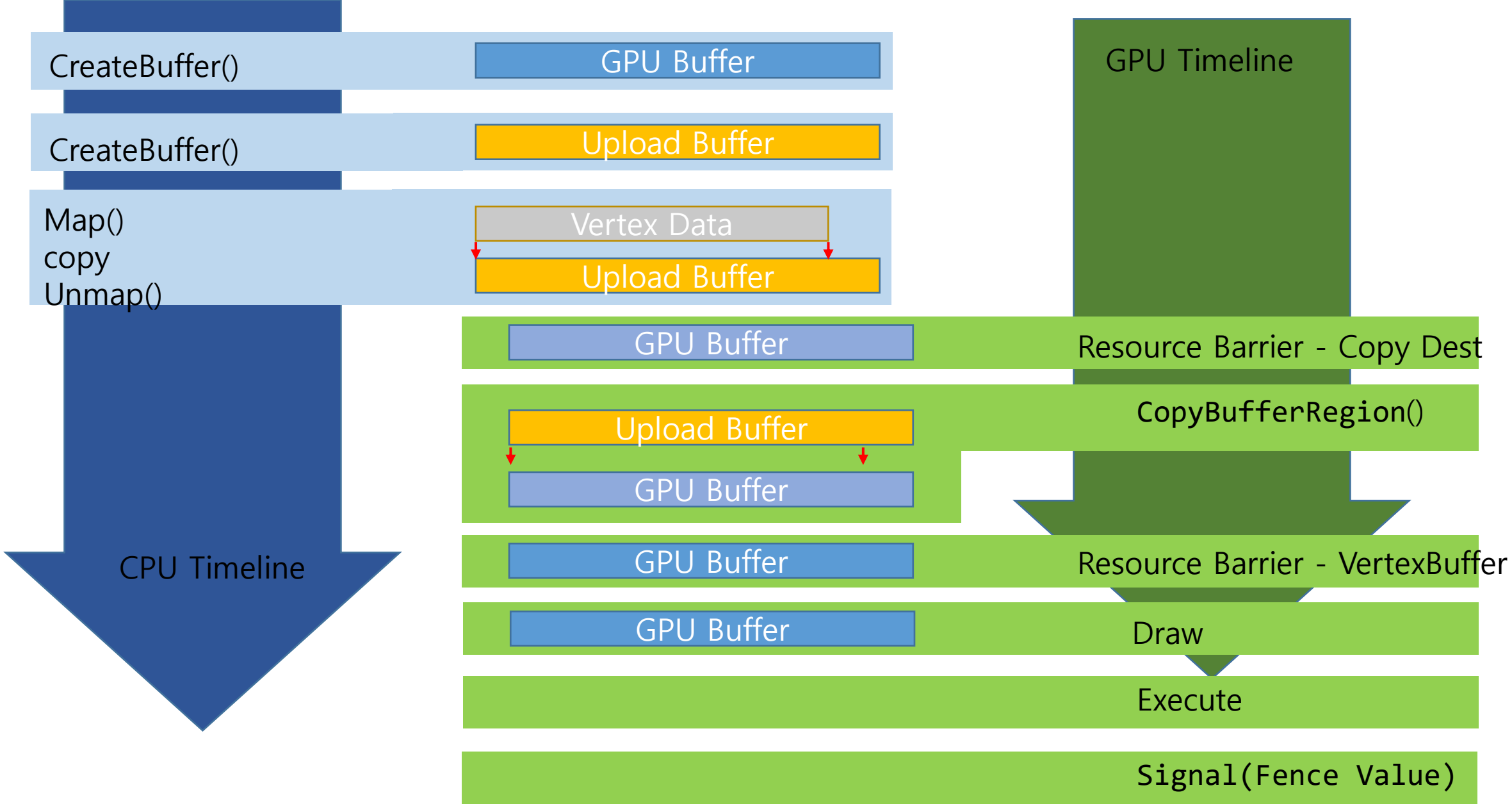
Draw

D3D11- Manually

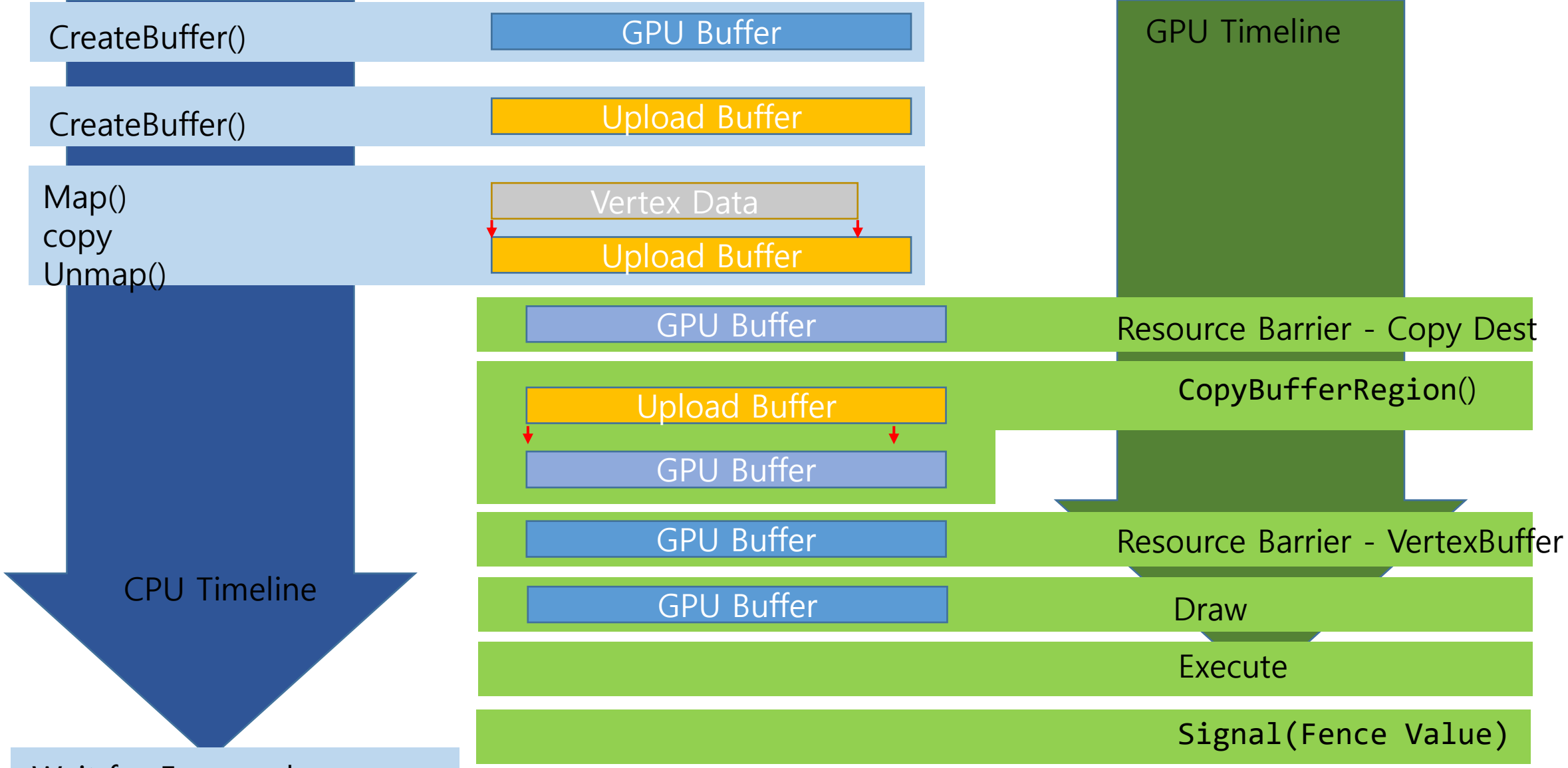


D3D12 Create – Update - Draw

- CreateBuffer() – GPU Buffer
- CreateBuffer() – UploadBuffer
- ResourceBarrier() – Copy Dest
- CopyResource (UploadBuffer -> GPU Buffer)
- ResourceBarrier() – Vertex Buffer
- Draw()
- Execute()



Upload Buffer는 언제 해제할까?



작업 완료를 확인한 후 해제 가능

자주 하는 실수

- D3D11의 Immediate Context에선 한번 설정한 state가 변경하기 전까지 계속 유지됨.
- D3D12에선 CommandQueue에 기본 설정된 RTV, DSV, Viewport 없음. 따라서 D3D12에선 Command List를 사용할 때마다 RTV, DSV, Viewport를 설정할 필요가 있음.
- OMSetRenderTarget(), RSSetViewport등은 커맨드 초반에 먼저 호출해줄것.
- Fence ->Wait-> Complete 될 때까지 리소스 상태를 유지할 것.
  - 성능을 위해서 결국은 fence->wait를 제거해야 하는데 리소스의 수명 관리를 위한 별도 자료구조가 필요하다.

Debugging

Debugging

- 비동기 렌더링 특성상 API에 잘못된 파라미터를 전달해도 그 즉시 알기 어려움.
- 꼭 버그를 잡기 위함이 아니라 작동원리를 알기 위해서라도 디버거로 추적해볼 필요가 있다.
- DX런타임 또는 드라이버의 버그로 보이더라도 내가 뭘 잘못했는지 먼저 체크하라.

D3D12 Debug Layer의 사용

- 전달된 파라미터의 유효성, Resource Barrier의 상태 등등 프로그래머의 실수를 미리 잡아준다.
- 에러가 아닌데 에러라고 판별할 가능성은 거의 없다. 반드시 무시하지 말고 체크할 것.
- 최근에는 GPU타임라인에서 버그체크를 수행하는 GBV(GPU Based Validation)기능이 추가되었다.

```
ID3D12Debug*pDebugController = nullptr;
ID3D12Debug5*pDebugController5 = nullptr;

// Enable the D3D12 debug layer.
if (SUCCEEDED(D3D12GetDebugInterface(IID_PPV_ARGS(&pDebugController))))
{
        pDebugController->EnableDebugLayer();
        if (S_OK == pDebugController->QueryInterface(IID_PPV_ARGS(&pDebugController5)))
        {
            pDebugController5->SetEnableGPUBasedValidation(TRUE);
            pDebugController5->SetEnableAutoName(TRUE);
        }
}
```

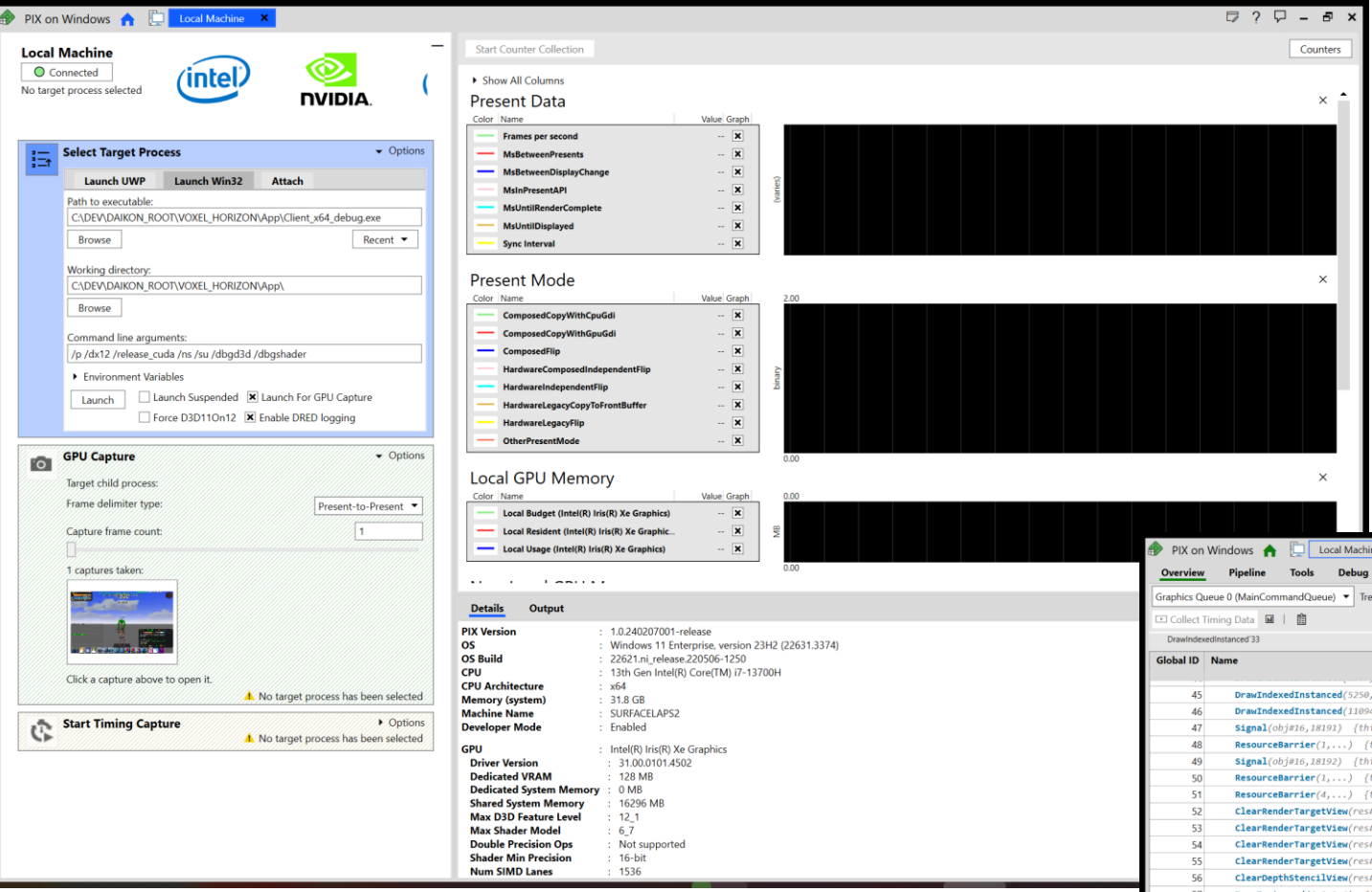
Visual Studio Graphics Debugger

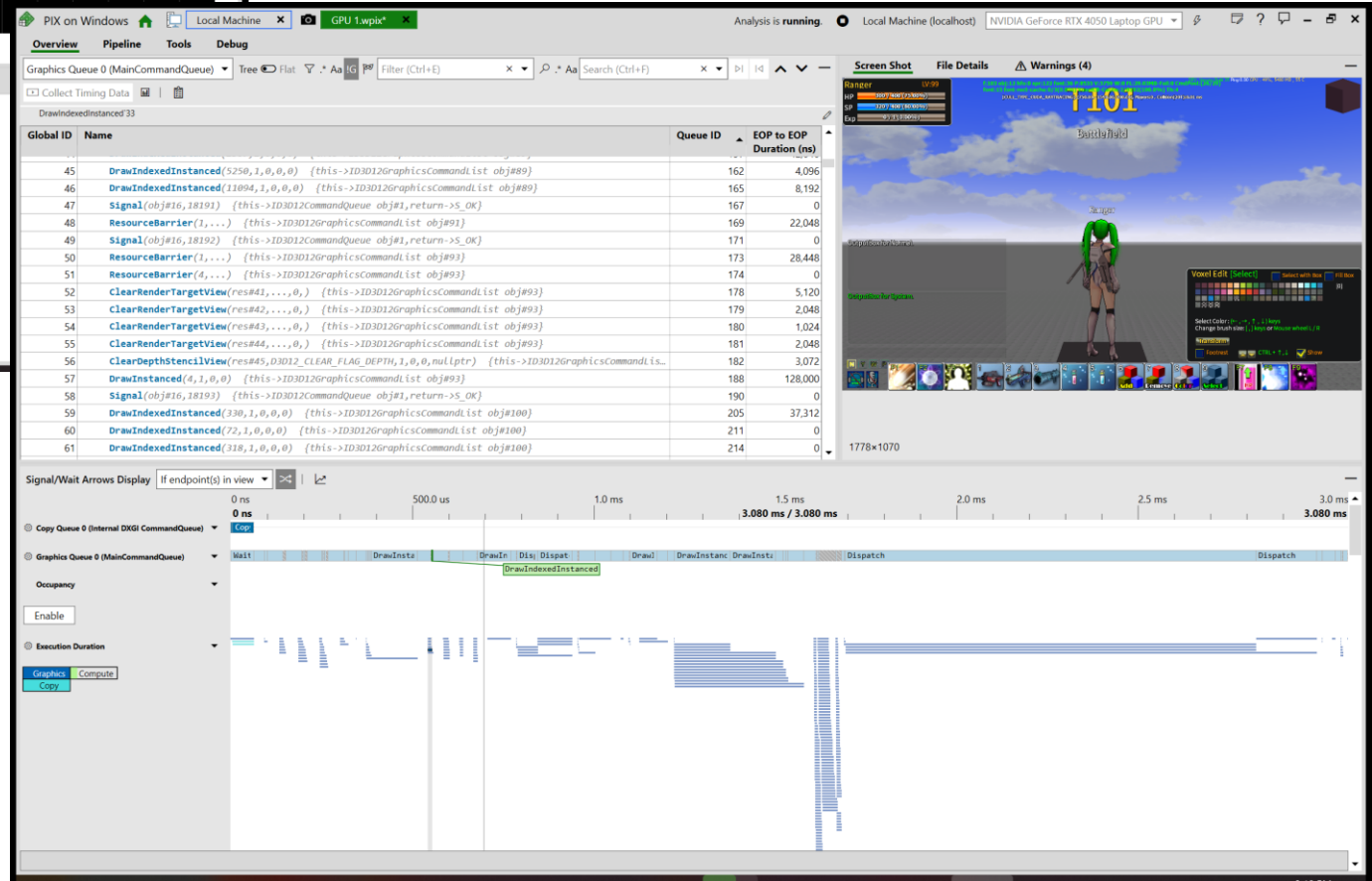
- D3D12 초기에는(2015-2017) VS Graphics Debugger로 D3D12 디버깅이 가능했으나 현재는 D3D11디버깅만 가능.

PIX

- 현재 D3D12디버깅은 pix로만 가능(nvidia Parallel Nsight로도 가능)
- Pipeline에 바인드된 D3DResource를 추적할 수 있다.
- Shader 디버깅 가능
- Descriptor Table의 내용을 볼 수 있는 것이 큰 장점.
- **ID3D12Object::SetName**()으로 이름을 지정해놓으면 디버거에서 쉽게 해당 객체를 찾을 수 있다.
- Shader디버깅을 위해서는 HLSL코드를 컴파일 할 때 optimize옵션을 꺼줘야 한다.

PIX


PIX on Windows
Local Machine
Local Machine
Connected
No target process selected
Start Counter Collection
Counters
Show All Columns
Present Data
Frames per second
MsBetweenPresents
MsBetweenDisplayChange
MsInPresentAPI
MsUntilRenderComplete
MsUntilDisplayed
Sync Interval
Present Mode
ComposedCopyWithCpuGdi
ComposedCopyWithGpuGdi
ComposedFlip
HardwareComposedIndependentFlip
HardwareIndependentFlip
HardwareLegacyCopyToFrontBuffer
HardwareLegacyFlip
OtherPresentMode
Local GPU Memory
Local Budget (Intel(R) Iris(R) Xe Graphics)
Local Resident (Intel(R) Iris(R) Xe Graphic...
Local Usage (Intel(R) Iris(R) Xe Graphics)
Select Target Process
Options
Launch UWP
Launch Win32
Attach
Path to executable:
C:\DEV\DAIKON_ROOT\VOXEL_HORIZON\App\Client_x64_debug.exe
Browse
Recent
Working directory:
C:\DEV\DAIKON_ROOT\VOXEL_HORIZON\App\
Browse
Command line arguments:
/p /dx12 /release_cuda /ns /su /dbgd3d /dbgshader
Environment Variables
Launch
Launch Suspended
Launch For GPU Capture
Force D3D11On12
Enable DRED logging
GPU Capture
Target child process:
Frame delimiter type:
Present-to-Present
Capture frame count:
1
1 captures taken:
Click a capture above to open it.
No target process has been selected
Start Timing Capture
No target process has been selected
Details
Output
PIX Version : 1.0.240207001-release
OS : Windows 11 Enterprise, version 23H2 (22631.3374)
OS Build : 22621.ni_release.220506-1250
CPU : 13th Gen Intel(R) Core(TM) i7-13700H
CPU Architecture : x64
Memory (system) : 31.8 GB
Machine Name : SURFACELAPS2
Developer Mode : Enabled
GPU : Intel(R) Iris(R) Xe Graphics
Driver Version : 31.00.0101.4502
Dedicated VRAM : 128 MB
Dedicated System Memory : 0 MB
Shared System Memory : 16296 MB
Max D3D Feature Level : 12_1
Max Shader Model : 6_7
Double Precision Ops : Not supported
Shader Min Precision : 16-bit
Num SIMD Lanes : 1536


PIX on Windows
Local Machine
GPU 1.wpix*
Analysis is running.
Local Machine (localhost)
NVIDIA GeForce RTX 4050 Laptop GPU
Overview
Pipeline
Tools
Debug
Graphics Queue 0 (MainCommandQueue)
Collect Timing Data
Global ID
Name
Queue ID
EOP to EOP Duration (ns)
45 DrawIndexedInstanced(5250,1,0,0,0) 162 4,096
46 DrawIndexedInstanced(11094,1,0,0,0) 165 8,192
47 Signal(obj#16,18191) 167 0
48 ResourceBarrier(1,...) 169 22,048
49 Signal(obj#16,18192) 171 0
50 ResourceBarrier(1,...) 173 28,448
51 ResourceBarrier(4,...) 174 0
52 ClearRenderTargetView(res#41,...,0,) 178 5,120
53 ClearRenderTargetView(res#42,...,0,) 179 2,048
54 ClearRenderTargetView(res#43,...,0,) 180 1,024
55 ClearRenderTargetView(res#44,...,0,) 181 2,048
56 ClearDepthStencilView(res#45,D3D12_CLEAR_FLAG_DEPTH,1,0,0,nullptr) 182 3,072
57 DrawInstanced(4,1,0,0) 188 128,000
58 Signal(obj#16,18193) 190 0
59 DrawIndexedInstanced(330,1,0,0,0) 205 37,312
60 DrawIndexedInstanced(72,1,0,0,0) 211 0
61 DrawIndexedInstanced(318,1,0,0,0) 214 0
Screen Shot
File Details
Warnings (4)
1778×1070
Signal/Wait Arrows Display
If endpoint(s) in view
0 ns
500.0 us
1.0 ms
1.5 ms
2.0 ms
2.5 ms
3.0 ms
3.080 ms / 3.080 ms
Copy Queue 0 (Internal DXGI CommandQueue)
Graphics Queue 0 (MainCommandQueue)
Occupancy
Enable
Execution Duration
Graphics
Compute
Copy
DrawIndexedInstanced
Dispatch

Optimization

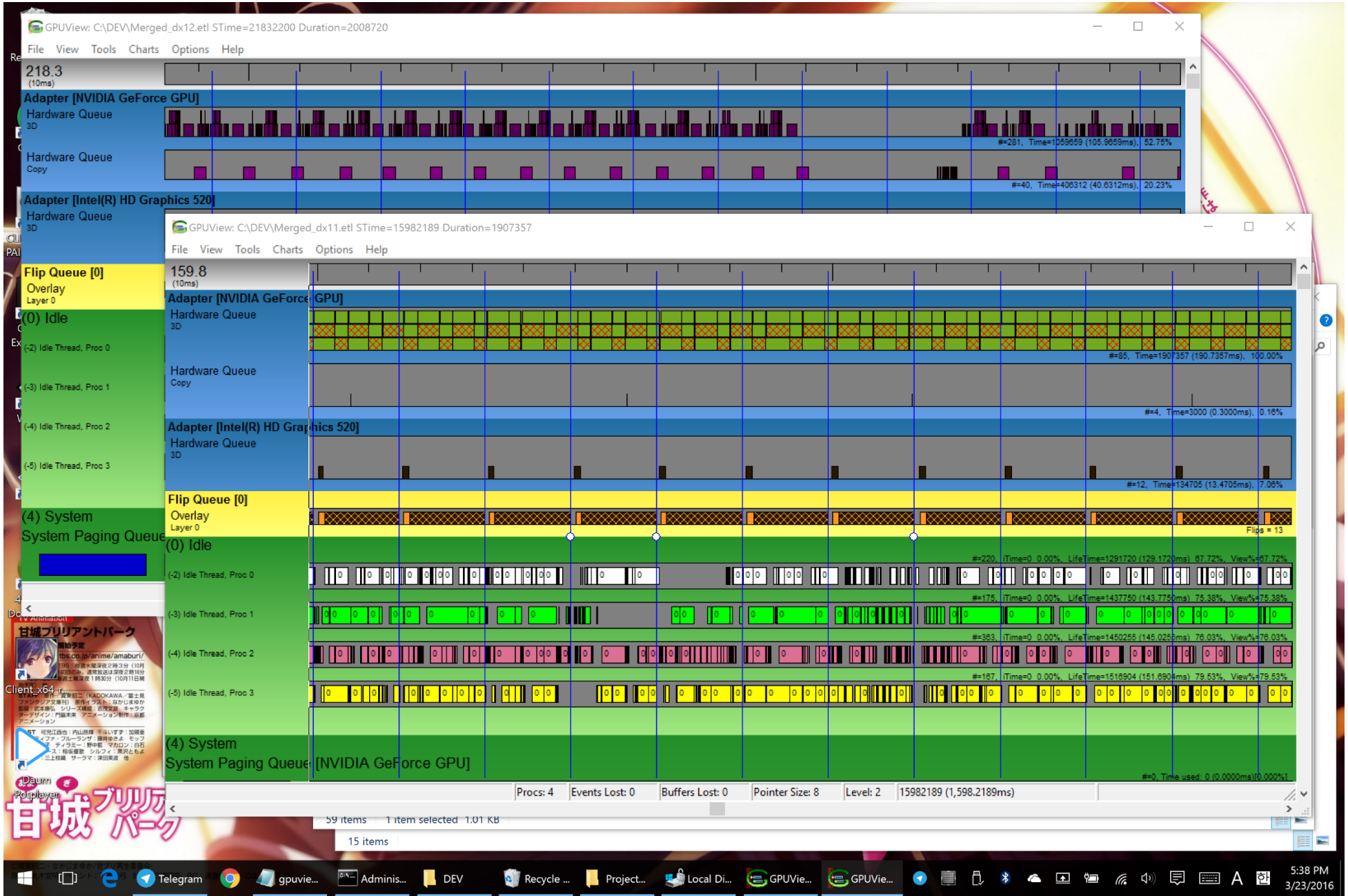
힘들게 D3D12로 포팅했는데 D3D11보다 느리네...

- 수동으로 GPU H/W Queue를 꽉 채워넣기가 어려움. D3D11에선 런타임과 드라이버가 자동으로 해줌.
- GPU가 펑펑 노는게(GPU점유율이 낮게 나옴) 근본적인 문제.

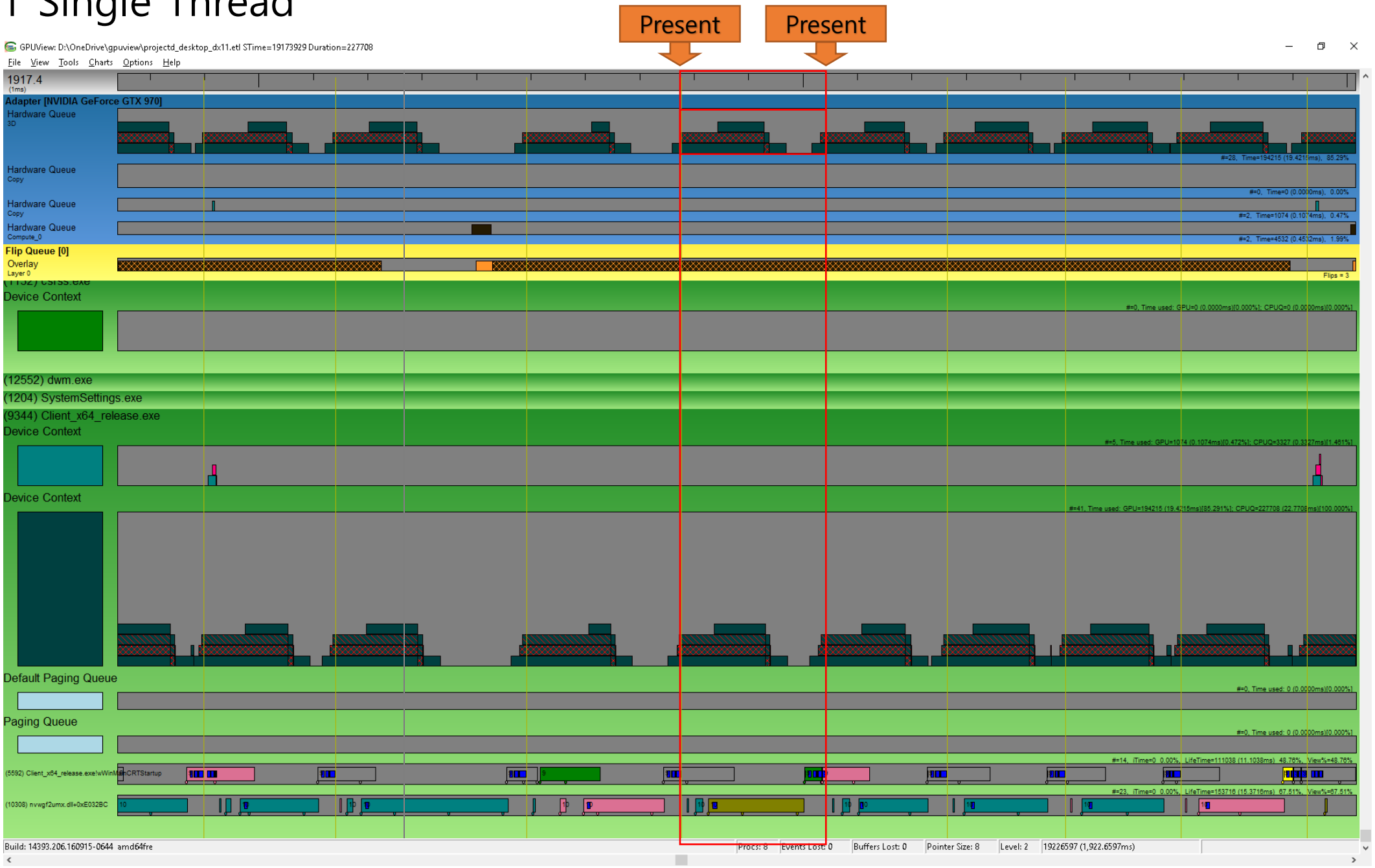
GPUView

- Windows ADK의 Performance Toolkit에 포함된 로그 분석기
  1. ADK설치
  2. Admin권한으로 CMD을 열고
  - C:₩Program Files (x86)₩Windows Kits₩10₩Windows Performance Toolkit₩gpuview 폴더로 이동
  - >Log.cmd 로 logging시작
  - 다시 Log.cmd를 입력해서 logging종료
  - Merged.etl파일을 GPUView에서 로드

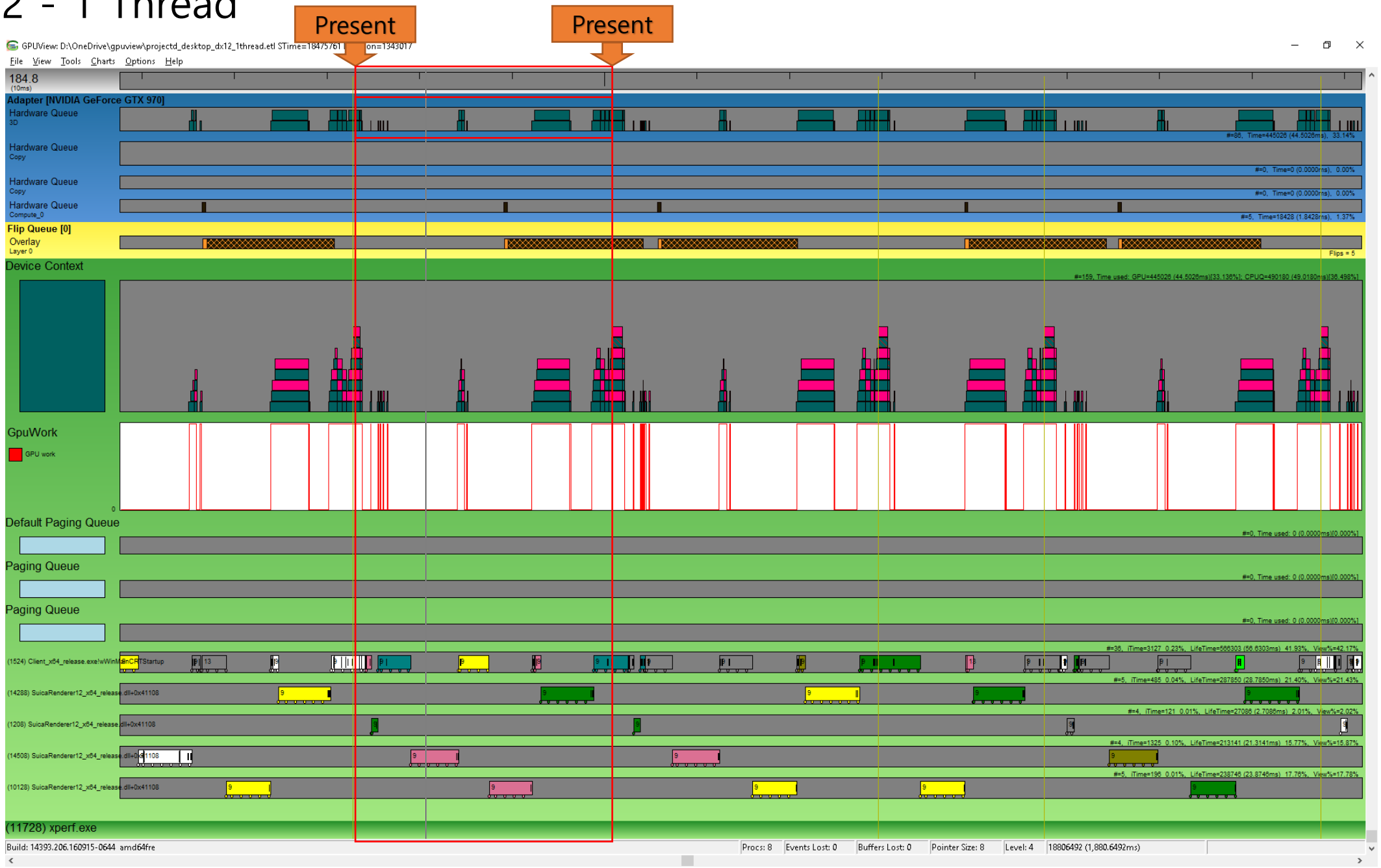
GPUView로 보는 D3D11 vs D3D12



D3D11 Single Thread



D3D12 - 1 Thread



D3D11 – 아무 최적화 안해도 빠르다.

- Single-Thread Rendering
- 신경 쓴 것이라곤 GPU Memory에 최대한 올려놓고 안건드렸을 뿐 (렌더링중 Map(), Unmap()을 피한다.)
- 심지어 D3DResource를 렌더링 중에 마구 변경해가며 재활용하고 있다. -> D3D11은 Resource Renaming으로 이 문제를 해결한다
- 그럼에도 불구하고 D3D11은 GPU Queue를 꽉꽉 채우며 최대한의 성능을 내주고 있다.

D3D12 – 나름의 최적화를 해도 느리다.

- 아직 최적화를 안해서 그렇다고 믿고 싶겠지만 사실 최적화에 꽤 신경써도 느리다.

원인분석

- D3D11에 대응하는 GPU의 user mode 드라이버가 해주는 최적화를 하나도 사용할 수 없다.
- 드라이버가 해주던 최적화는 어플리케이션 프로그래머가 직접 해줘야 한다.

최적화할 수 있는 포인트

- Command List 작성과 Execute의 적절한 배분.
  - 하나의 Command List에 몰아서 Command를 기록하고 마지막에 Execute 한번만 하면? -> GPU가 펑펑 놀다가 마지막에 한번에 과부하를 받게 된다.
  - 여러 개의 Command List를 사용해서 Command기록과 Execute를 동시에 처리해야할 필요가 있다.
- Command List 작성 시간을 줄이기 - 멀티 스레드 렌더링이 필요.
- D3D11에선 자동으로 비동기식 중첩 렌더링이 지원되지만 D3D12에선 수동으로 처리해야 한다.
  - D3D11 - 1번 프레임 렌더링 준비중일때, GPU에선 0번 프레임 렌더링
  - D3D12 – execute – fence –wait로 처리한다면 0번 프레임 렌더링이 끝나야 1번 프레임 렌더링을 준비할 수 있고 그 동안 GPU는 논다.

Execute호출 빈도 조절

- Execute()를 해야만 GPU H/W Queue로 Command전송
- D3D12엔진의 초기버전에선 Execute()를 present()직전에 한번만 호출했었다.
  - 그래서 Present 직전까지 GPU가 그냥 놀았다-_-;;;;;
- Command Queue를 너무 적게 채우고 execute를 자주 호출해도 문제.
- Commane Queue를 너무 꽉 채우고 execute를 적게 호출해도 문제.
- 테스트 하면서 튜닝이 필요하다.

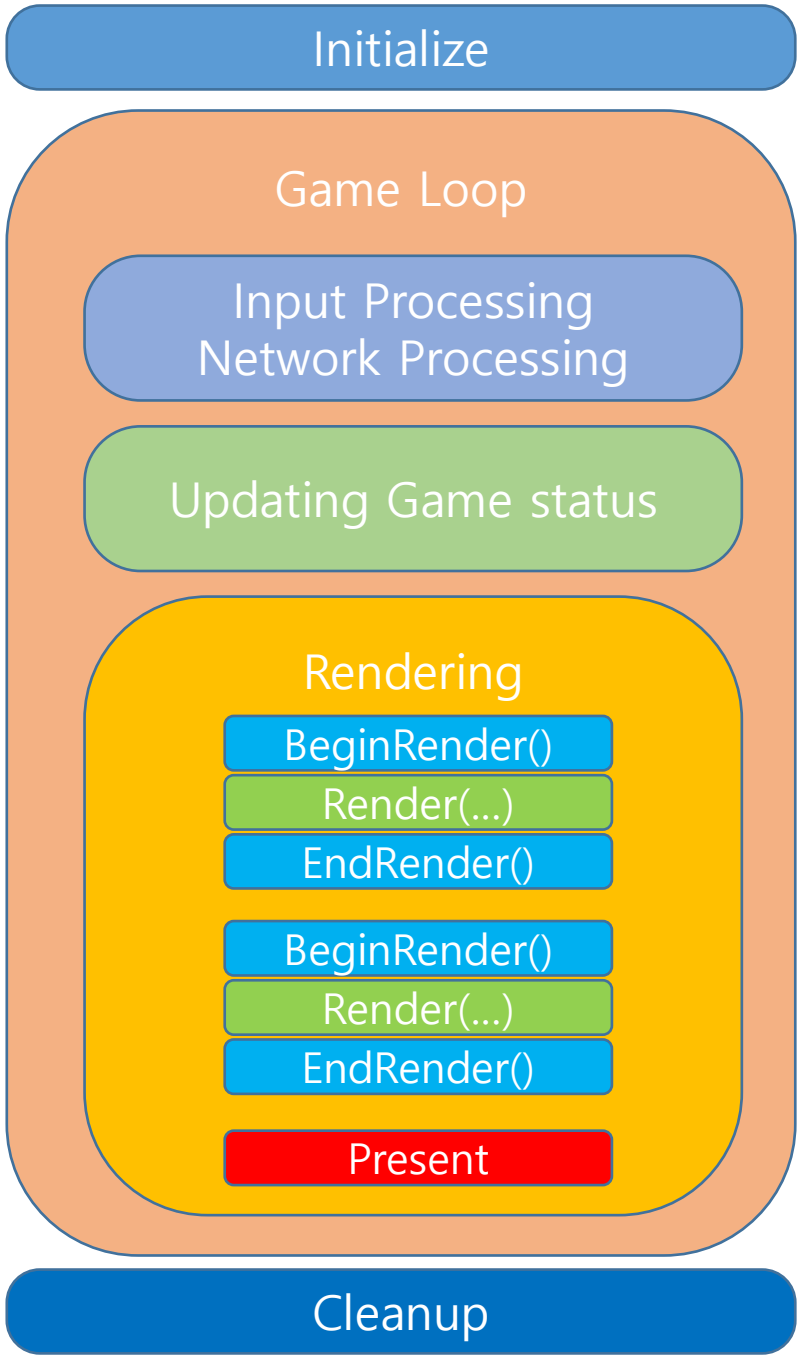
Command List Pooling

- 여러 개의 Command List에 나눠 담아서 execute해야하므로 당연히 여러 개의 Command List를 생성해야 한다.
- 따라서 다수의 Command List를 미리 할당해 둔다(실시간으로 Command List를 생성하는 바보짓을 누가 할까 싶긴 하지만....)
- Command List 하나당 N개의 오브젝트 렌더링에 대한Command를 기록.
- N개에 도달하면 Execute(), 다음번 Command List에 계속해서 렌더링 Command 기록.
- 렌더링이 끝났음을 확인한 경우 pool에 반납.
- 여분의 Command List가 없을 경우 추가 생성 또는 wait.
- Command List를 너무 많이 생성하면 메모리를 미친듯이 잡아먹는다!

스케쥴링 기능의 조정

- 멀티 스레드 렌더링
- 중첩 프레임 렌더링

Game Loop

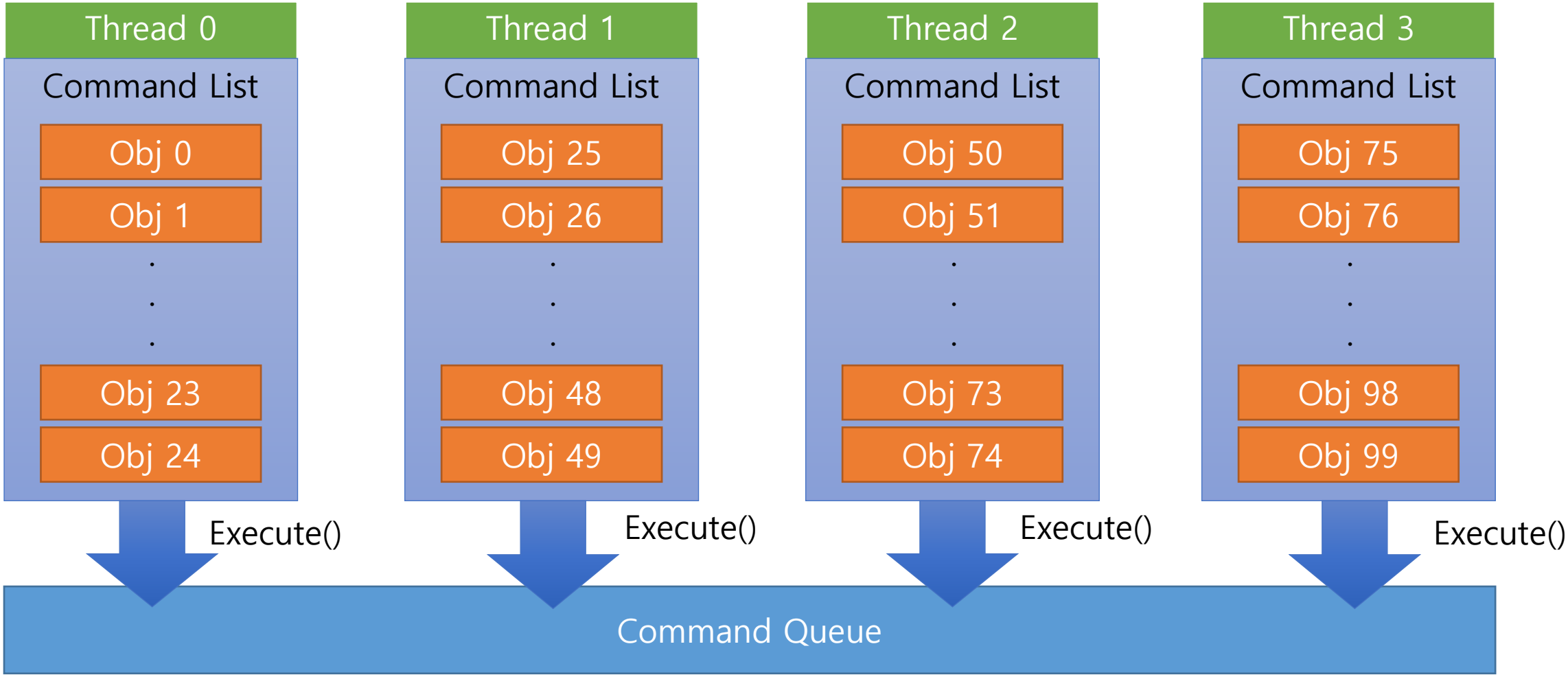


- BeginRender() 과 EndRender()사이에 draw/dispatch
- Present()호출 후 wait

멀티 스레드 렌더링

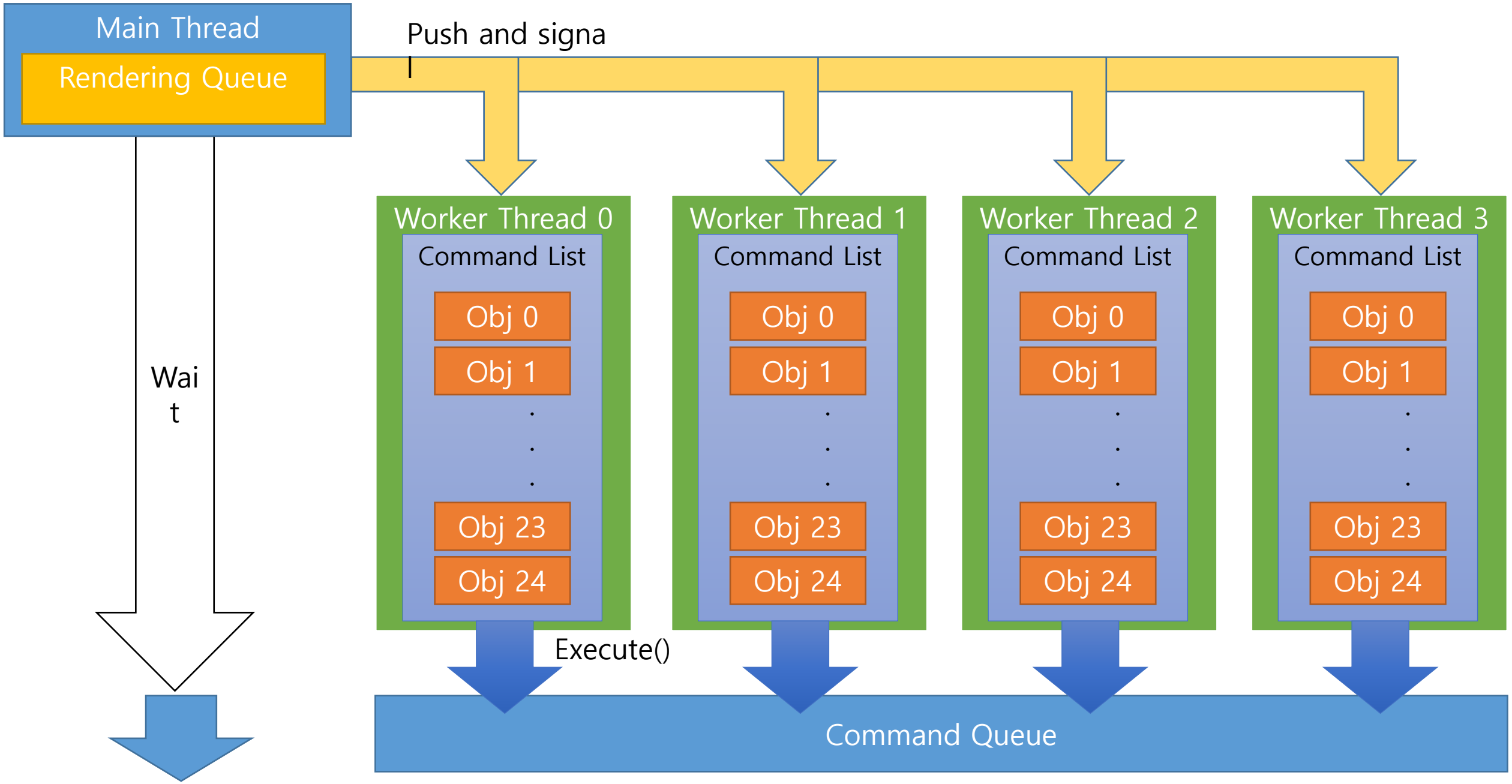
- D3D12에서 멀티 스레드 렌더링은 필수.
- 커맨드 리스트 작성에 걸리는 총시간을 줄일 수 있다.
- 더 빠른 속도를 위해서 위해서 (x)
- 말이 되는 속도를 얻기 위해서(o)

Multi-Thread Rendering 구현



Multi Thread 렌더링 - Blocking 방식

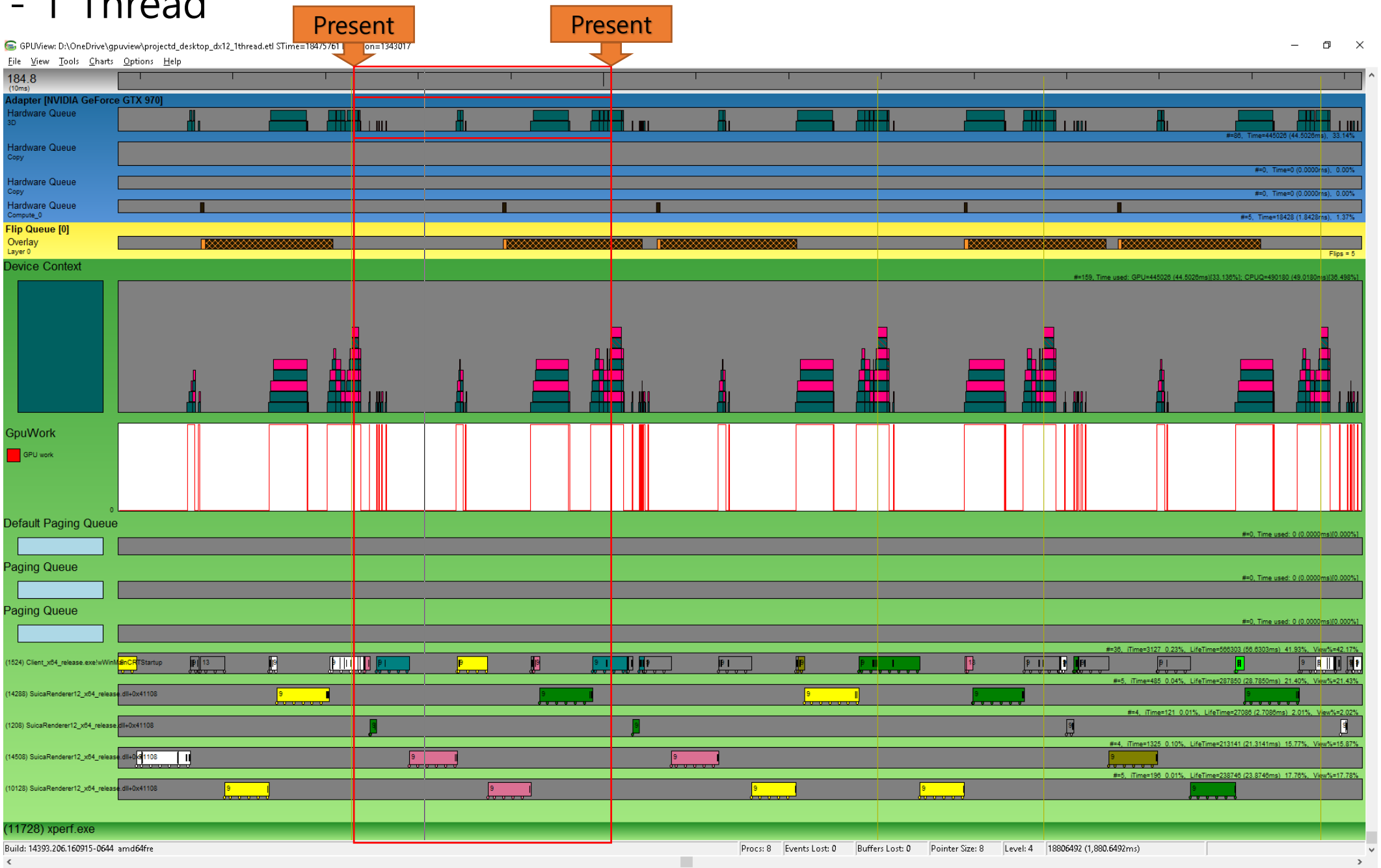
- 렌더링할 오브젝트를 Queue에 쌓아놨다가 일괄처리
- 단 시간 내에 GPU파워를 최대한 활용.
- 대신 커맨드 리스트 작성 시간동안 메인스레드는 wait.


Main Thread
Rendering Queue
Push and signa
l
Wai
t
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

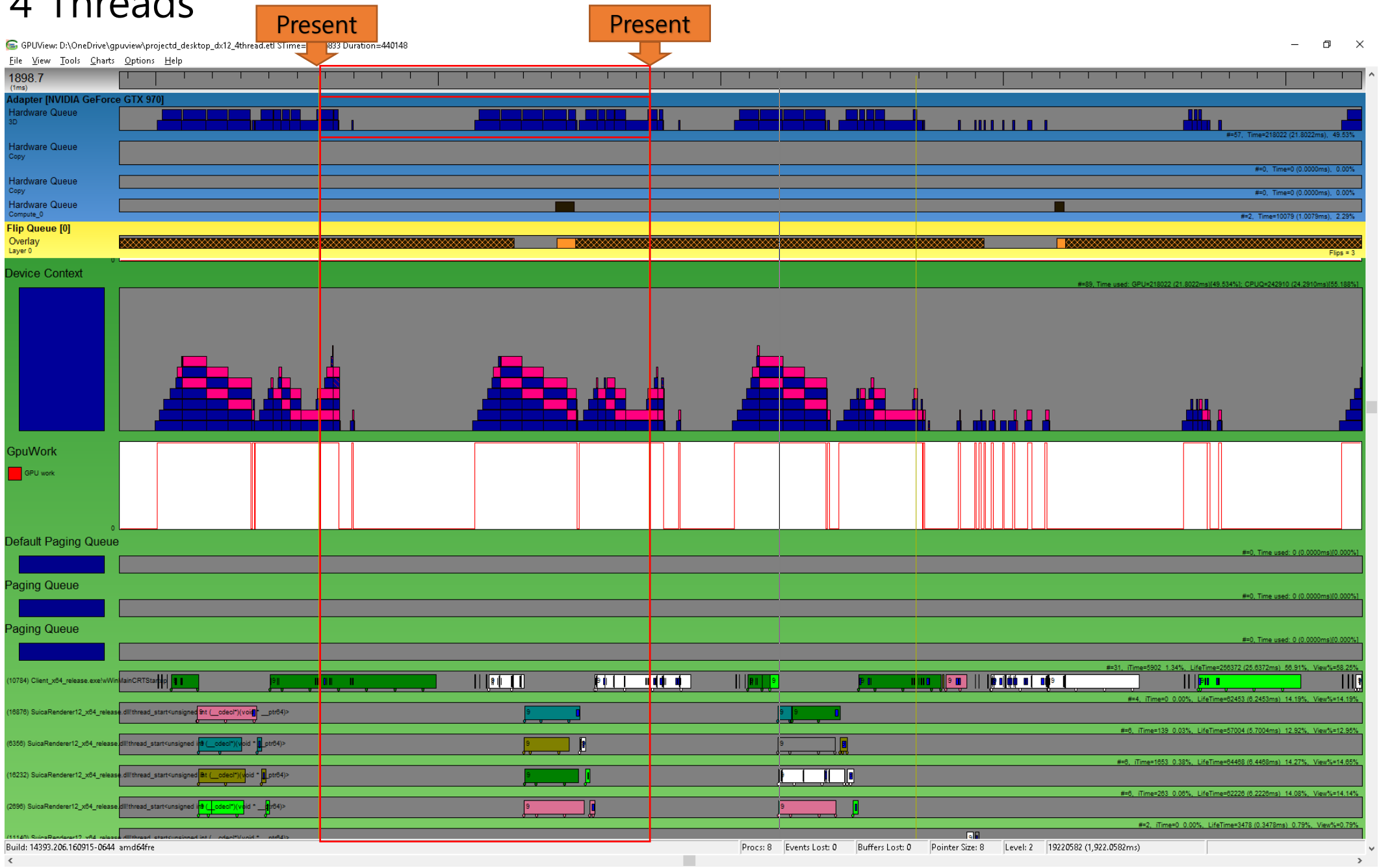
CPU Timeline(main thread)

GPU Timeline

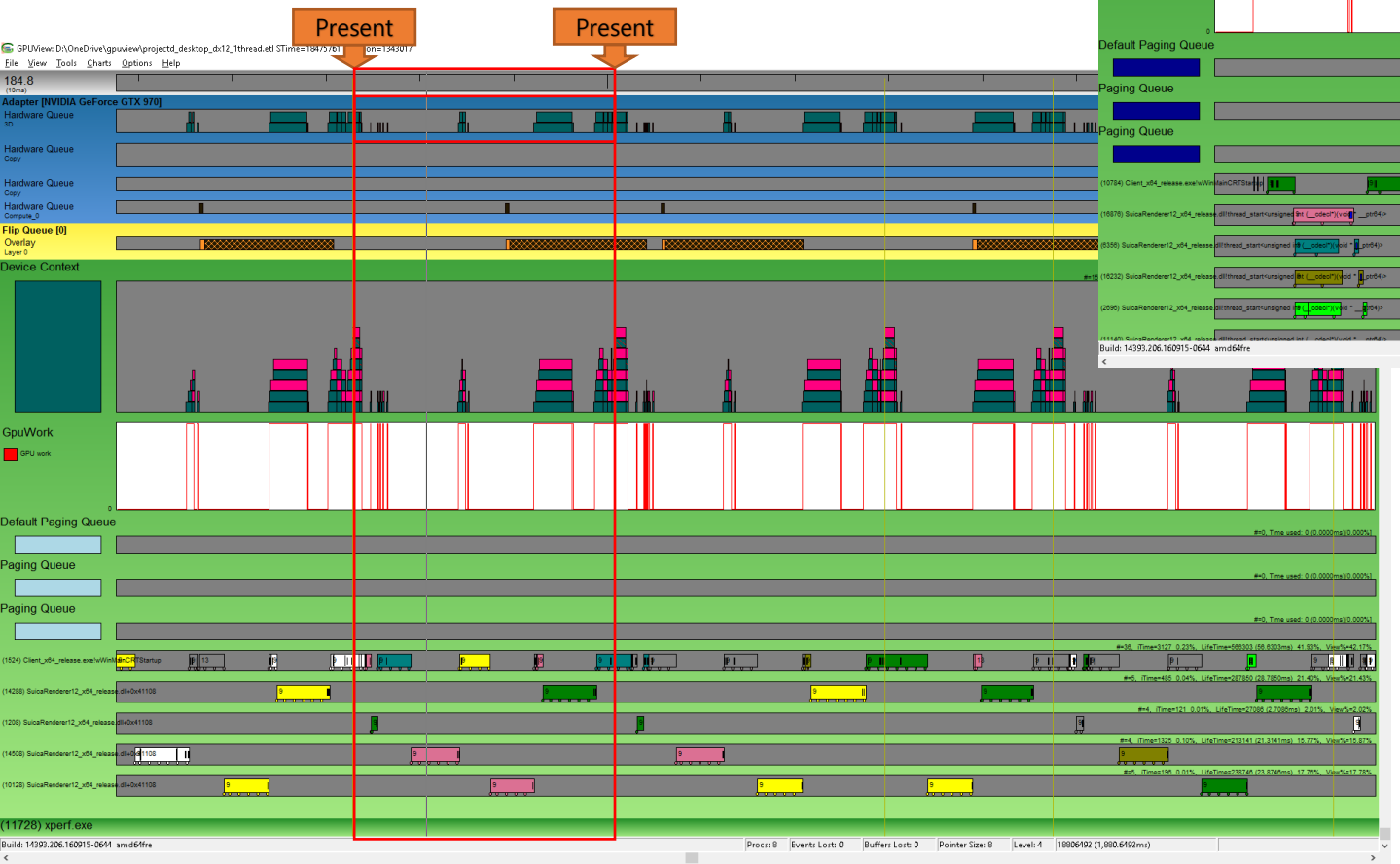
DX12 - 1 Thread

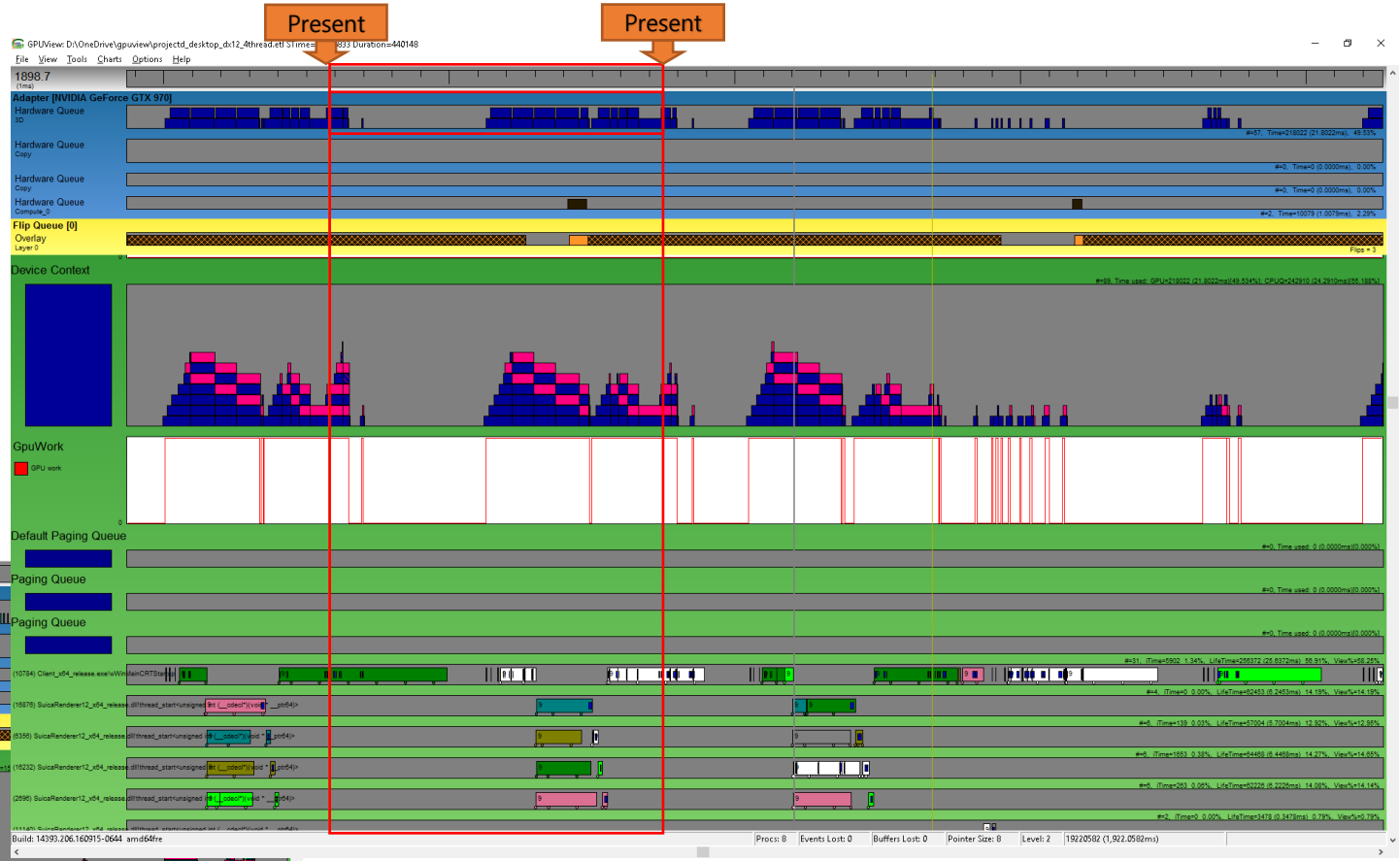


DX12 4 Threads



DX12 1 Threads – 42fps

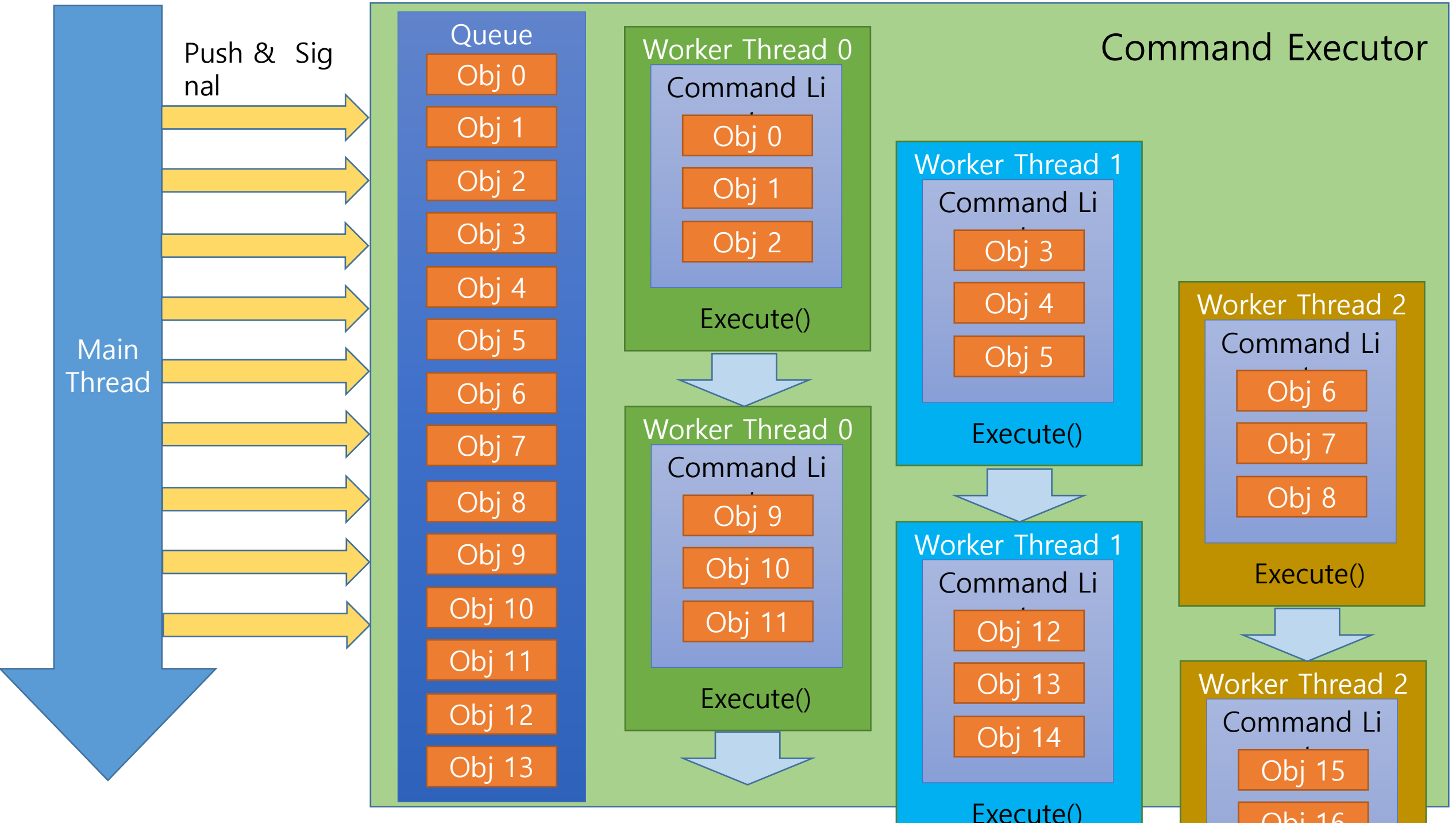




DX12 4 Threads – 107fps

Multi Thread 렌더링 - Non-blocking 방식

- Queue를 사용하지 않고 메인스레드의 요청에 따라 그 즉시 렌더링
- 메인 스레드는 wait하지 않는다.
- 코드가 복잡해짐.
- 스레드 동기화로 인한 성능 저하.


Command Executor
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
Worker Thread 0
Command Li
Obj 0
Obj 1
Obj 2
Execute()
Worker Thread 0
Command Li
Obj 9
Obj 10
Obj 11
Execute()
Worker Thread 1
Command Li
Obj 3
Obj 4
Obj 5
Execute()
Worker Thread 1
Command Li
Obj 12
Obj 13
Obj 14
Execute()
Worker Thread 2
Command Li
Obj 6
Obj 7
Obj 8
Execute()
Worker Thread 2
Command Li
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.

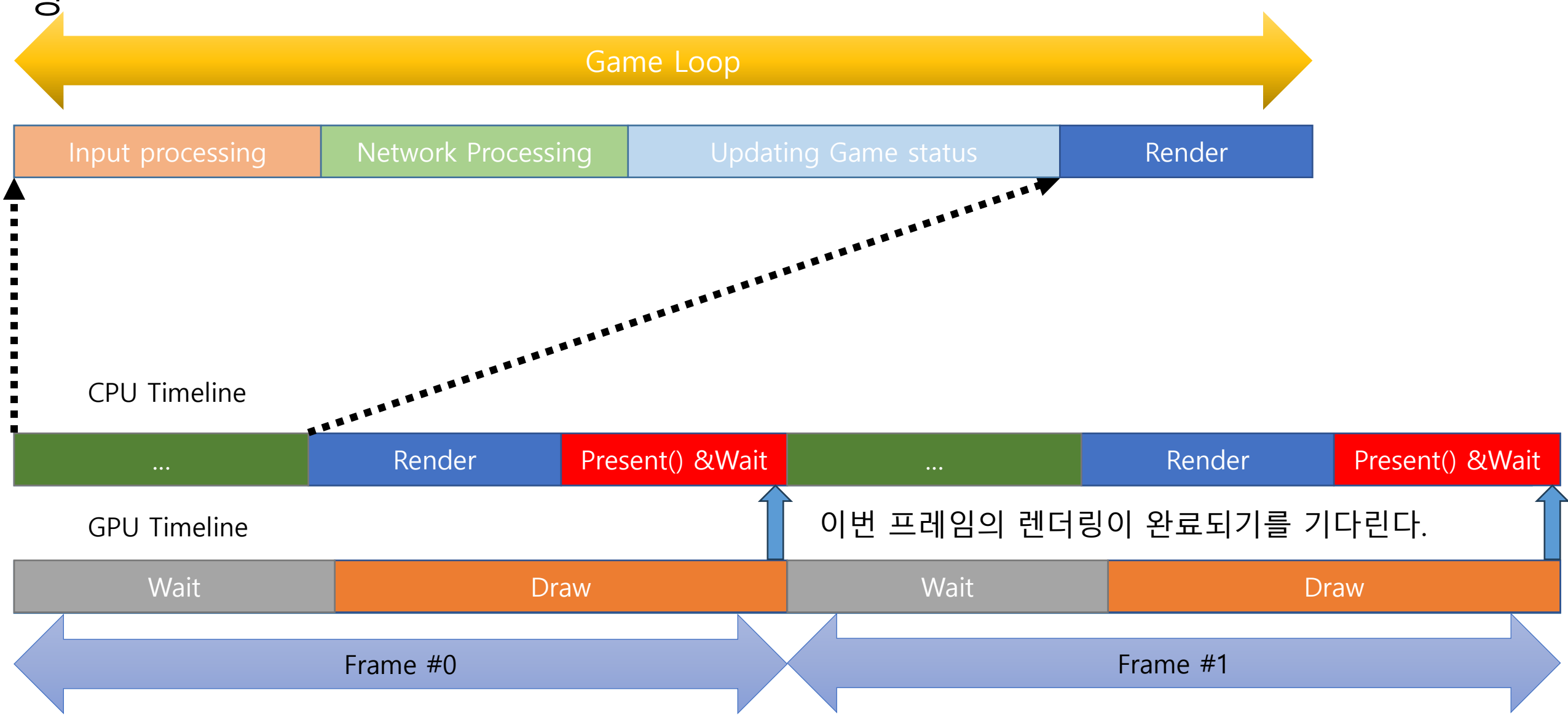
렌더링 스레드 개수

- 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 넘기지 않는 편이 좋다.

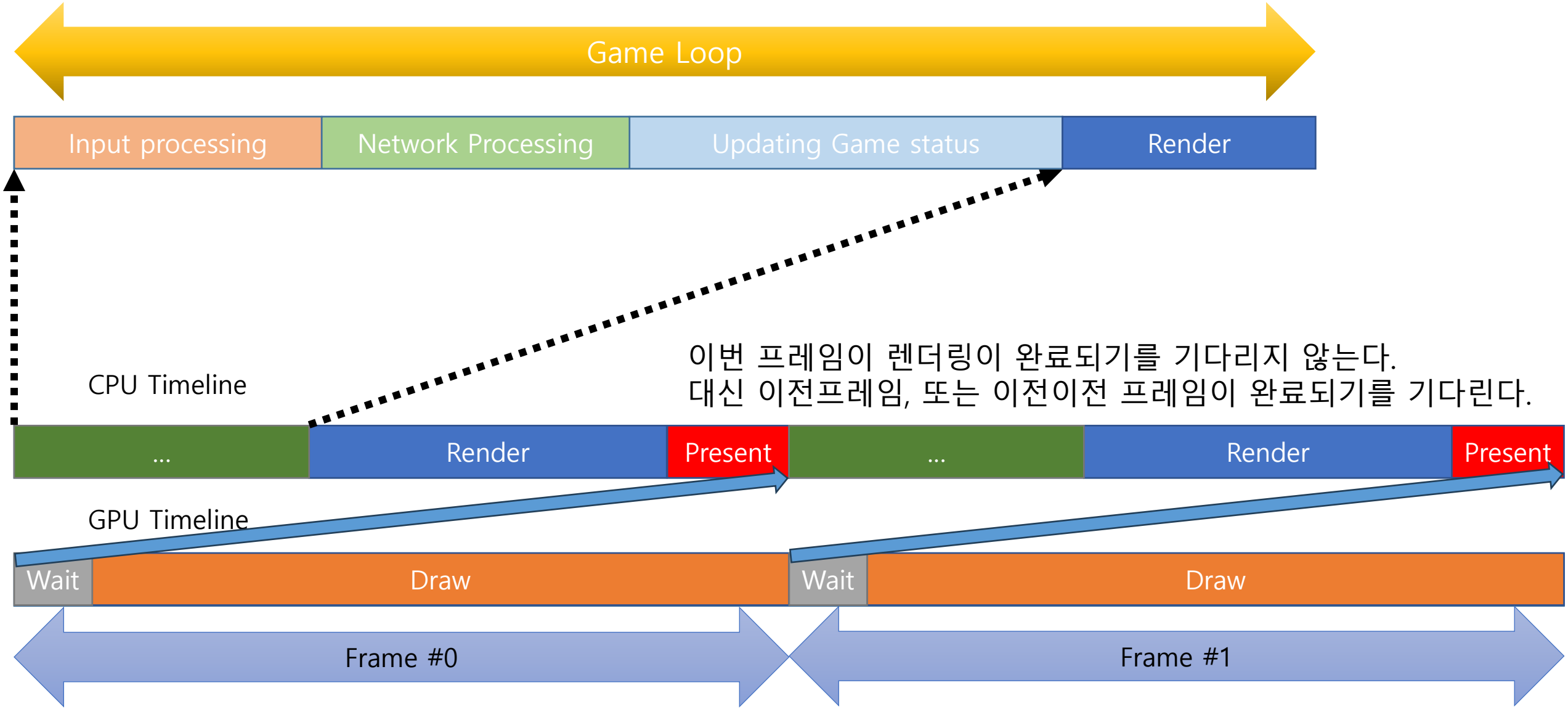
비동기식 중첩 렌더링

- 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 – 동기식 렌더링



게임 루프 내에서의 timeline – 비동기식 중첩 렌더링



게임 루프 내에서의 timeline – 비동기식 중첩 렌더링

CPU Timeline

Present & wait #0

Present & wait #1

Present & wait

GPU Timeline

성능 비교

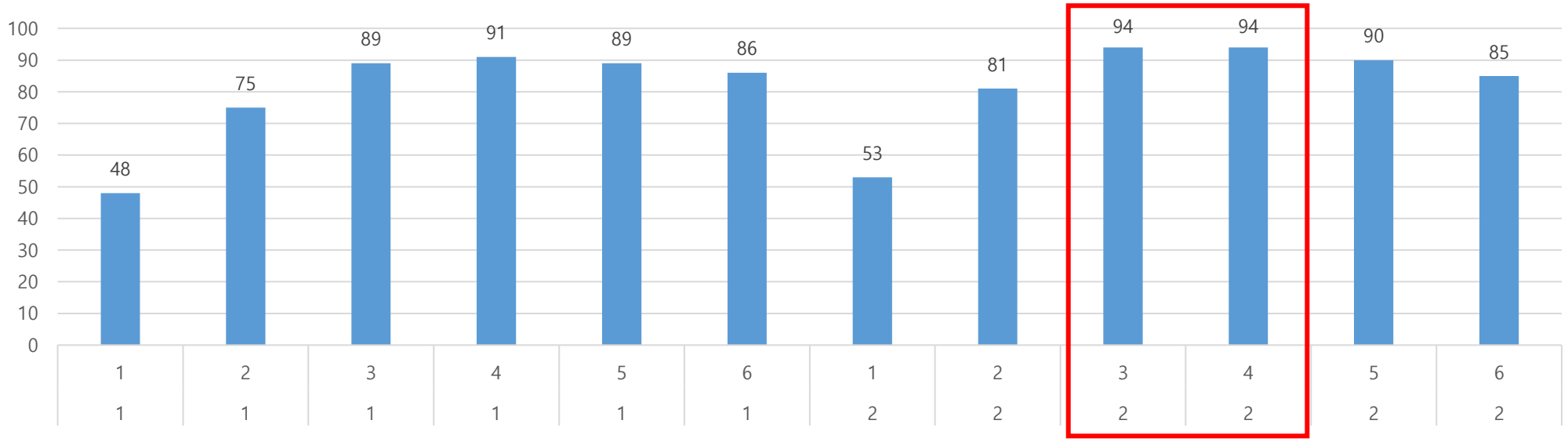
Rasterization – D3D11

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

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

Rasterization – D3D12

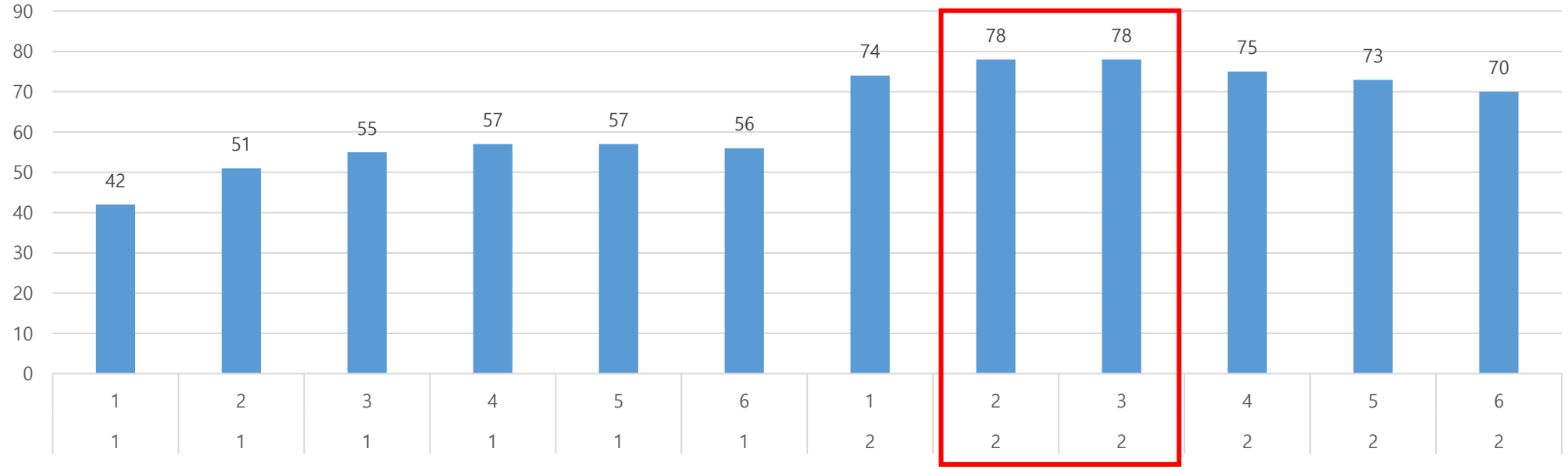
- Triangle Map
- Res : 1200x800
- HW Occ off , SW Occ off



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



D3D12 , 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

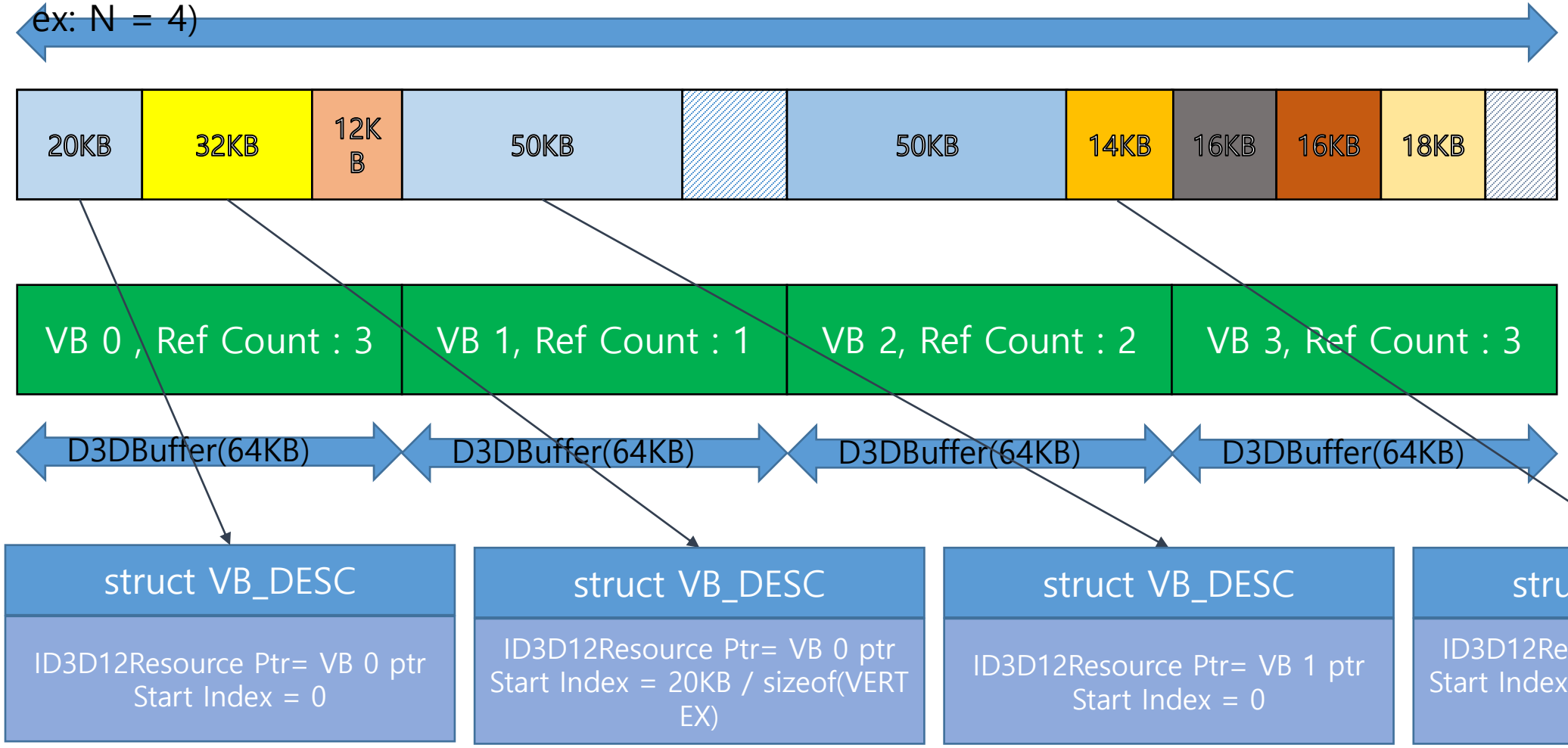
메모리는 왜 이리 많이 먹어?

- 64KB Align
- 한 개짜리 Vertex Buffer를 할당해도 최소 64KB
- 한 개짜리 Index Buffer를 할당해도 최소 64KB
- 4Bytes 짜리 Constant Buffer를 할당해도 최소 64KB.
- Buffer -> CBV로 사용할 땐 256KB Align 필요
- Bulk Memory로 할당해서 쪼개 사용할 것.

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 쪼개써야 할 필요가 있다.

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



- 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.

최적화 가이드

- GPU 점유율을 어떻게 높이는가, 즉 GPU 큐에 얼마나 작업을 꽉꽉 채워넣을지가 성능향상의 열쇠. DirectX Runtime, Driver로 인한 성능 향상은 꿈꾸지 말것.
- Execute()를 자주 호출하되, 너무 자주 호출하지 않는다(?)
- 하나의 Command List에 너무 많이 몰아서 Execute()하지 않는다.
- 되도록 Wait 하지않는다. 적어도 로딩 등 특수한 상황이 아닌 렌더링 중에는 절대로 wait하지 않는다.
- 성능 향상을 위해서 중첩 프레임 렌더링을 반드시 지원해야 한다.
- 성능 향상을 위해서 멀티 스레드 렌더링을 반드시 지원해야 한다.
- 64KB Align에 따른 메모리 낭비를 막기 위해 큰 덩어리의 D3D12Resource를 쪼개쓰는 자료구조를 만들어야 한다.

D3D12 최적화에 대한 좀더 깊은 내용은 다음을 참고할 것

- DirectX 12 프로그래밍에 대한 고찰 (youtube.com)
  - https://www.youtube.com/watch?v=sWyTNC1vDLk
- D3D12 게임 프로젝트 로딩 성능 개선 (youtube.com)
  - https://www.youtube.com/watch?v=R_sMY_jjZnE

D3D12프로그래밍을 위한 조언

- D3D를 아예 안할거면 모르되, 계속 할거면 D3D12는 필수.
- D3D12로 포팅만 하면 11보다 빠를거라고 생각하면 큰 착각.
- D3D12를 학습하려면 D3D11선학습이 필수인가요?
  - 하면 할 수 있을것 같은데 아직 D3D12부터 시작해서 성공한 케이스를 한 번도 못봤음.
- CPU/BUS시스템/GPU에 대한 기초적인 하드웨어 지식이 필요하다.

Q/A