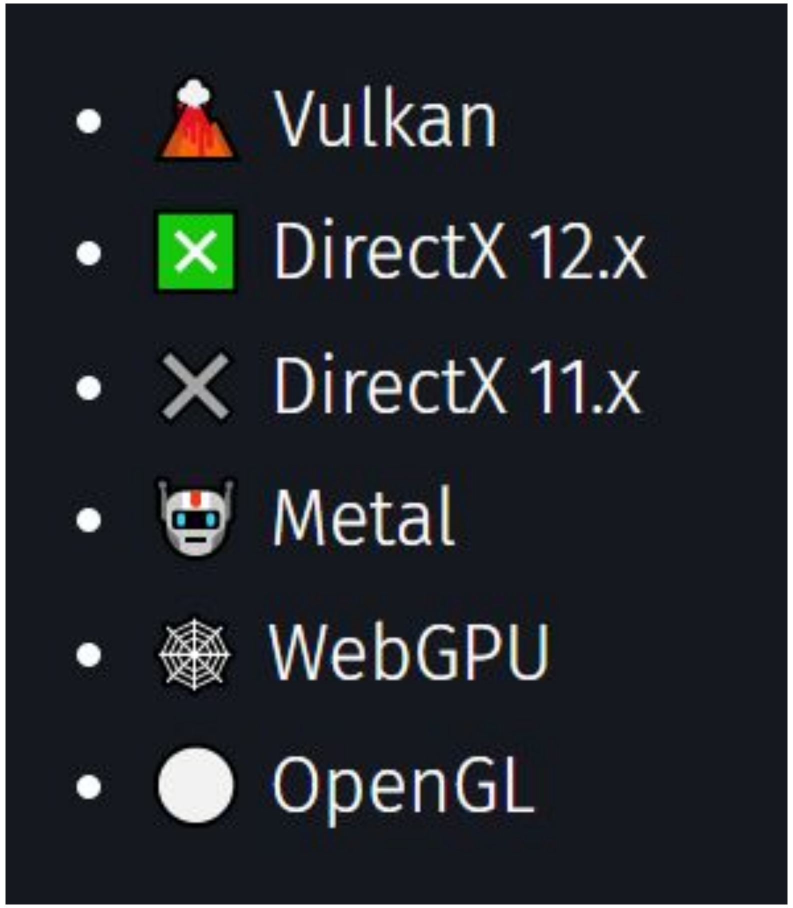
Differences between dx11 and dx12

Zhu Zekun 2022.12

Index

What is Direct X

DirectX is a series of application programming interfaces (API) that provide low-level access to hardware components like video cards, the sound card, and memory.



- 🌋 Vulkan
- ❎ DirectX 12.x
- ✖ DirectX 11.x
- 🤖 Metal
- 🕸 WebGPU
- ⚪ OpenGL

What is Direct X 12

- "DX12's focus is on enabling a dramatic increase in visual richness through a significant decrease in API-related CPU overhead. DX12 gives the application the ability to directly manage resources and state, and perform necessary synchronization. As a result, developers of advanced applications can efficiently control the GPU, taking advantage of their intimate knowledge of the game's behavior."

---Henry Moreton(Nvidia)

Direct X 12 Components

Direct3D	**A low-level API for 3d rendering.**
Direct2D	**A 2d vector graphics API.**
DirectWrite	**A device-independent text layout system.**
DXGI	A DirectX Graphics Infrastructure.
DirectInput	An API for input devices.
XAudio2	A low-level audio API.

Direct3D 12 advantages

Direct3D 12 enables richer scenes, more objects, more complex effects, and full utilization of modern GPU hardware.

- Vastly reduced CPU overhead.

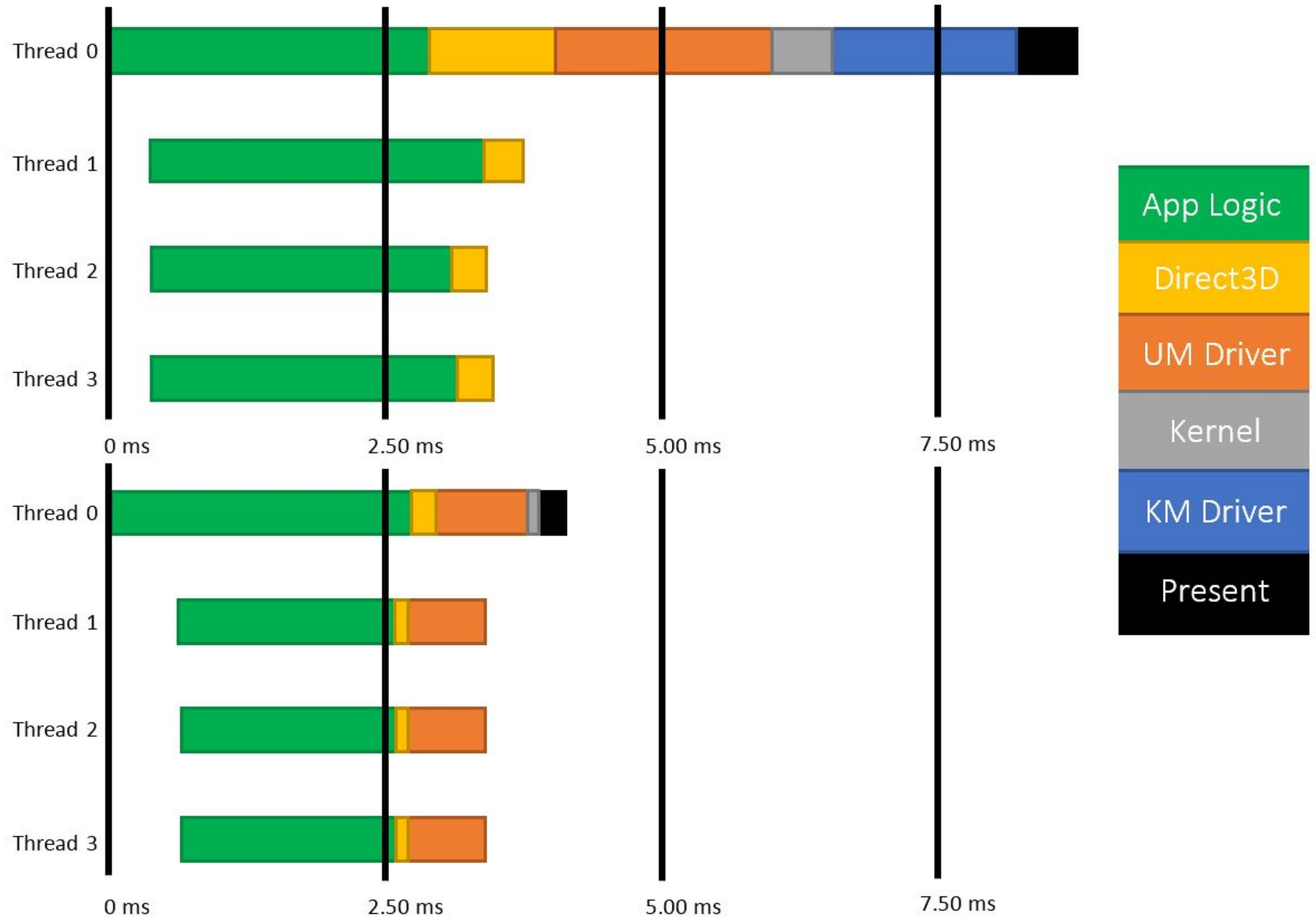
- Significantly reduced power consumption.

- Up to (approximately) 20% improvement in GPU efficiency.

- Cross-platform development for a Windows 10 device (PC, tablet, console, mobile).

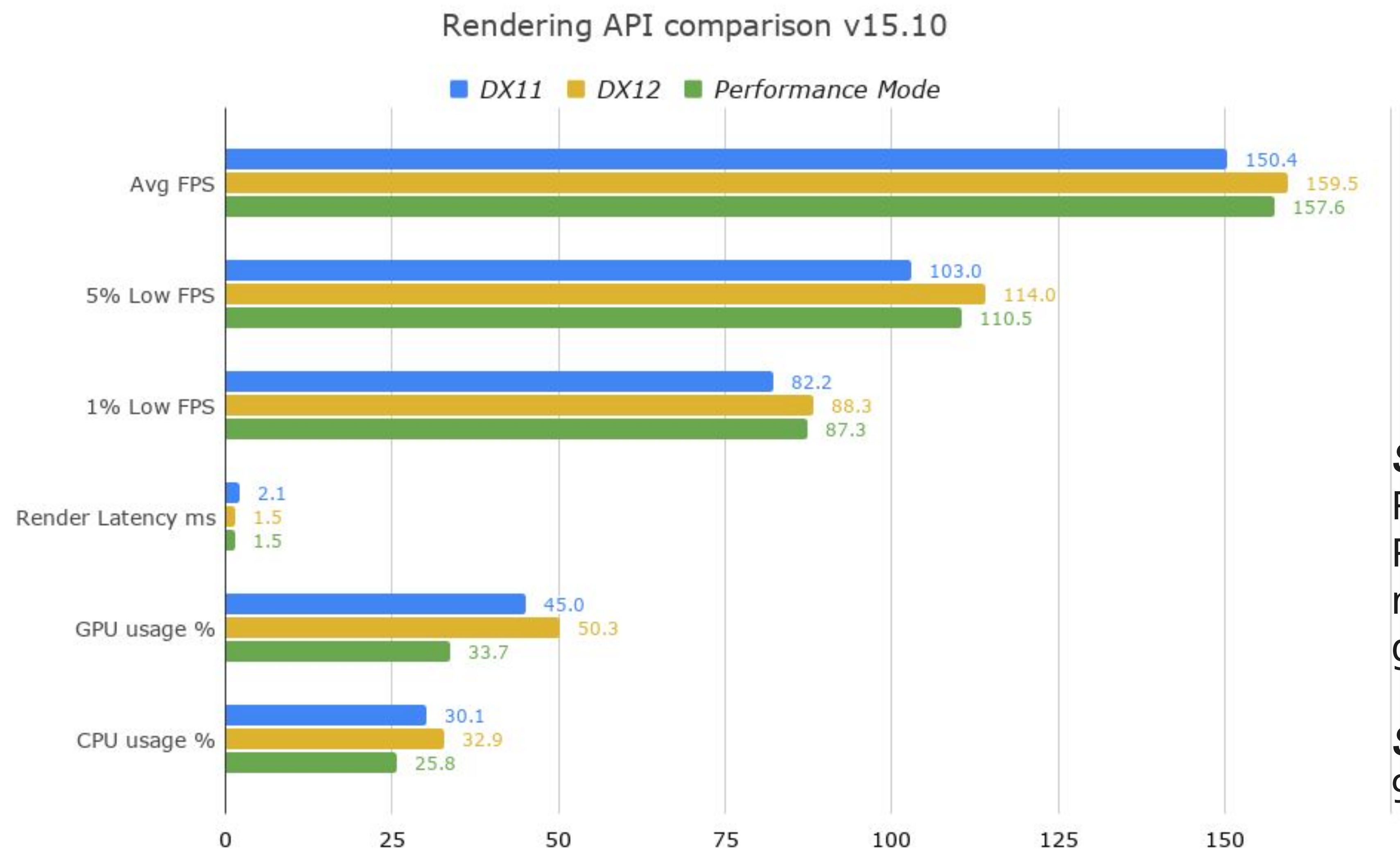
Direct3D 12 advantages

Thread work comparison



Direct3D 12 advantages

FPS comparison



Sample
Frosty Frenzy tournament game in Replay mode, 50 players first moving zone until the end of the game.

Specs
9900K, RTX 2060, 2*8 GB RAM.

Direct3D 12 Messiah

#8347 DirectX 12支持: https://km.netease.com/wiki/723/page/78292

渲染效果：dx12各项渲染，后处理效果功能正常，SSR和MotionBlur效果对比dx11更佳，其余无明显区别。

UI: dx12各UI功能正常，与dx11对比无明显区别。

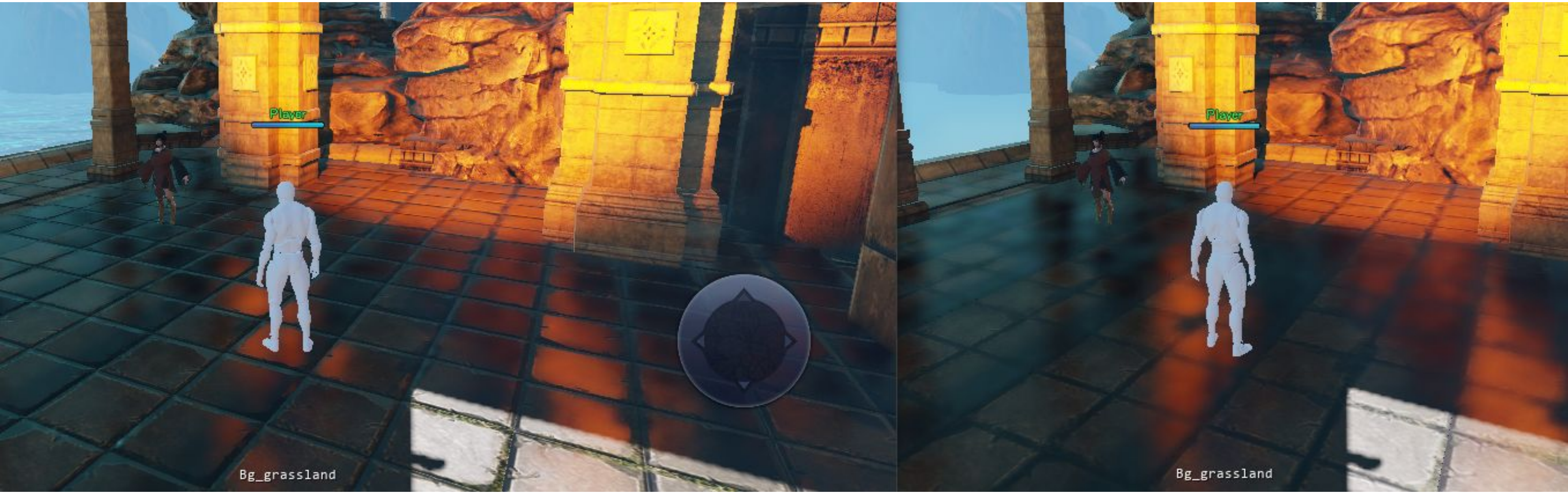
模型：dx12加载、卸载模型功能正常，暴力测试加载、卸载多个模型，无内存泄漏。

材质：dx12各材质效果正常，与dx11对比无明显区别。

性能：dx12对比dx11，CPU占用大幅减少，但是内存占用大小有增加(根据场景不同可能增加几十兆甚至几百兆)，另外不限帧时，帧率反而比dx11低

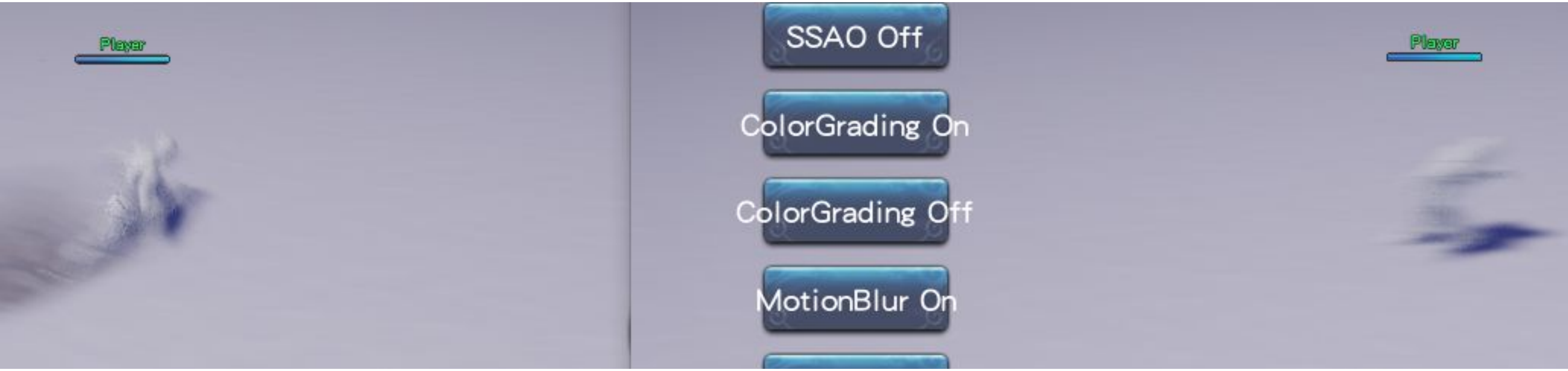
Direct3D 12 Messiah

SSR dx12(左) 与dx11（右）效果对比



Direct3D 12 Messiah

MotionBlur dx12(左) 与dx11（右）效果对比



Direct3D 12 initialization

1. Create the ID3D12Device using the D3D12CreateDevice function.
2. Create an ID3D12Fence object and query descriptor sizes.
3. Check 4X MSAA quality level support.
4. Create the command queue, command list allocator, and main command list.
5. Describe and create the swap chain.
6. Create the descriptor heaps the application requires.
7. Resize the back buffer and create a render target view to the back buffer.
8. Create the depth/stencil buffer and its associated depth/stencil view.
9. Set the viewport and scissor rectangles.

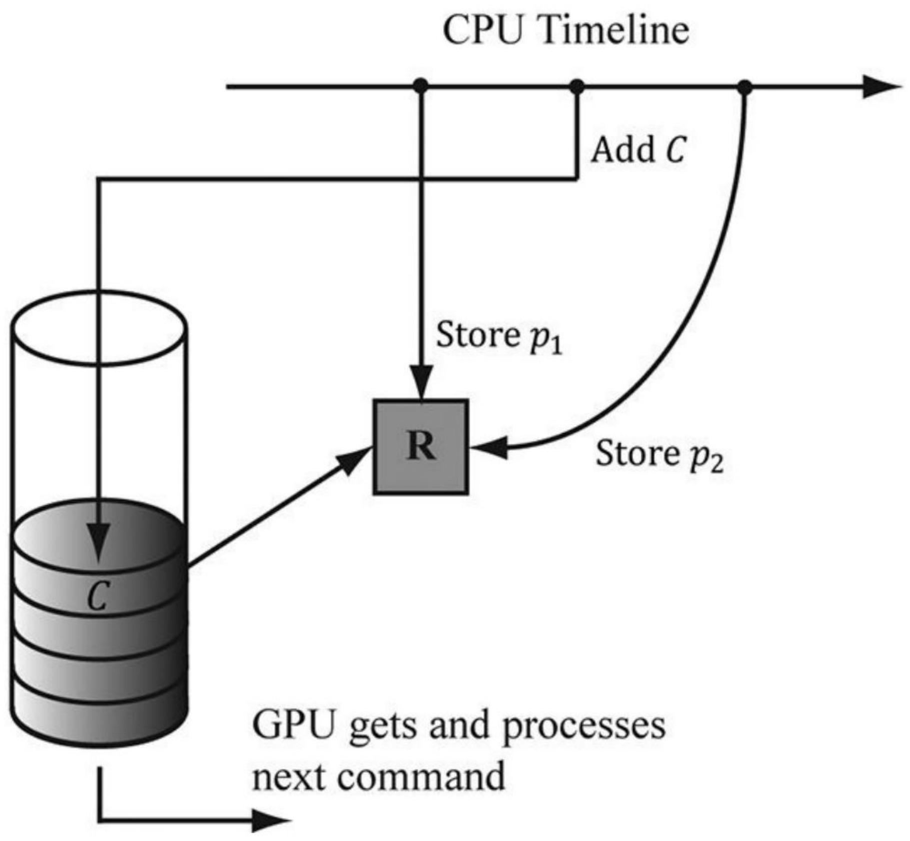
1 Create the Device创建设备

The device represents a display adapter. Usually it is a physical piece of 3D hardware.

DX12 device is used to check feature support, and create all other D3D interface objs like resources, views and command lists.

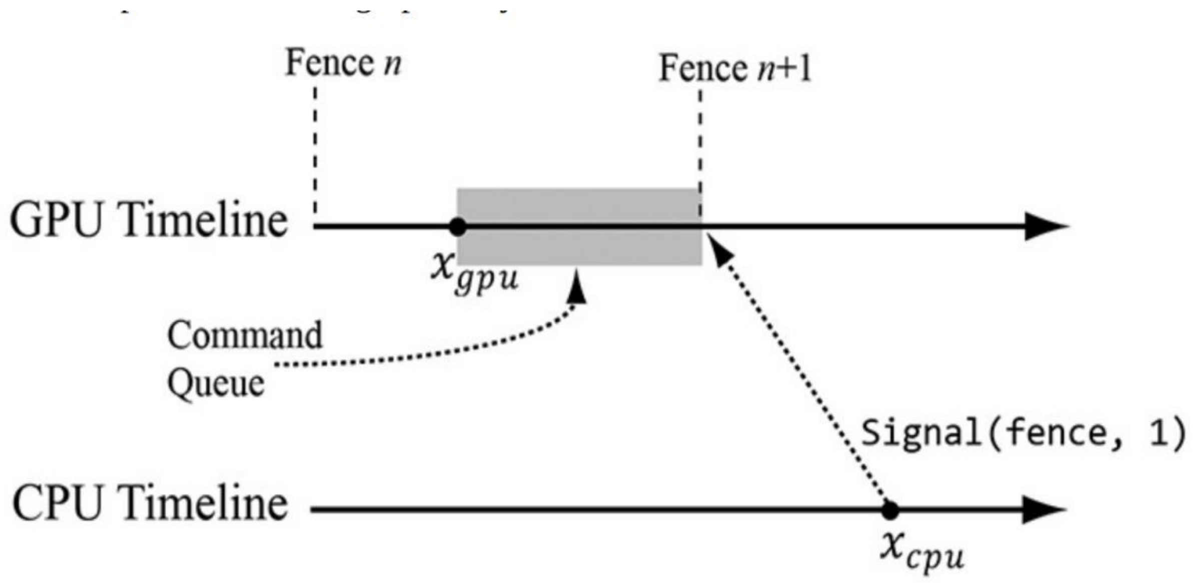
2 Create the Fence and Descriptor Sizes创建fence

We need to create Fence object for GPU/CPU synchronization.
We need to query Descriptor sizes. We cache it so that it is available when we need it for various descriptor types.

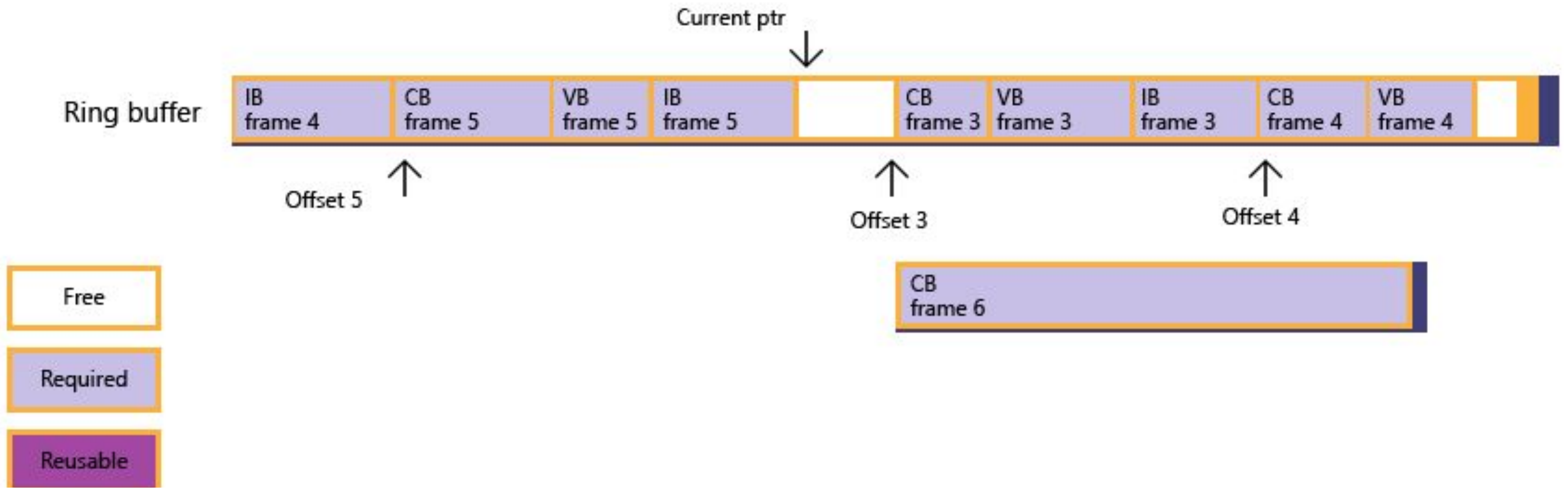


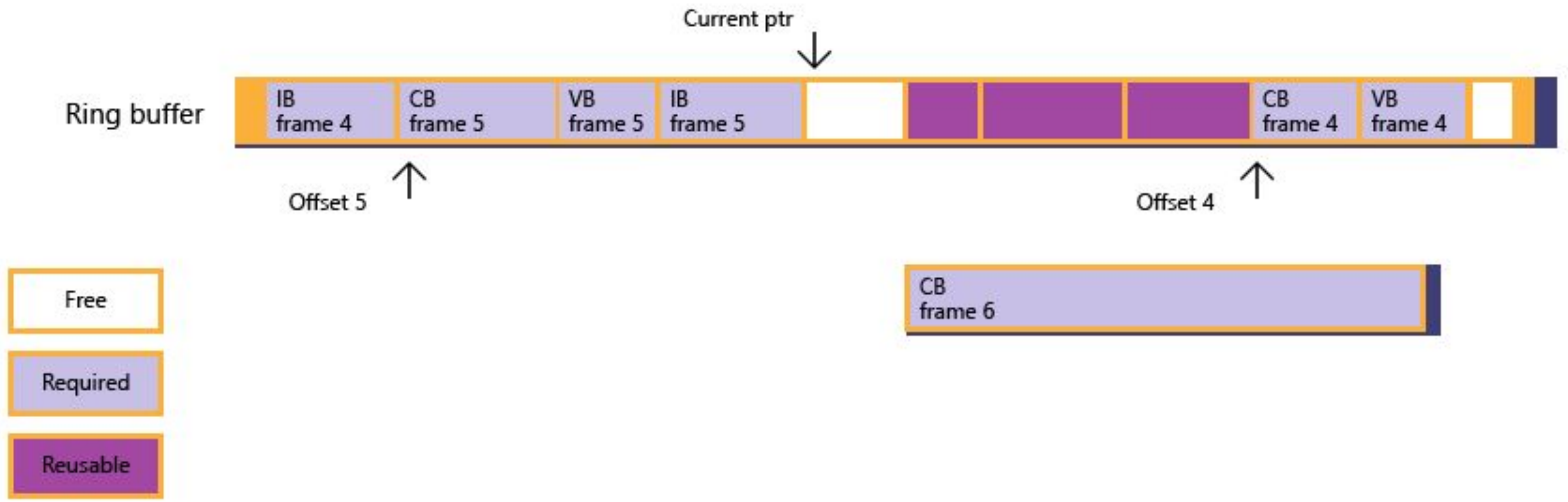
Fence: ID3D12Fence interface. It is used to synchronize the GPU and CPU.
md3dDevice->CreateFence(0,D3D12_FENCE_FLAG_NONE, IID_PPV_ARGS(&mFence));
A fence object maintains a UINT64 value, an integer to identify a fence point at a time.

You can flush the command queue to execute the initialization.
->To make sure all the GPU commands have been executed before.

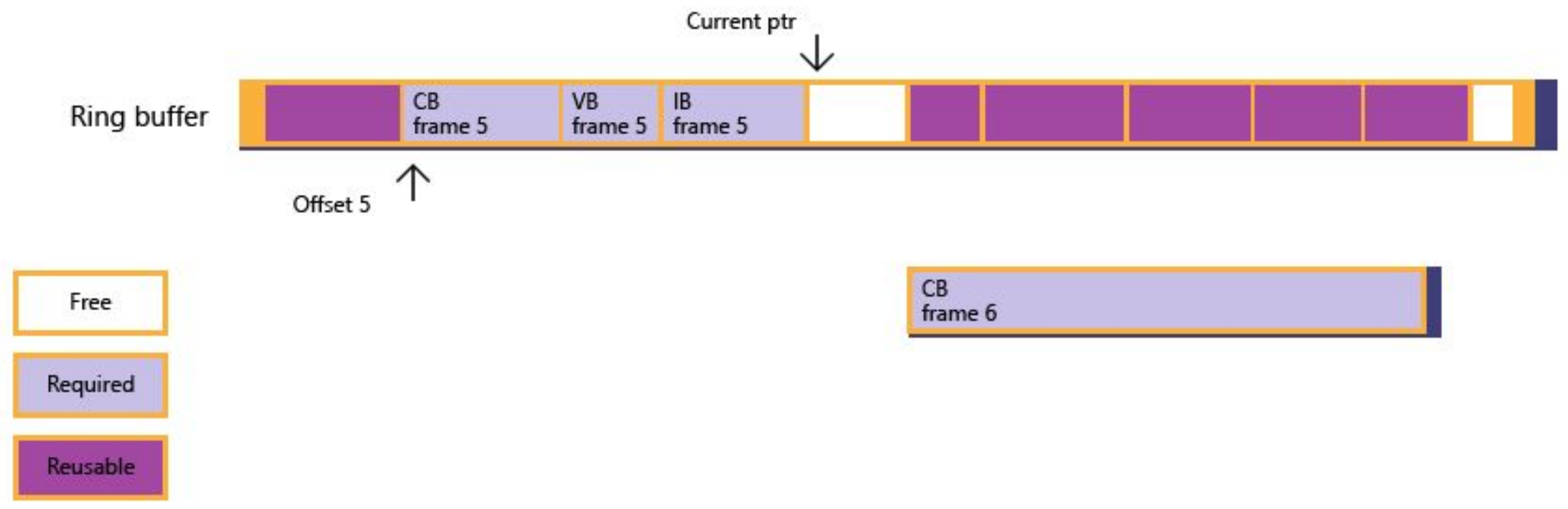


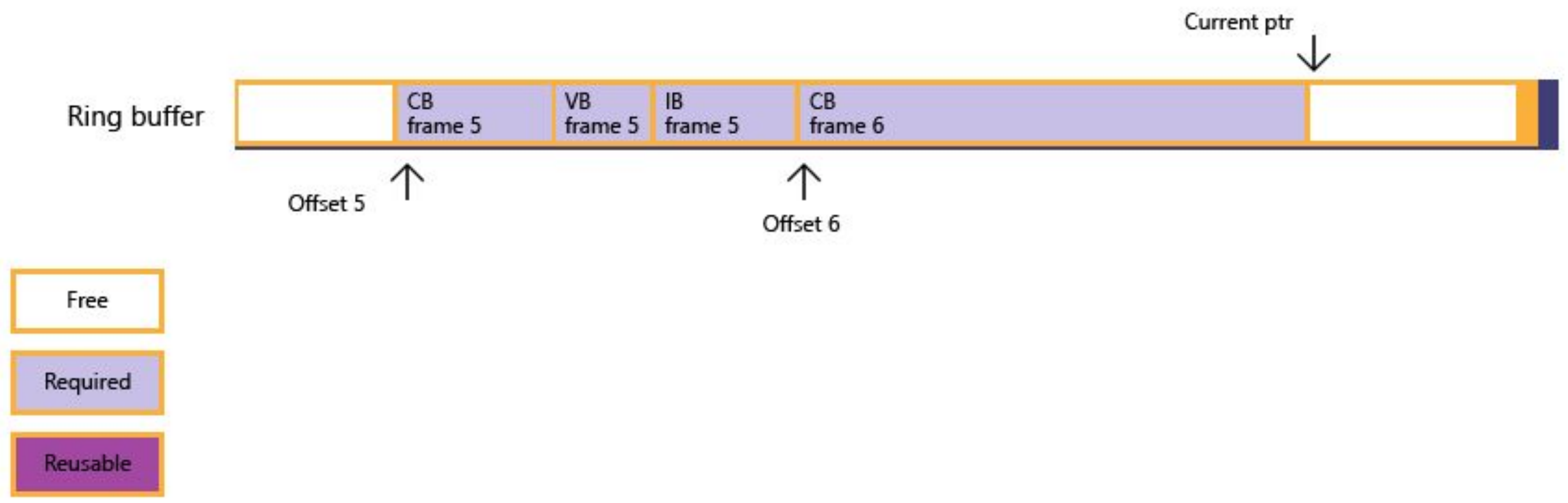
Fence-Based Resource Management



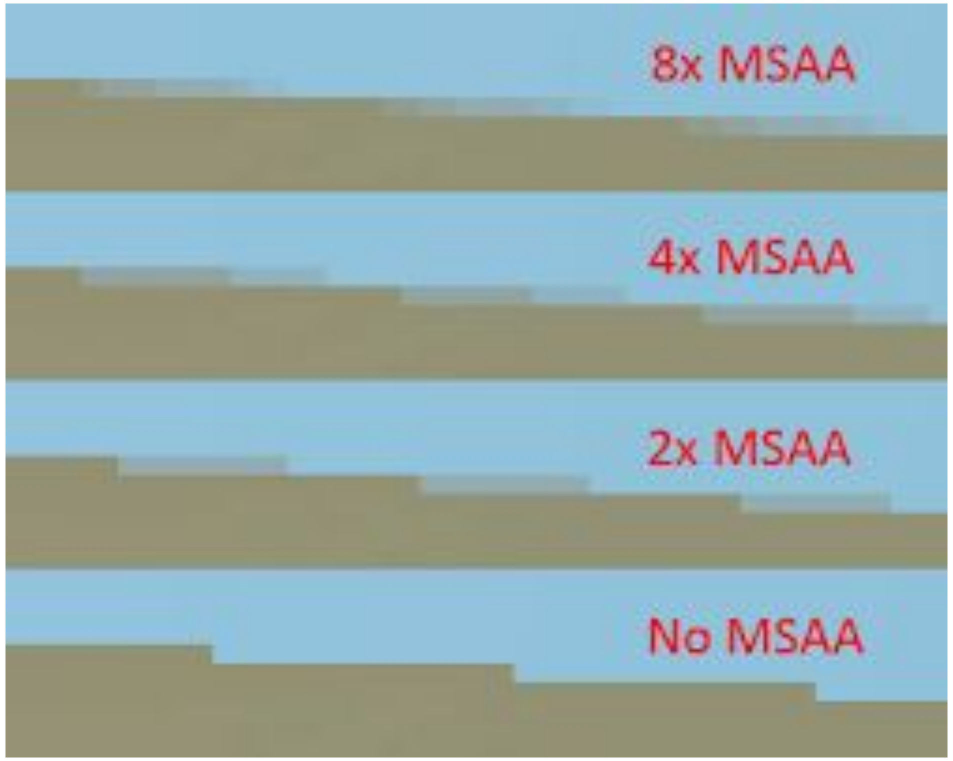


Fence-Based Resource Management

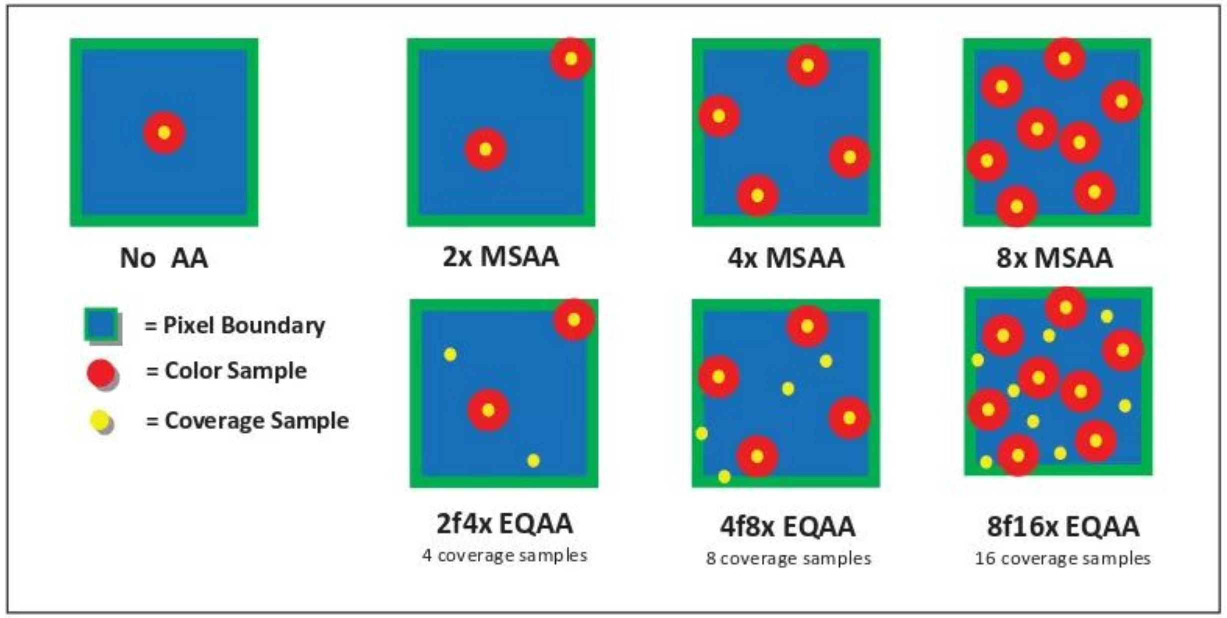




3 Check 4X MSAA Quality Support

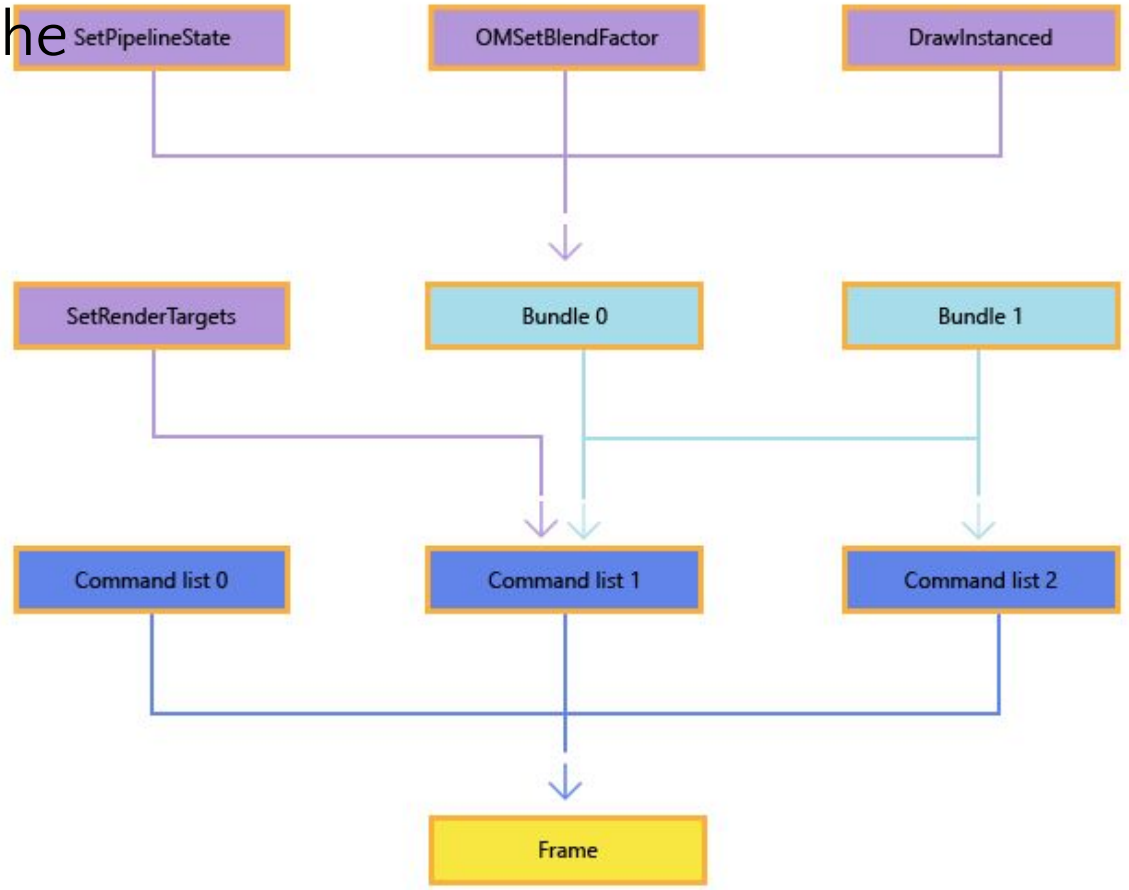


EQAA Modes for AMD Graphics Cards



4 Create Command Queue and Command List创建命令队列和列表

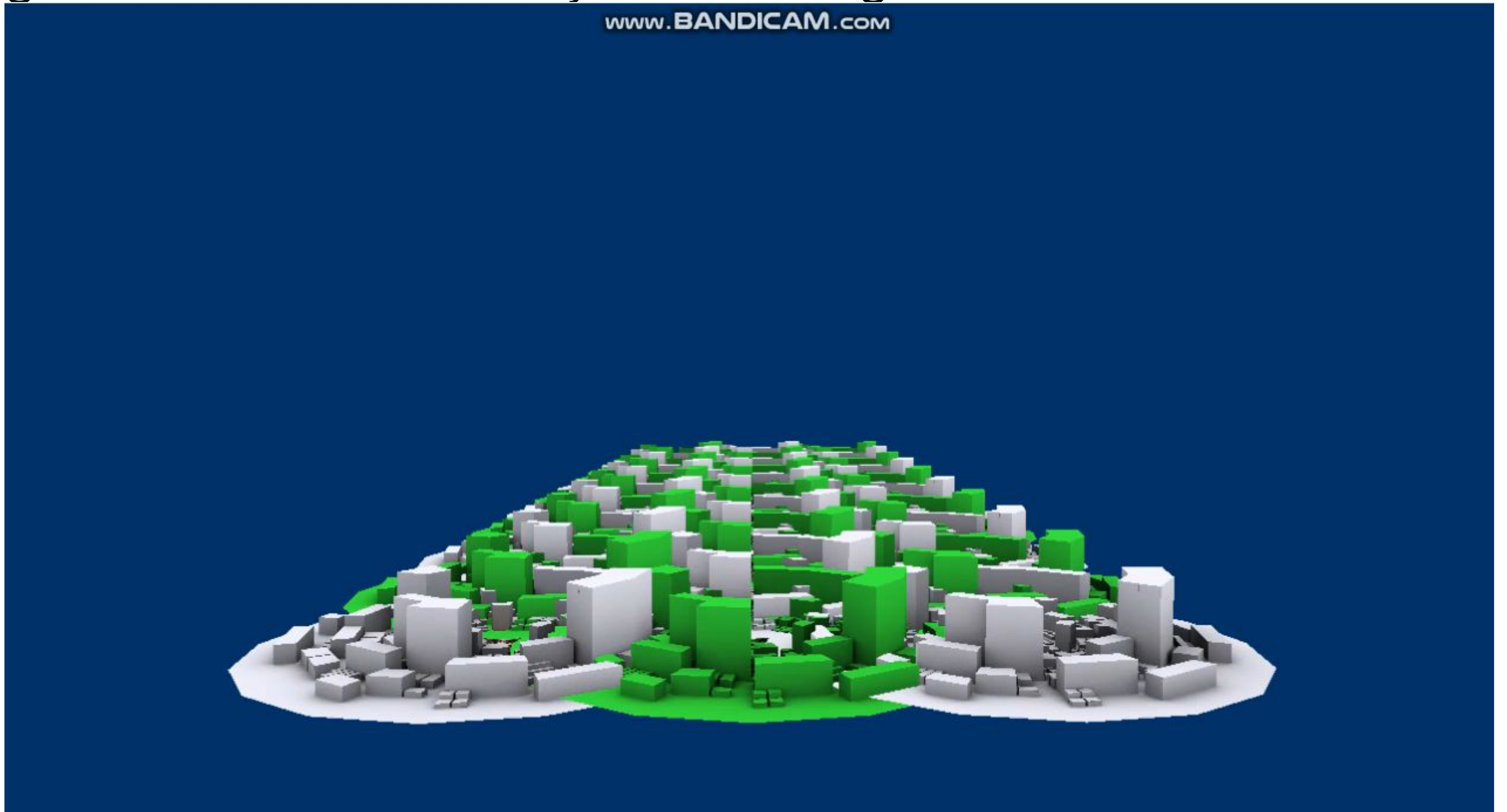
A **Command List** is used to issue copy, compute (dispatch), or draw commands. In DirectX 12 all command lists are deferred; that is, the commands in a command list are only run on the GPU after they have been executed on a command queue



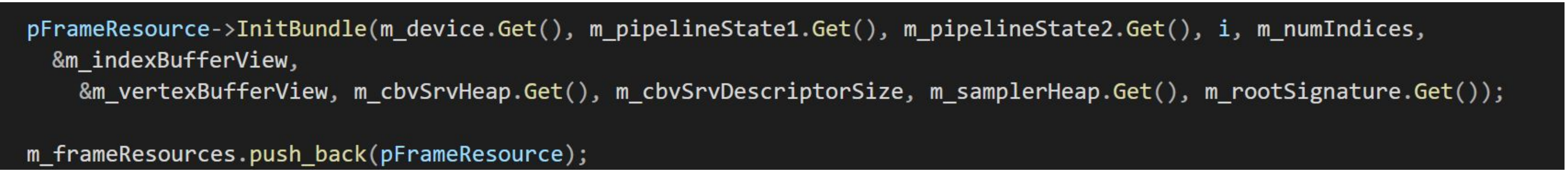
The **Command Queue** in DirectX 12 has a very simple interface. For most common cases only the ExecuteCommandLists method and the Signal method are used.

3.4 Create Command Queue and Command List: Bundle

Bundles
Nice way to submit work early in the frame
Nothing inherently faster about bundles on the GPU
Inherits state from calling Command List – use to your advantage

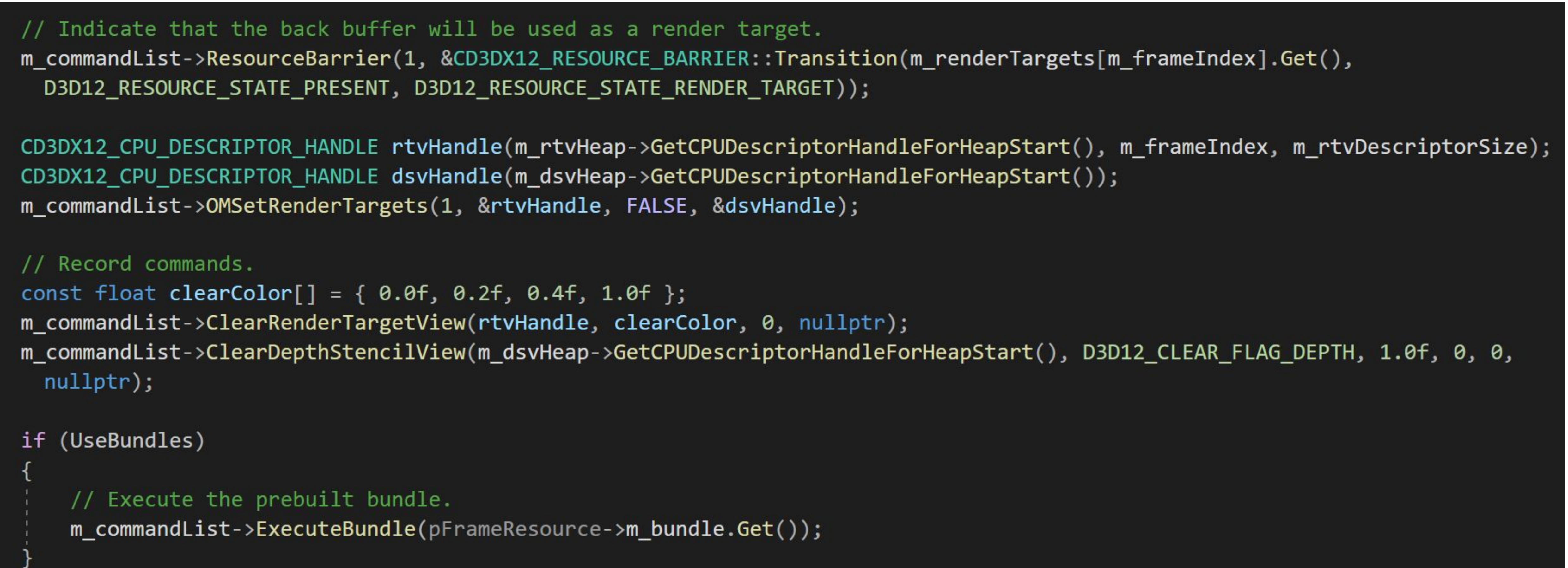


4 Create Command Queue and Command List: Bundle



```
pFrameResource->InitBundle(m_device.Get(), m_pipelineState1.Get(), m_pipelineState2.Get(), i, m_numIndices,
  &m_indexBufferView,
    &m_vertexBufferView, m_cbvSrvHeap.Get(), m_cbvSrvDescriptorSize, m_samplerHeap.Get(), m_rootSignature.Get());

m_frameResources.push_back(pFrameResource);
```



```
// Indicate that the back buffer will be used as a render target.
m_commandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(m_renderTargets[m_frameIndex].Get(),
  D3D12_RESOURCE_STATE_PRESENT, D3D12_RESOURCE_STATE_RENDER_TARGET));

CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHandle(m_rtvHeap->GetCPUDescriptorHandleForHeapStart(), m_frameIndex, m_rtvDescriptorSize);
CD3DX12_CPU_DESCRIPTOR_HANDLE dsvHandle(m_dsvHeap->GetCPUDescriptorHandleForHeapStart());
m_commandList->OMSetRenderTargets(1, &rtvHandle, FALSE, &dsvHandle);

// Record commands.
const float clearColor[] = { 0.0f, 0.2f, 0.4f, 1.0f };
m_commandList->ClearRenderTargetView(rtvHandle, clearColor, 0, nullptr);
m_commandList->ClearDepthStencilView(m_dsvHeap->GetCPUDescriptorHandleForHeapStart(), D3D12_CLEAR_FLAG_DEPTH, 1.0f, 0, 0,
  nullptr);

if (UseBundles)
{
    // Execute the prebuilt bundle.
    m_commandList->ExecuteBundle(pFrameResource->m_bundle.Get());
}
```

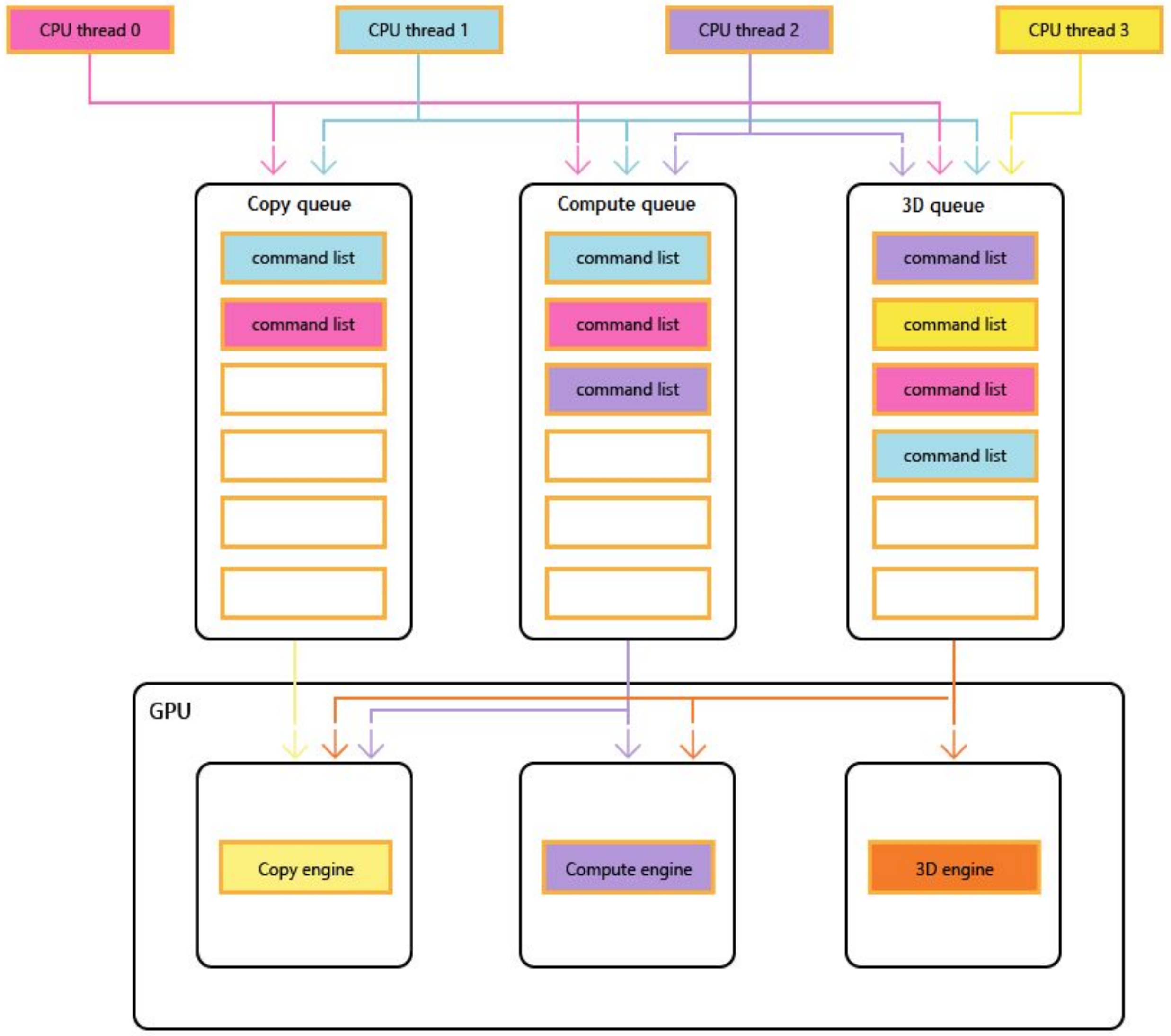
4 Create Command Queue and Command List

Each command queue must track it’s own fence and fence value.

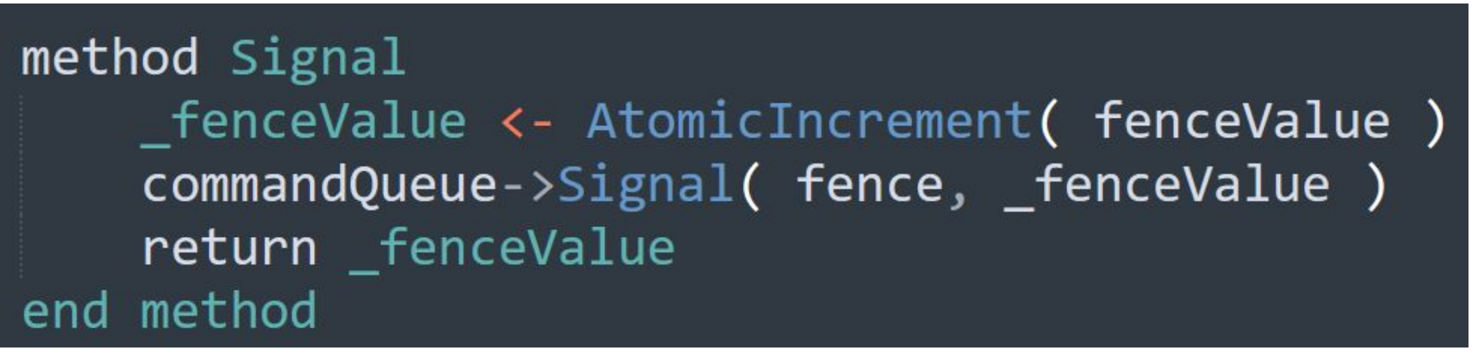
Copy: Can be used to issue commands to copy resource data (CPU -> GPU, GPU -> GPU, GPU -> CPU).

Compute: Can do everything a Copy queue can do and issue compute (dispatch) commands.

Direct: Can do everything a Copy and a Compute queue can do and issue draw commands.

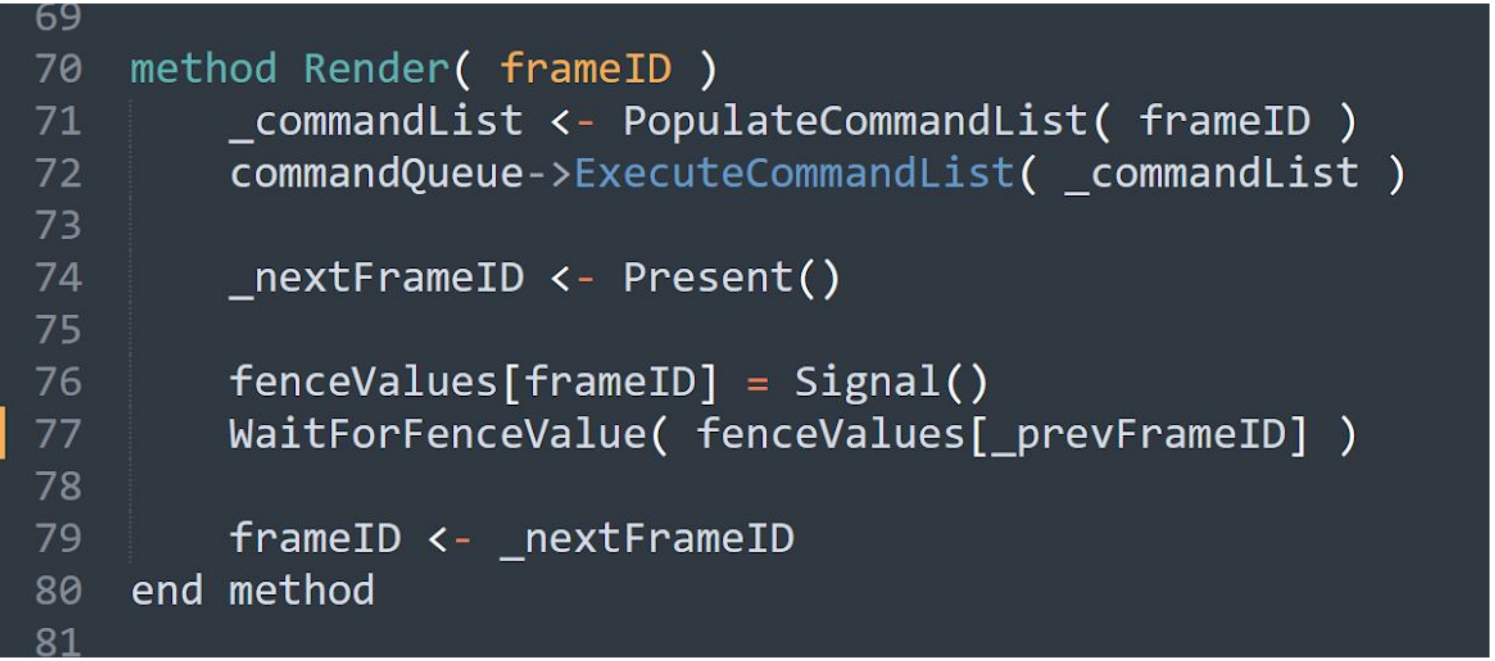


4 Create Command Queue and Command List



```
method Signal
    _fenceValue <- AtomicIncrement( fenceValue )
    commandQueue->Signal( fence, _fenceValue )
    return _fenceValue
end method
```

Signal: Insert a fence value into the command queue. The fence used to signal the command queue will have it's completed value set when that value is reached in the command queue.



```
method Render( frameID )
    _commandList <- PopulateCommandList( frameID )
    commandQueue->ExecuteCommandList( _commandList )

    _nextFrameID <- Present()

    fenceValues[frameID] = Signal()
    WaitForFenceValue( fenceValues[_prevFrameID] )

    frameID <- _nextFrameID
end method
```

Render a frame. Do not move on to the next frame until that frame's previous fence value has been reached.

4 Create Command Queue and Command List

Guidance:

Command List:

Use multiple threads to construct CLs in paralle

Do not execute too many CLs every frame:

 15–30 command list

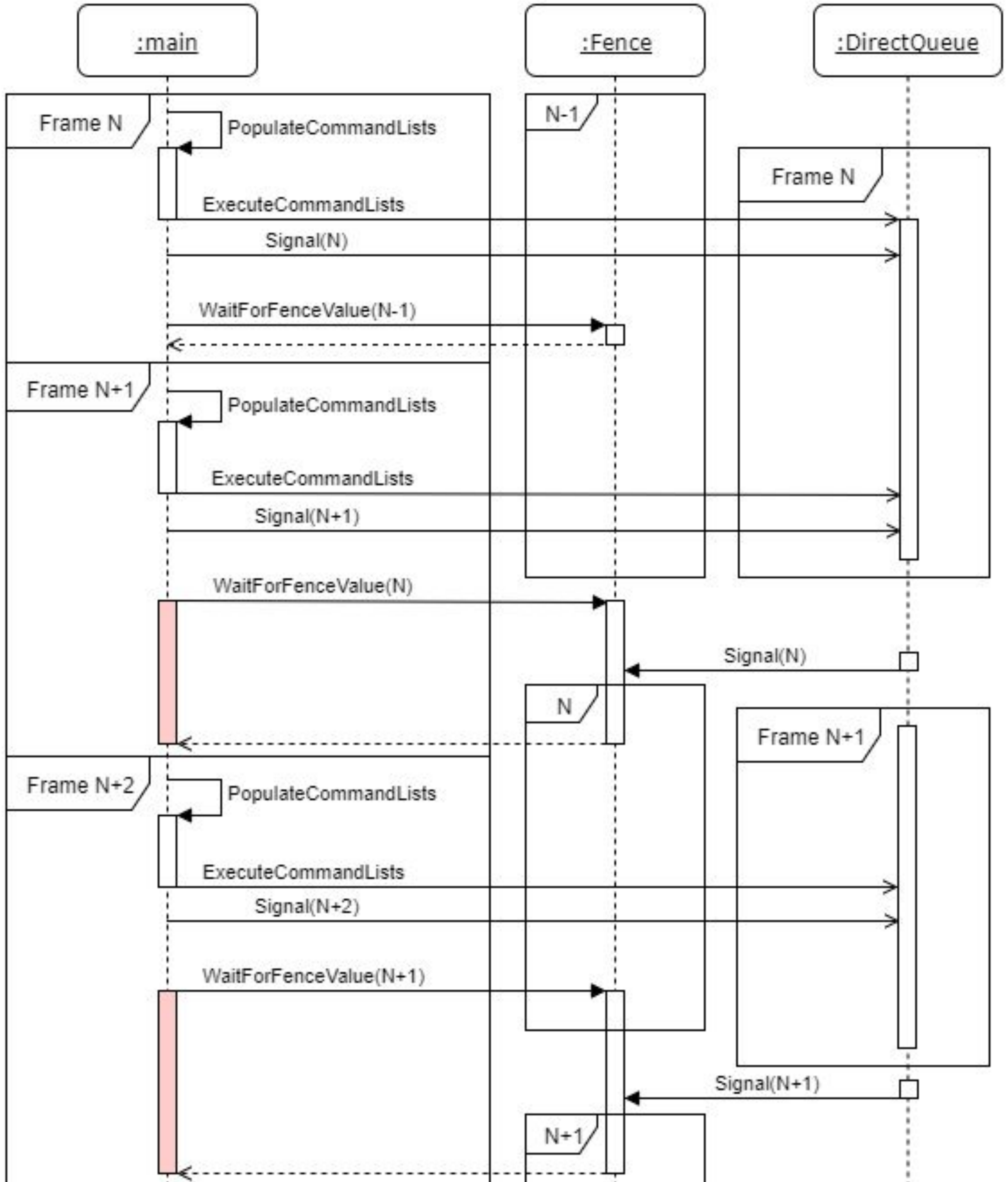
 5–10 execute command list

Avoid short CLs

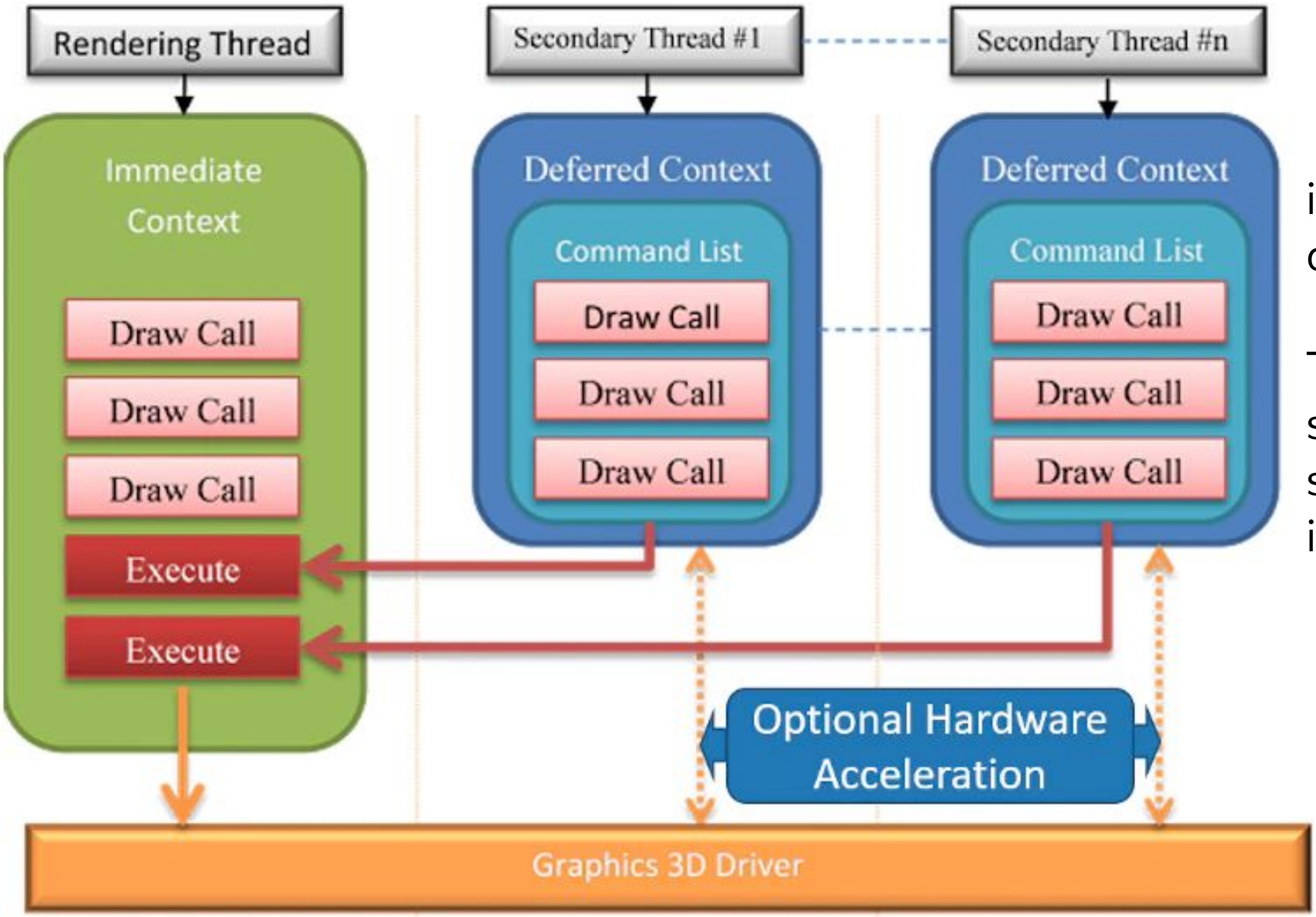
Command Queue:

Use copy queue for asyc transfer operations.

Use compute queue with care.



Multi threaded rendering in DX 11

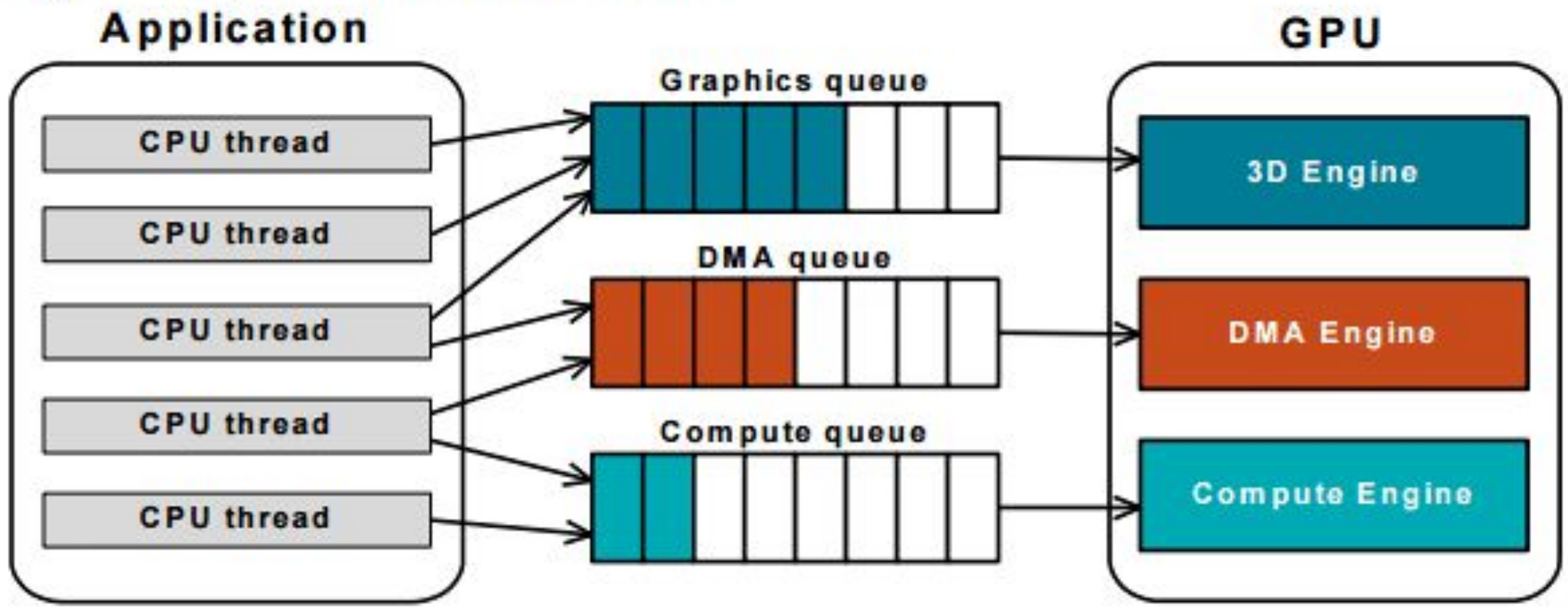


immediate context and deferred context

This multithreading strategy allows complex scenes to be broken up into concurrent tasks.

Multi threaded rendering in DX 12

Figure 2. Queue submission model

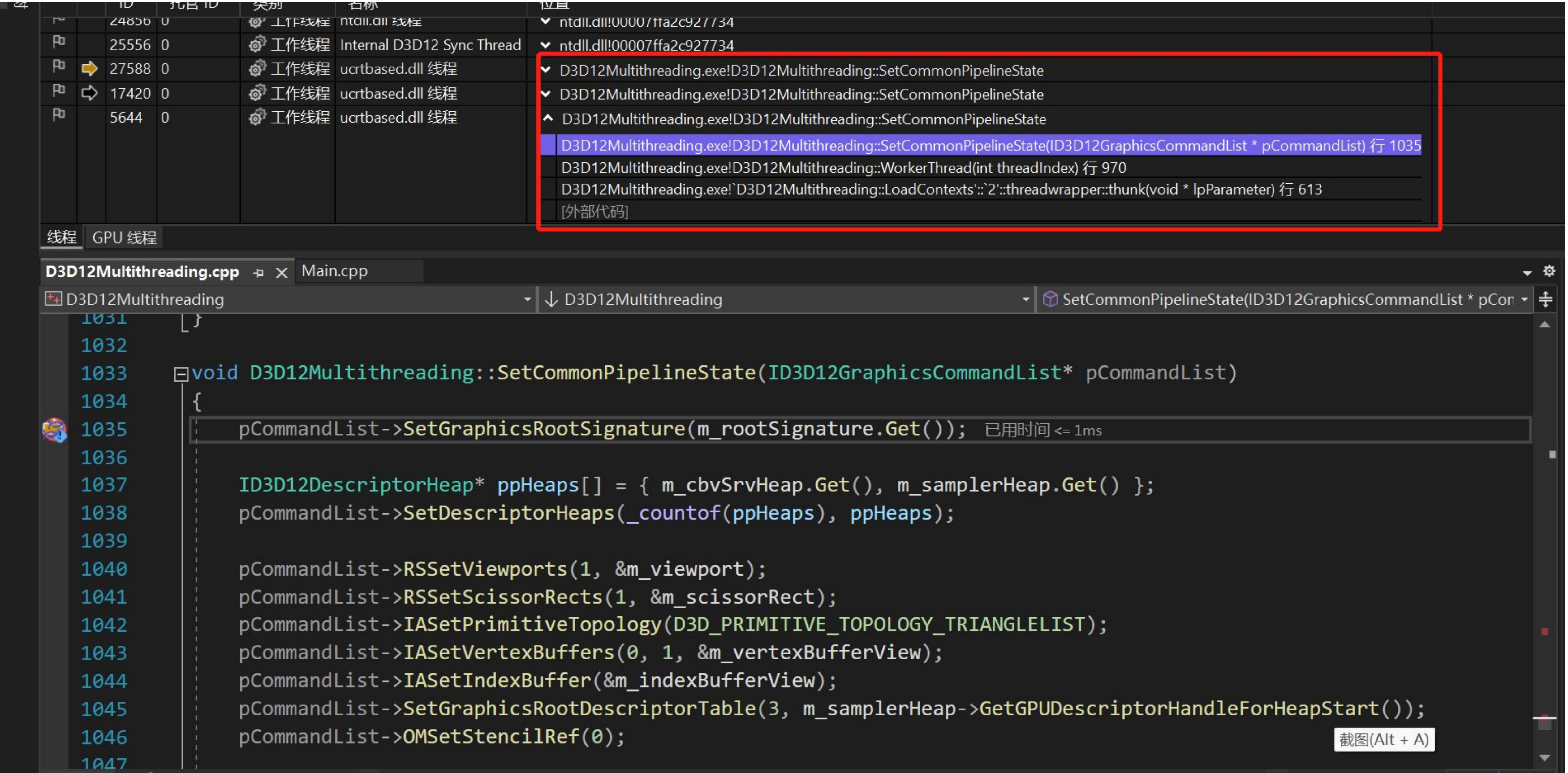


Dividing the workload into smaller commands requiring different resources. Allowing simultaneous execution.

This is how Asynchronous compute works by dividing the compute and graphics commands into separate queues and executing them concurrently.

Multi threaded rendering in DX 12

Multi threaded rendering in DX 12



```
24856  0   工作线程  ntdll.dll 线程                   ntdll.dll!00007ffa2c927734
25556  0   工作线程  Internal D3D12 Sync Thread       ntdll.dll!00007ffa2c927734
27588  0   工作线程  ucrtbased.dll 线程               D3D12Multithreading.exe!D3D12Multithreading::SetCommonPipelineState
17420  0   工作线程  ucrtbased.dll 线程               D3D12Multithreading.exe!D3D12Multithreading::SetCommonPipelineState
5644   0   工作线程  ucrtbased.dll 线程               D3D12Multithreading.exe!D3D12Multithreading::SetCommonPipelineState
                                                      D3D12Multithreading.exe!D3D12Multithreading::SetCommonPipelineState(ID3D12GraphicsCommandList * pCommandList) 行 1035
                                                      D3D12Multithreading.exe!D3D12Multithreading::WorkerThread(int threadIndex) 行 970
                                                      D3D12Multithreading.exe!`D3D12Multithreading::LoadContexts'::`2'::threadwrapper::thunk(void * lpParameter) 行 613
                                                      [外部代码]
```

```
    }

void D3D12Multithreading::SetCommonPipelineState(ID3D12GraphicsCommandList* pCommandList)
{
    pCommandList->SetGraphicsRootSignature(m_rootSignature.Get());  已用时间 <= 1ms

    ID3D12DescriptorHeap* ppHeaps[] = { m_cbvSrvHeap.Get(), m_samplerHeap.Get() };
    pCommandList->SetDescriptorHeaps(_countof(ppHeaps), ppHeaps);

    pCommandList->RSSetViewports(1, &m_viewport);
    pCommandList->RSSetScissorRects(1, &m_scissorRect);
    pCommandList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
    pCommandList->IASetVertexBuffers(0, 1, &m_vertexBufferView);
    pCommandList->IASetIndexBuffer(&m_indexBufferView);
    pCommandList->SetGraphicsRootDescriptorTable(3, m_samplerHeap->GetGPUDescriptorHandleForHeapStart());
    pCommandList->OMSetStencilRef(0);
```

Multi threaded rendering in DX 12

Modern APIs support multi-thread encoding mainly to:

- **Reduce API calling drawback**
- **Scaling with increase number of CPU cores**

In order to ensure all pending work:

1. You can insert a fence, submit the command-list, and wait for the GPU to stall.

2. You can insert a resource barrier into the command-list which indicates the point at which one state moves to the next.

Resource Barrier

Resource Barrier notifies the graphics driver of situations in which the driver may need to synchronize multiple accesses to the memory in which a resource is stored

In Dx 11, drivers were required to track this state in the background.
Expensive;
Complicate multi-threaded design;

Transition barrier – A transition barrier indicates that a set of subresources transition between different usages.

Aliasing barrier – An aliasing barrier indicates a transition between usages of two different resources which have overlapping mappings into the same heap.

Unordered access view (UAV) barrier – A UAV barrier indicates that all UAV accesses, both read or write, to a particular resource must complete between any future UAV accesses, both read or write.

Synchronization

Modern APIs provide synchronization primitives

- **Inter-command-buffer** synchronization:
  semaphore(GPUtoGPU), fence(GPUto CPU)

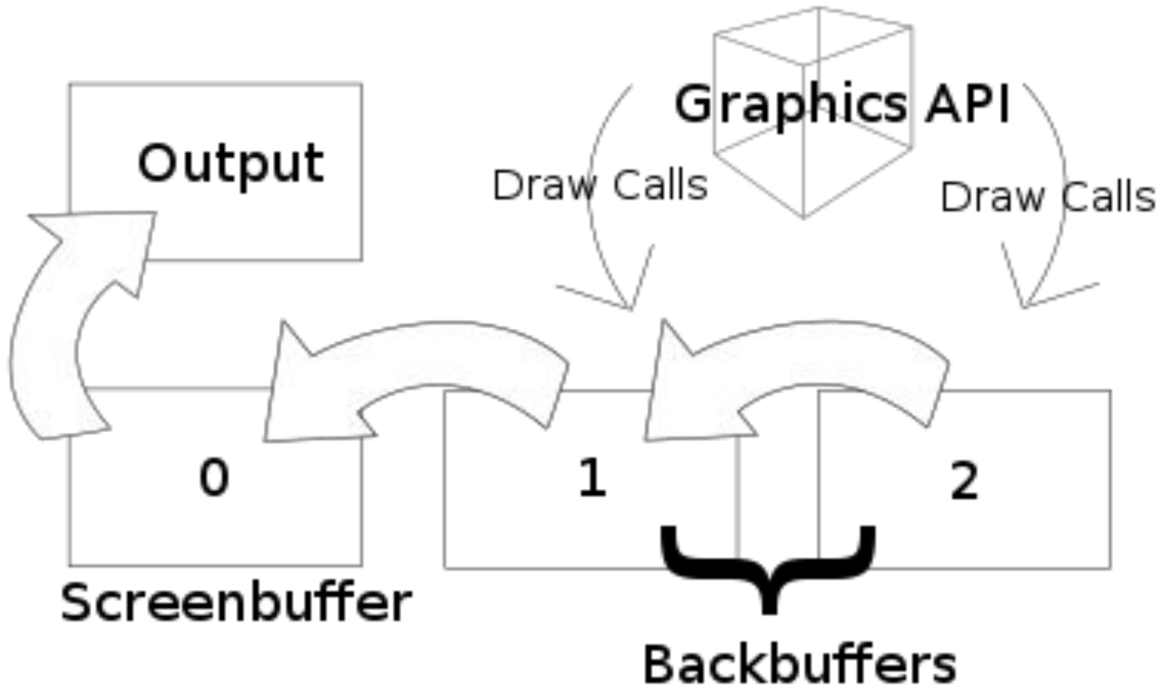
- **Intra-command-buffer** synchronization: barriers

Barrier: GPU exe is heavily overlapped

- Use drawcalls to build a exe dependency
- Use barrier to do layout transition(flush, invalidate caches; decompress and resolve)

5 Describe and Create the Swap Chain创建交换链

Filling out the instance of DXGI_SWAP_CHAIN_DESC
The create func can be called multi times, It will destroy the old swap chain before creating the new one.



6 Create the Descriptor Heaps创建解释器堆

We need descriptor heaps(*ID3D12DescriptorHeap*) to store descriptor views our app needs.

It is ceated with *ID3D12Device::CreateDescriptorHeap* We need *SwapChainBufferCount* many RTV to describe buffer resources in swap chain we will render into. And one DSV to describe the depth/stensil buffer resource for depth testing. We need a heap for storing *SwapChainBufferCount* RTVs, and a heap for DSV.

Our app reference descriptors through handles. The handle to the first descriptor in a heap is obtained with *ID3D12DescriptorHeap::GetCPUDescriptorHandleForHeapStart* method.

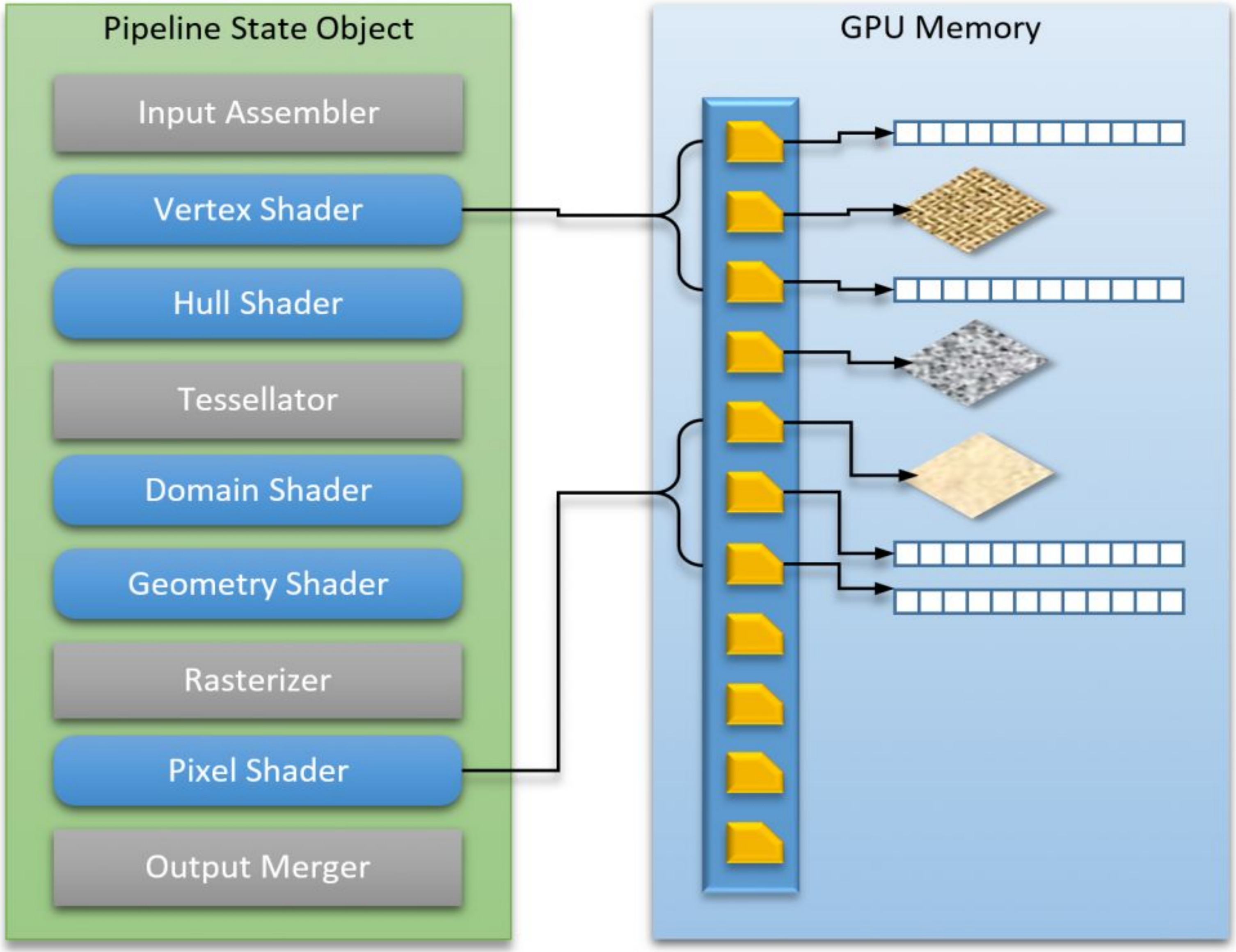
Descriptor & Descriptor Heap

Descriptor:
A descriptor is a small block of data that fully describes an object to the GPU, in a GPU specific opaque format.

Descriptor heap is essentially an array of descriptors.

Resource binding is a two-stage process:

1. Shader register is first mapped to the descriptor in a descriptor heap as defined by the root signature.
2. The descriptor (which may be SRV, UAV, CBV or Sampler) then references the resource in GPU memory.

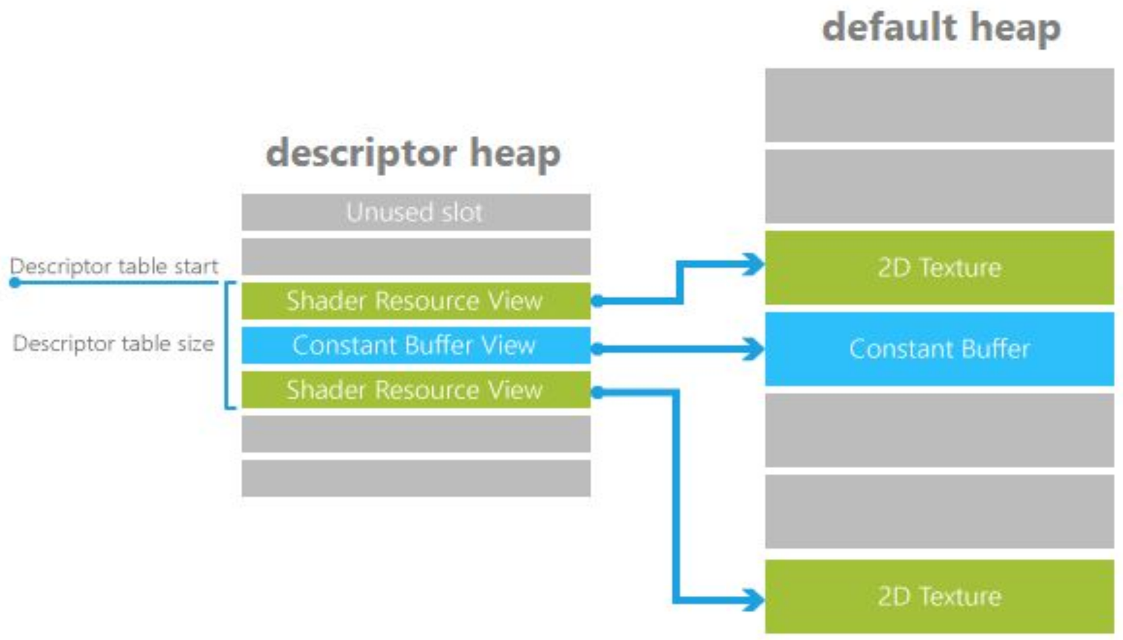


Descriptor & Descriptor Heap

Descriptor types

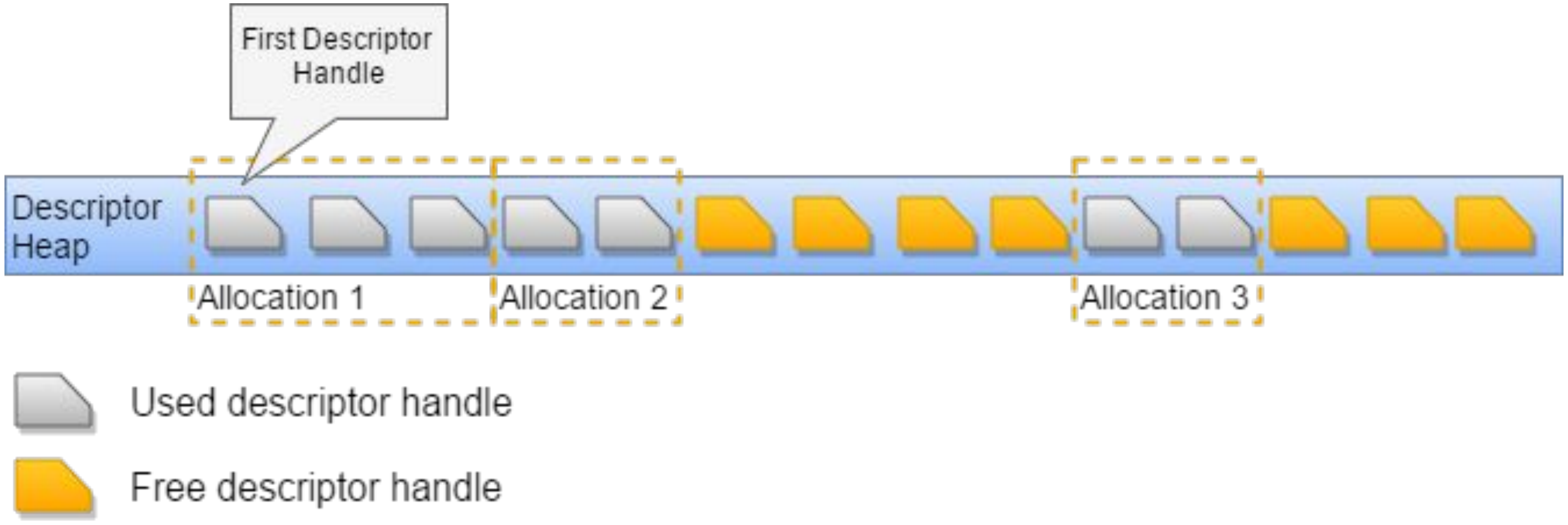
Shader visible:
1. CBV/SRV/UAV descriptors describe constant buffers, shader resources and unordered access view resources.
2. Sampler descriptors describe sampler resources (used in texturing).

CPU visible:
3. RTV descriptors describe render target resources.
4. DSV descriptors describe depth/stencil resources

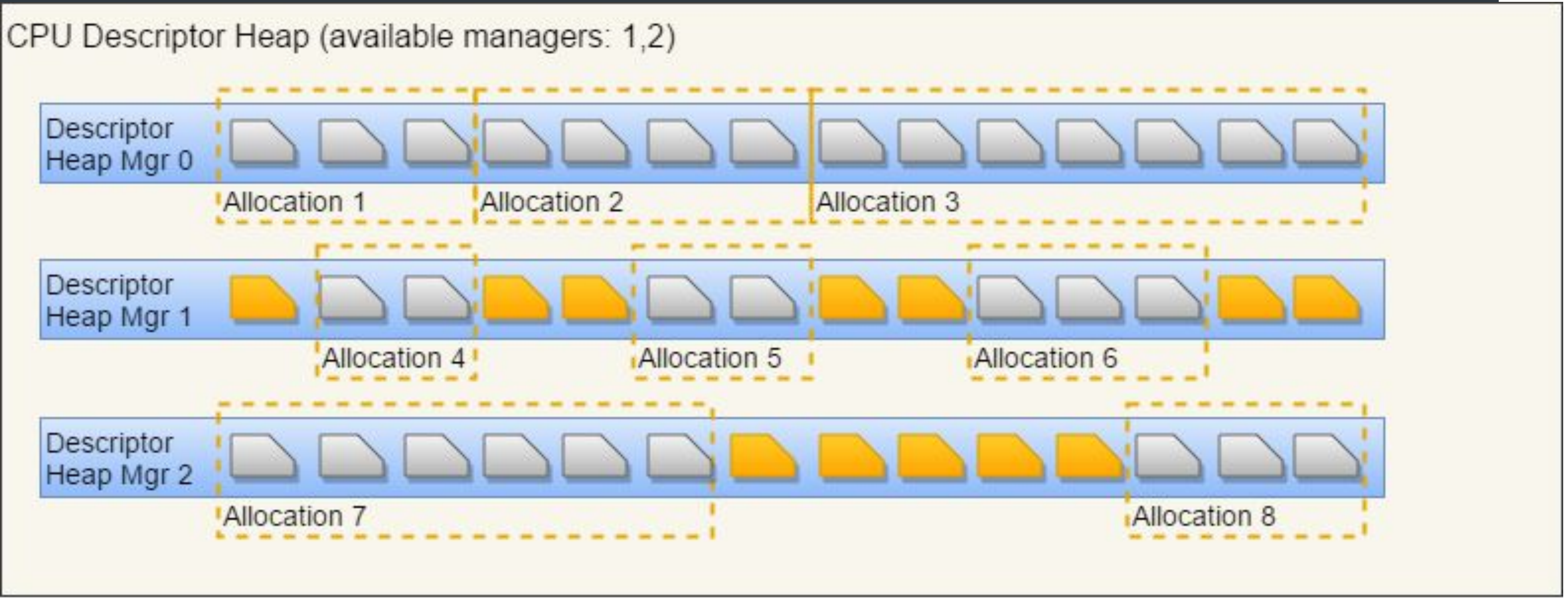


Descriptor Heap Allocation Manager

Uses variable-size GPU allocations manager to handle allocations within the descriptor heap.



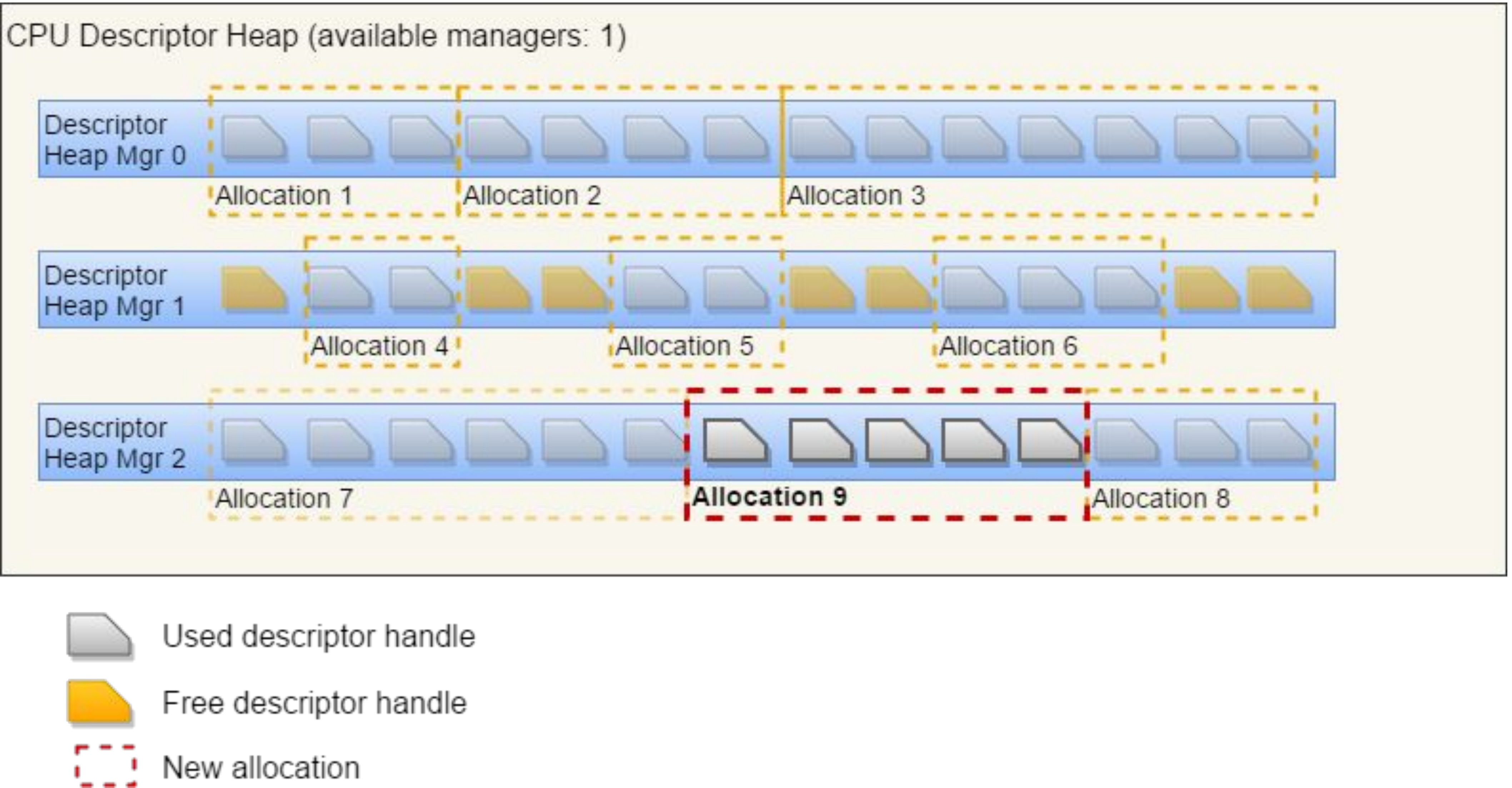
```
// Pool of descriptor heap managers
std::vector<DescriptorHeapAllocationManager> m_HeapPool;
// Indices of available descriptor heap managers
std::set<size_t> m_AvailableHeaps;
```



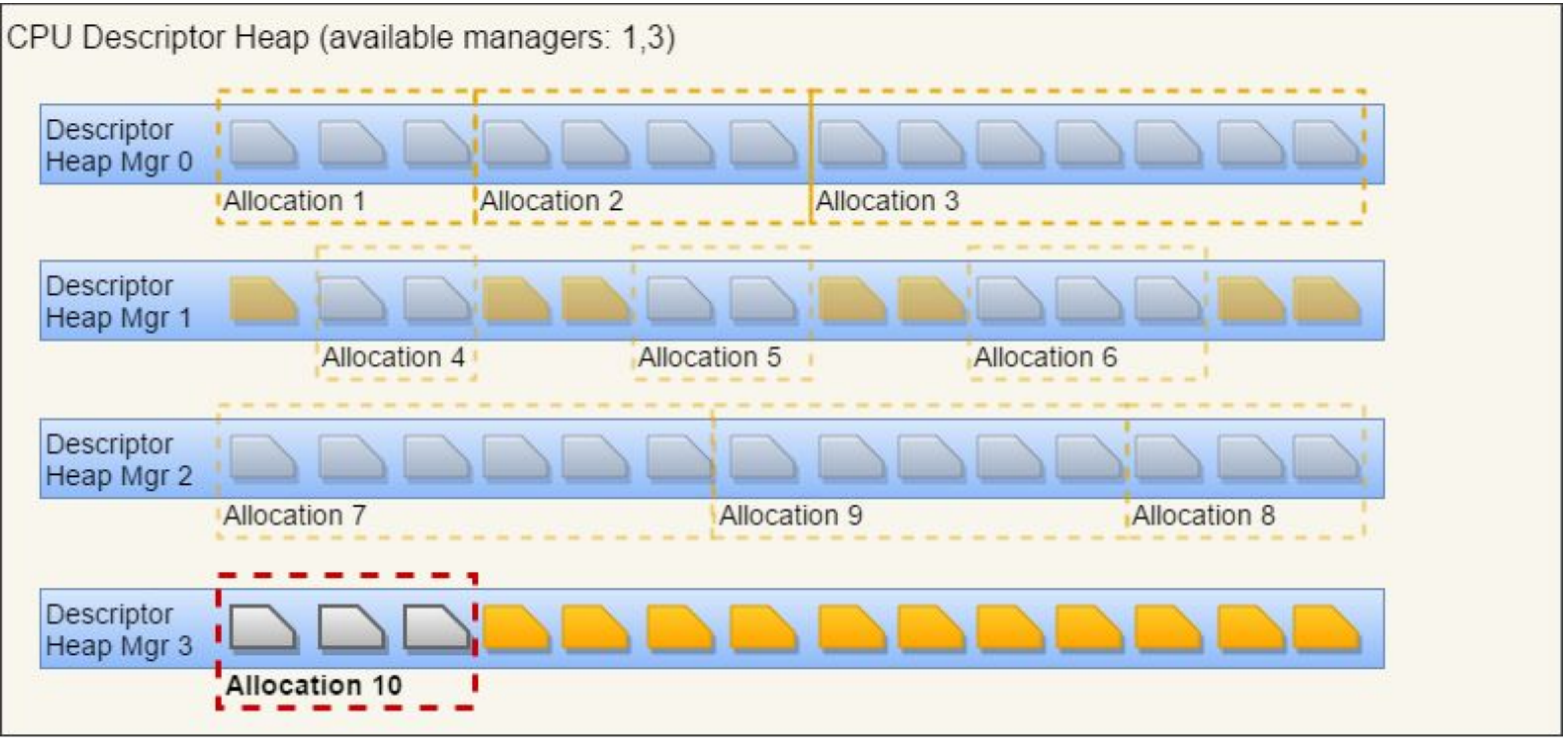
Used descriptor handle
Free descriptor handle

CPU Descriptor Heap

The main goal of the CPU descriptor heap is to provide storage for the resource view descriptors.



CPU Descriptor Heap



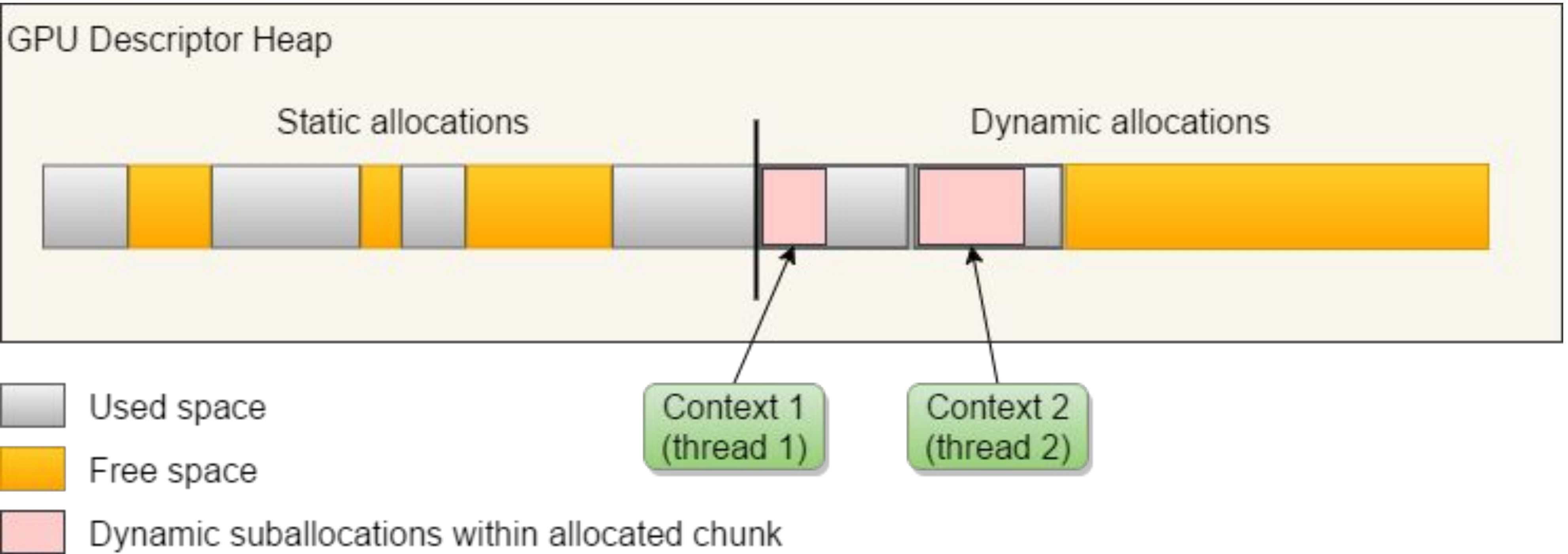
▭	Used descriptor handle
🟧	Free descriptor handle
⬚	New allocation

GPU Descriptor Heap

For GPU to be able to access the descriptors, they must reside in a shade-visible descriptor heap.
Only one SRV_CBV_UAV and one SAMPLER heap can be bound to the GPU at the same time.

the first part is intended to keep rarely changing descriptor handles

The second part is used to hold dynamic descriptor handles, i.e. temporary handles that live during the current frame only



GPU Descriptor Heap

Bottleneck: Dynamic descriptor allocation is a very frequent operation, and if several threads record commands simultaneously, allocating dynamic descriptor handles would cause bottleneck.

On the first stage, every command context recording commands allocates a chunk of descriptors from the shared dynamic part of the GPU descriptor heap. This operation requires exclusive access to the GPU heap, but happens infrequently.

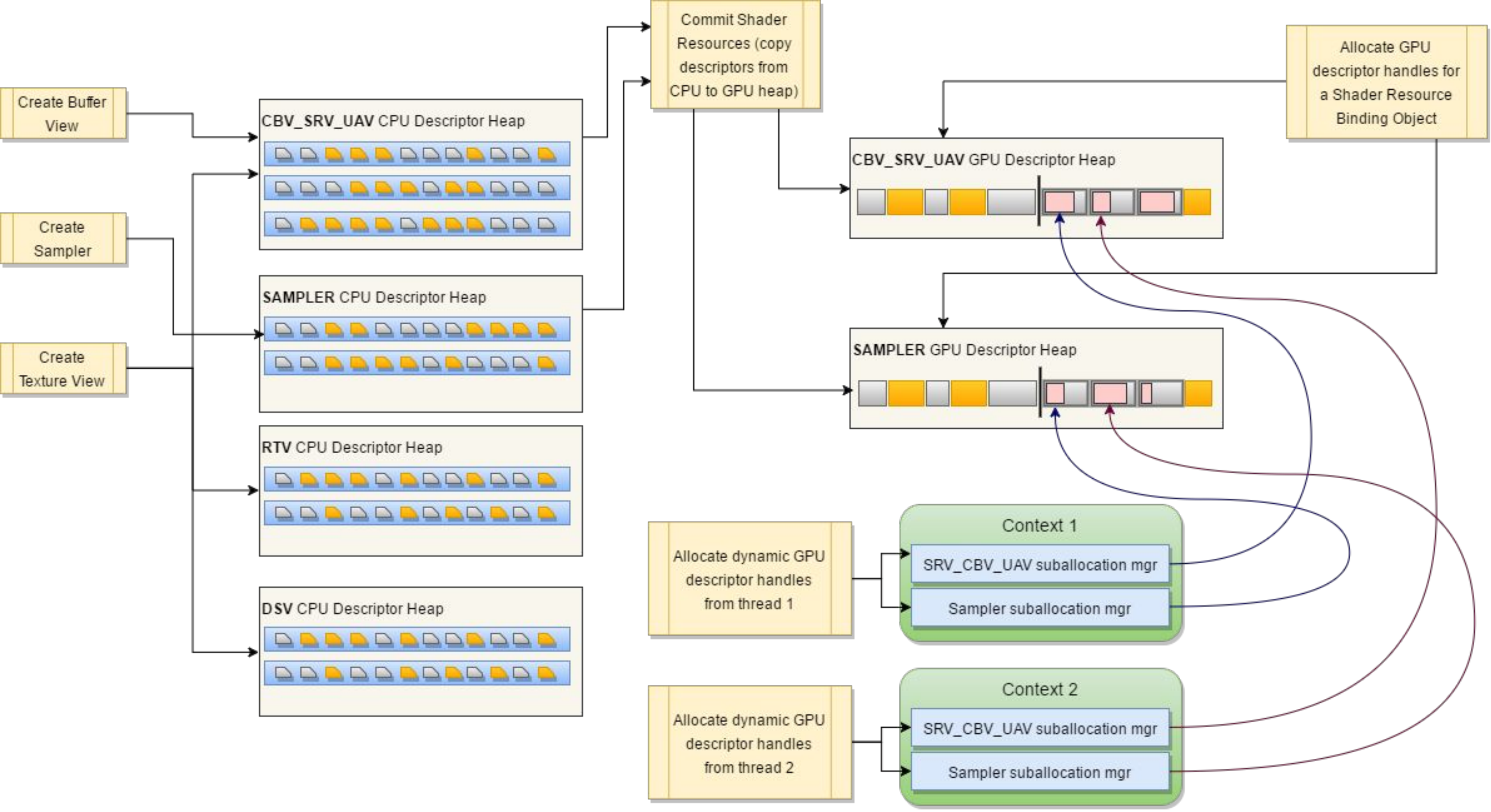
```
// Check if there are no chunks or the last chunk does not have enough space
if( m_Suballocations.empty() ||
    m_CurrentSuballocationOffset + Count > m_Suballocations.back().GetNumHandles() )
{
    // Request new chunk from the GPU descriptor heap
    auto SuballocationSize = std::max(m_DynamicChunkSize, Count);
    auto NewDynamicSubAllocation = m_ParentGPUHeap.AllocateDynamic(SuballocationSize);
    m_Suballocations.emplace_back(std::move(NewDynamicSubAllocation));
    m_CurrentSuballocationOffset = 0;
}
```

GPU Descriptor Heap

The second stage is suballoction from that chunk. This part is lock-free and can be done in parallel by every thread.

```
// Perform suballocation from the last chunk
auto &CurrentSuballocation = m_Suballocations.back();

auto ManagerId = CurrentSuballocation.GetAllocationManagerId();
DescriptorHeapAllocation Allocation( this,
                                     CurrentSuballocation.GetDescriptorHeap(),
                                     CurrentSuballocation.GetCpuHandle(m_CurrentSuballocationOffset),
                                     CurrentSuballocation.GetGpuHandle(m_CurrentSuballocationOffset),
                                     Count,
                                     static_cast<Uint16>(ManagerId) );
m_CurrentSuballocationOffset += Count;
```

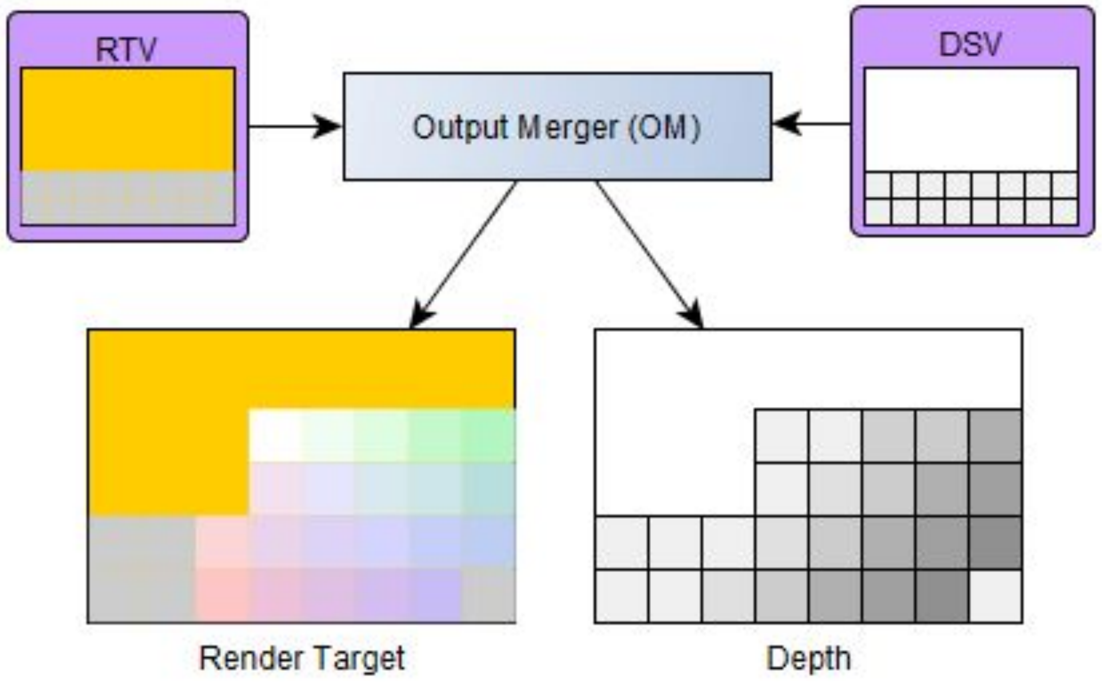

Create Buffer View
Create Sampler
Create Texture View
CBV_SRV_UAV CPU Descriptor Heap
SAMPLER CPU Descriptor Heap
RTV CPU Descriptor Heap
DSV CPU Descriptor Heap
Commit Shader Resources (copy descriptors from CPU to GPU heap)
Allocate GPU descriptor handles for a Shader Resource Binding Object
CBV_SRV_UAV GPU Descriptor Heap
SAMPLER GPU Descriptor Heap
Allocate dynamic GPU descriptor handles from thread 1
Context 1
SRV_CBV_UAV suballocation mgr
Sampler suballocation mgr
Allocate dynamic GPU descriptor handles from thread 2
Context 2
SRV_CBV_UAV suballocation mgr
Sampler suballocation mgr

7 Create render target view创建渲染目标视图

We must create a resource view(descriptor) to the resource and bind the view to pipeline stage.

Bind back buffer to the output merge stage of pipeline, need to create a render target view.

*HRESULT IDXGISwapChain::GetBuffer(UINT Buffer, REFIID riid, void **ppSurface);*



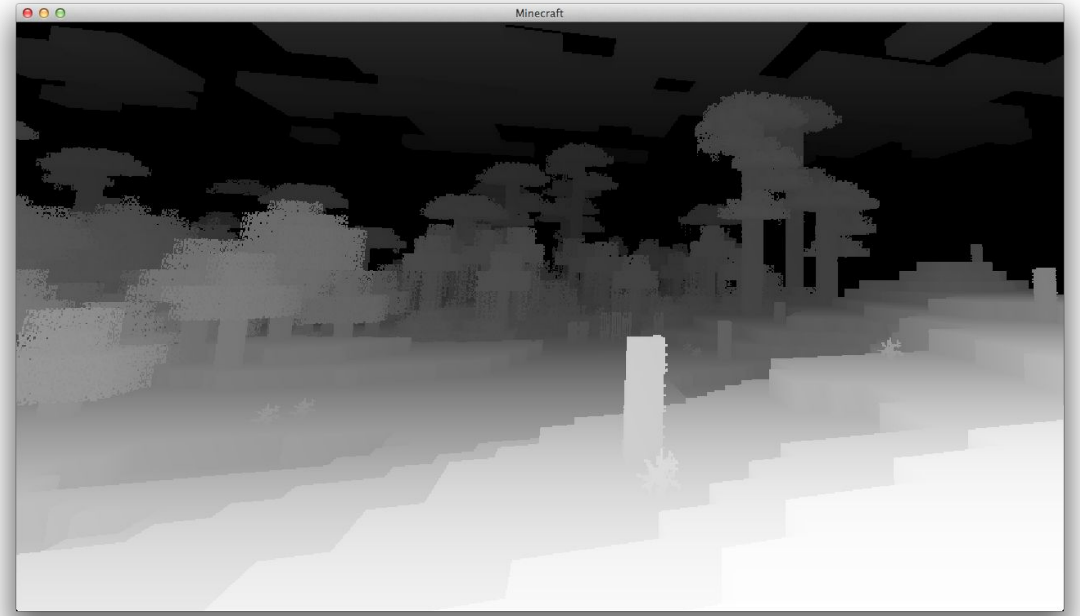
8 Create the Depth/Stencil buffer and view创建深度、模板缓冲

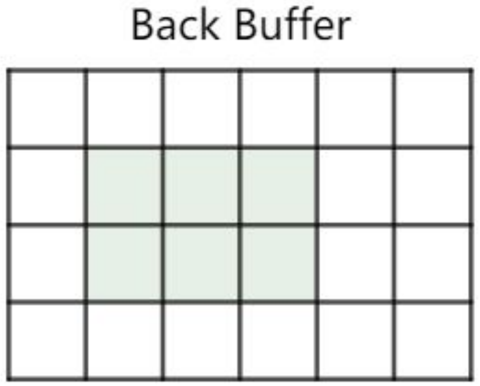
Depth buffer: a 2D texture stores the depth info of the nearest visible objects.

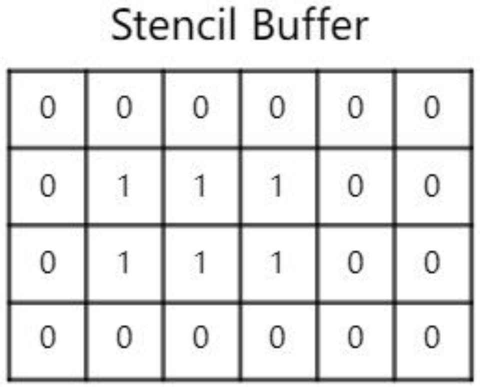
stencil buffer: is an extra data buffer.

Texture: a GPU resource.

GPU resources live in heaps, which are essentially blocks of GPU memory with certain properties. The method *ID3D12Device::CreateCommittedResource* creates and commits a resource to a particular heap with properties we specify.







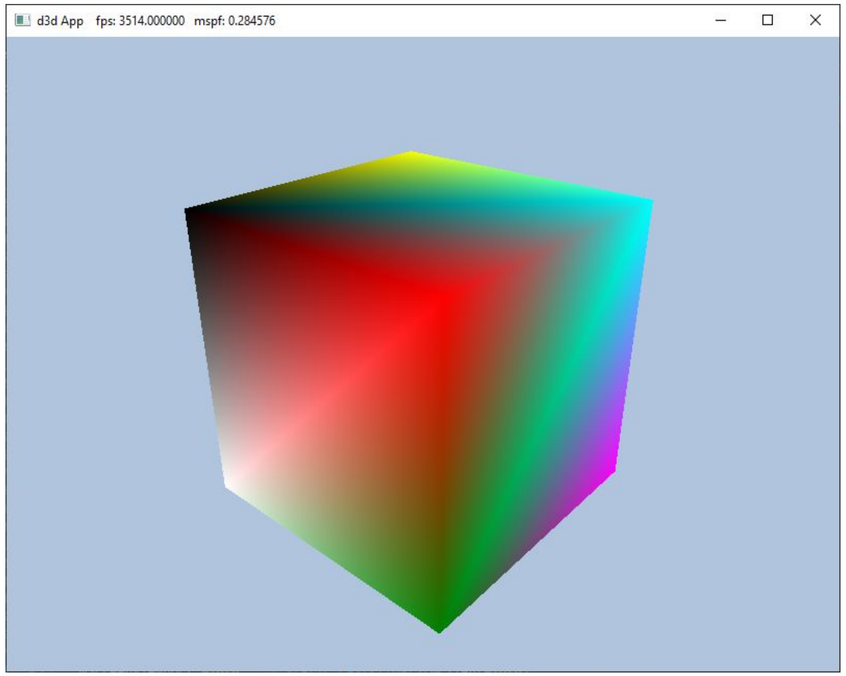
0	0	0	0	0	0
0	1	1	1	0	0
0	1	1	1	0	0
0	0	0	0	0	0

9 Set the viewport设置视窗

Usually we like to draw the 3D scene to the entire back buffer. It's size corresponds to the entire screen.

Viewport: the subrectangle of back buffer we draw into.
In D3D depth values are stored in the depth buffer in a normalized range of 0 to 1. We set the viewport by RSSetViewports.

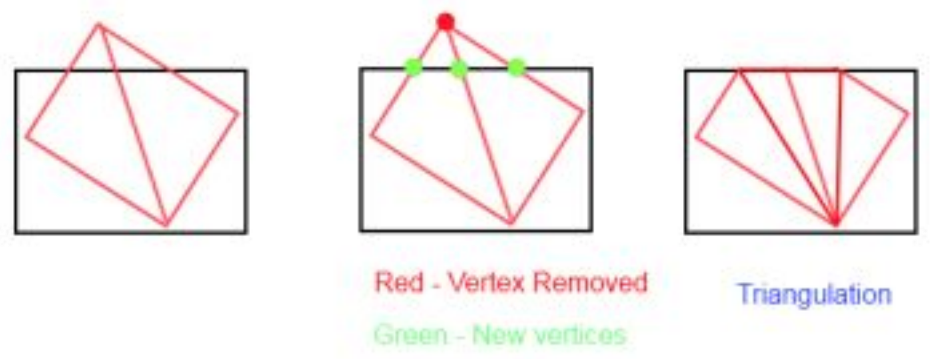
Cannot specify multiple viewports to the same render target.
Viewport needs to be reset whenever command list is reset.



10 Set the Scissor Rectangles设置剪裁长方形

We define a **scissor rectangle** relative to the back buffer such that pixels outside this rectangle are culled. We set the rectangle by *RSSetScissorRects*.

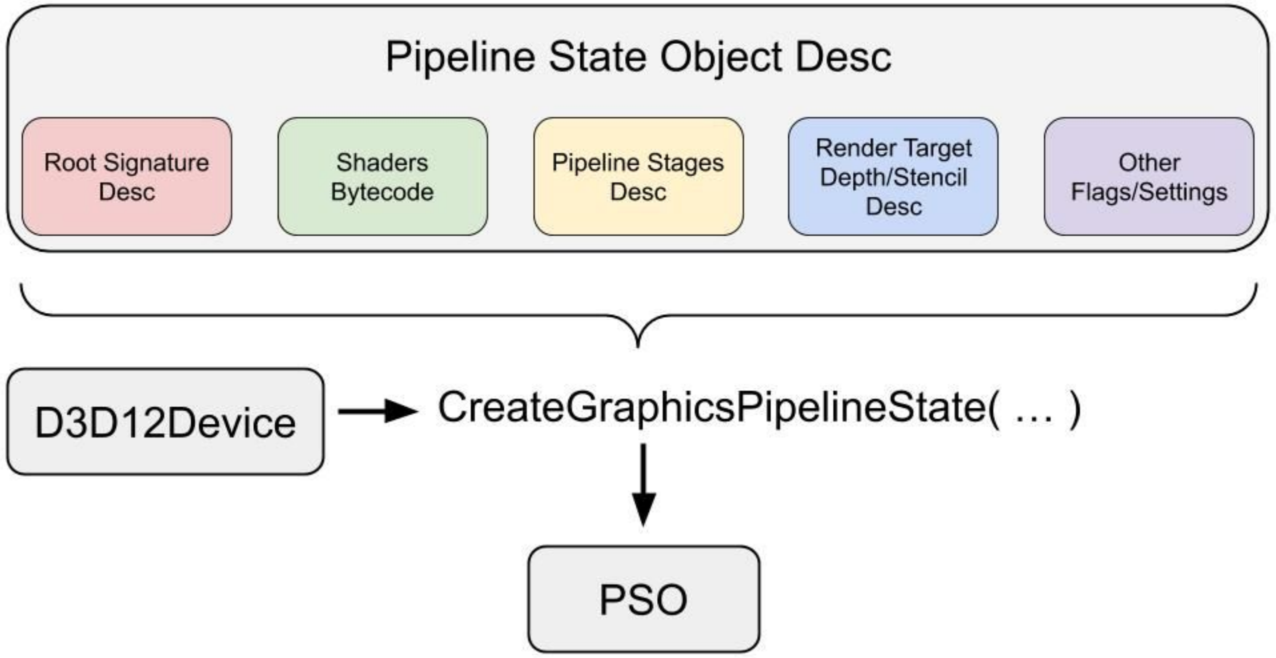
Cannot specify multiple scissor rectangles on the same render target.



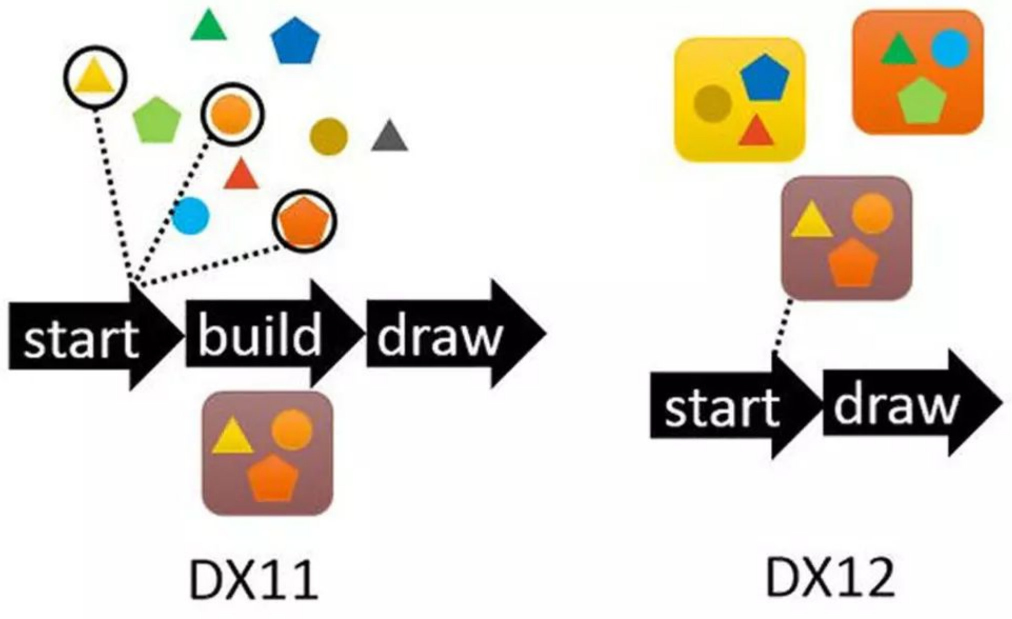
Pipeline State Object

The PSO is an object that represents the settings for our graphics device (GPU) in order to draw or dispatch something.

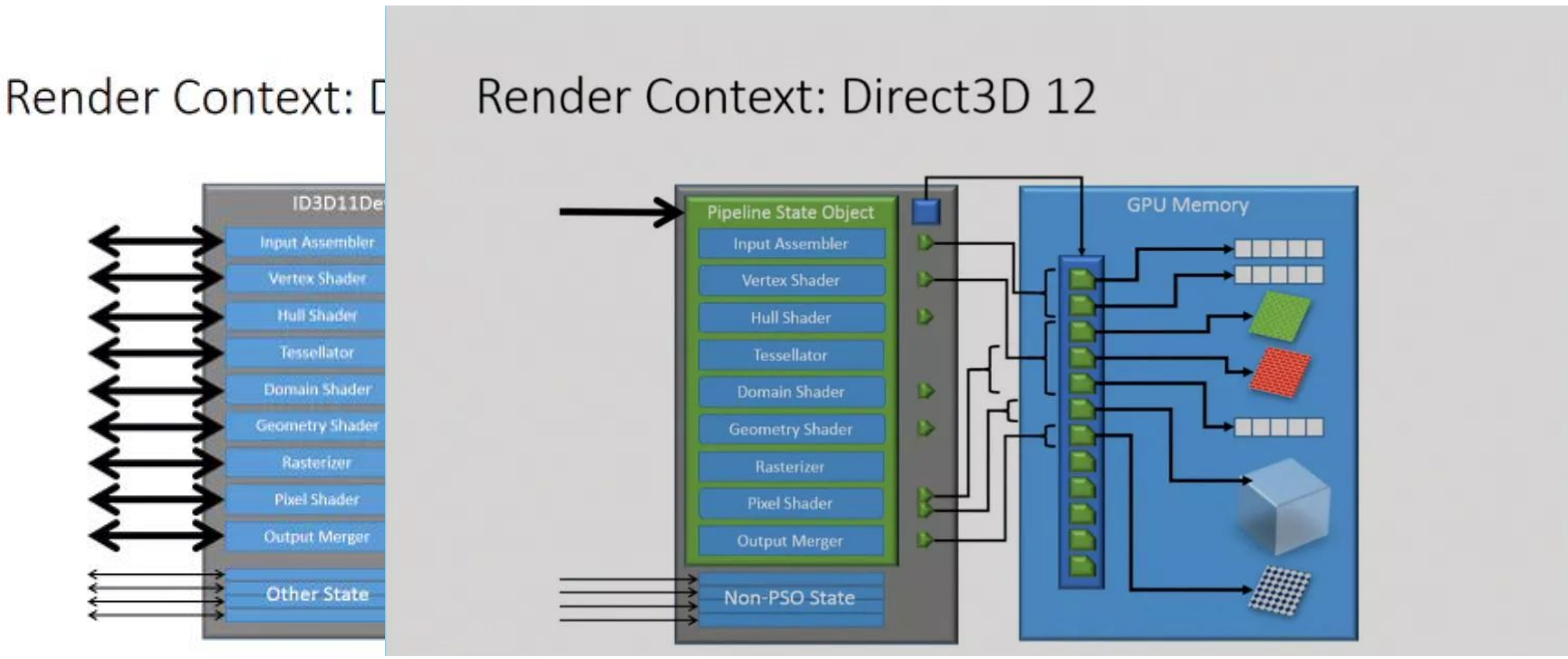
In DirectX 11, the objects in the GPU pipeline exist across a wide range of states such as Vertex Shader, Hull Shader, Geometry shader, etc.



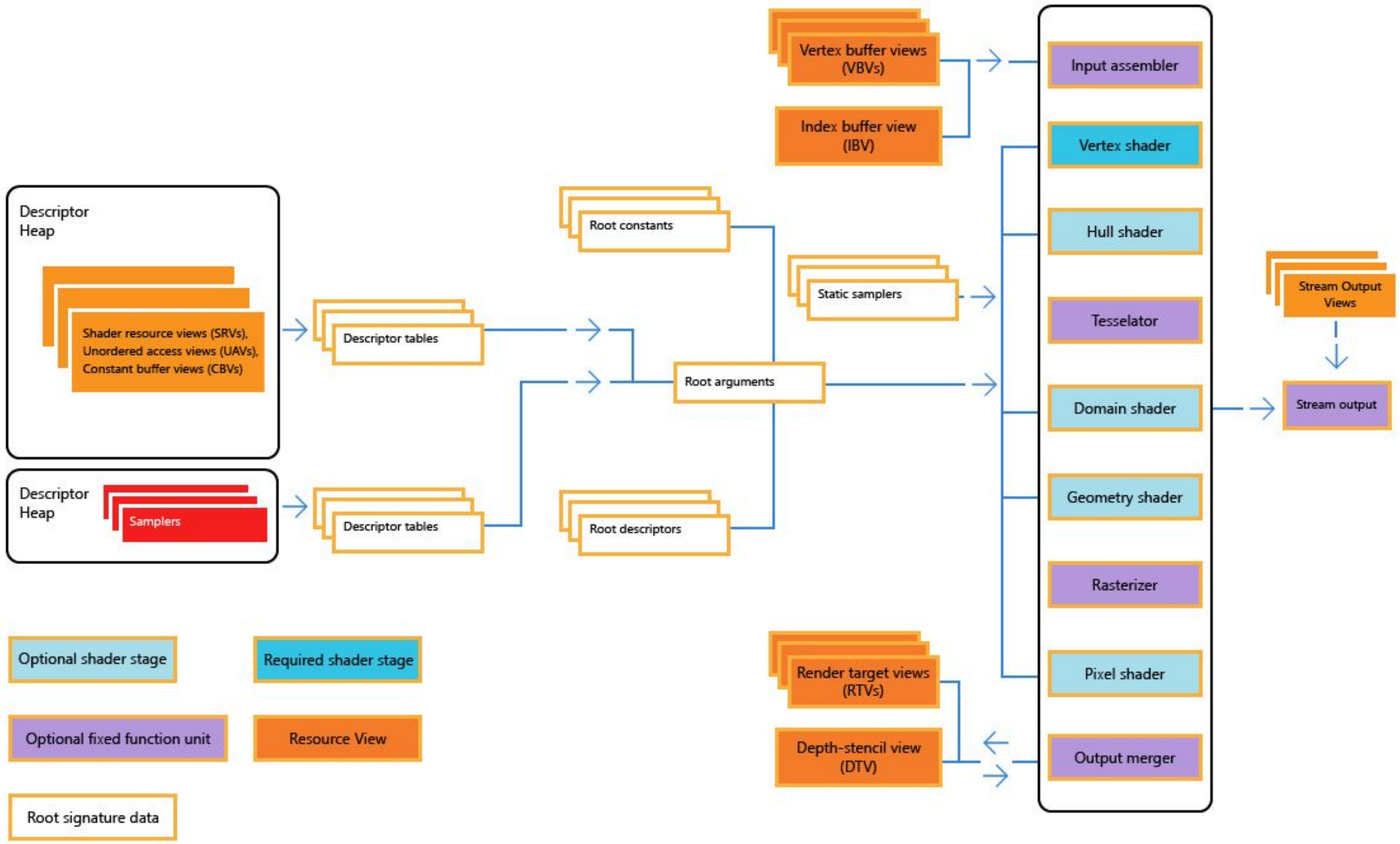
Pipeline State Object Desc
Root Signature Desc
Shaders Bytecode
Pipeline Stages Desc
Render Target Depth/Stencil Desc
Other Flags/Settings
D3D12Device
CreateGraphicsPipelineState(…)
PSO



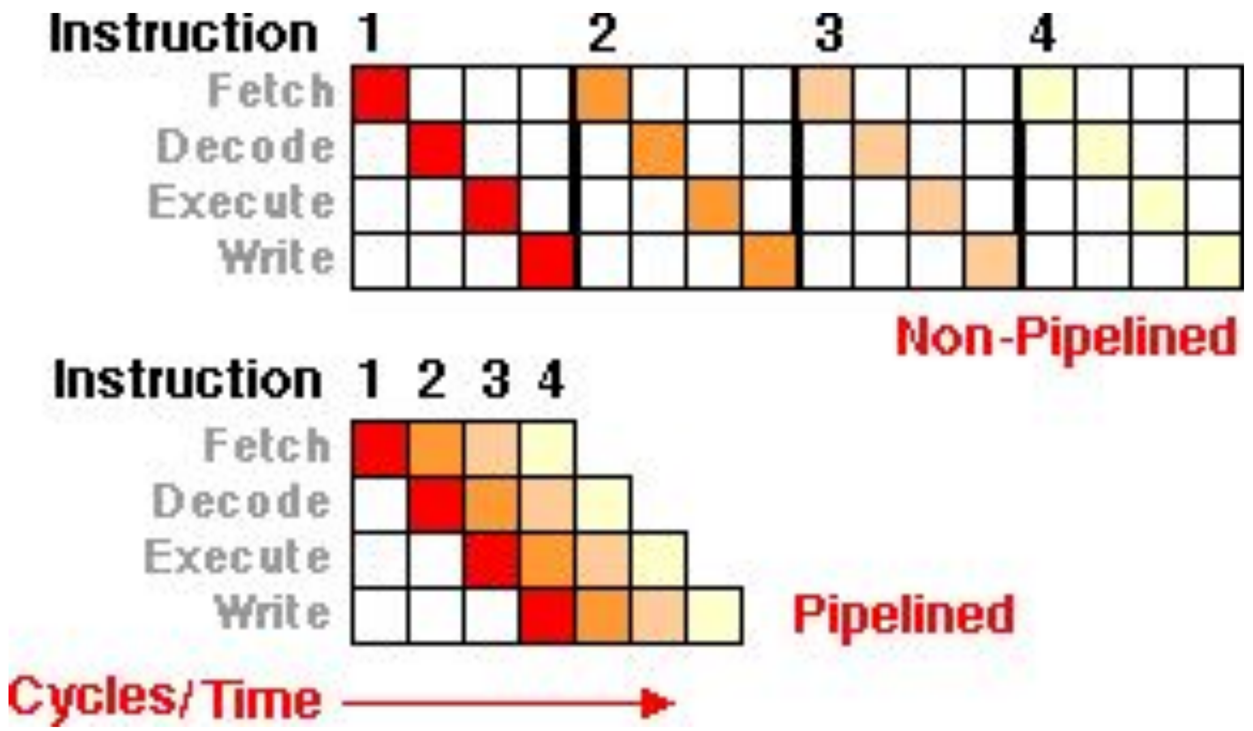
Pipeline State Object



Graphic Pipeline: Dx12



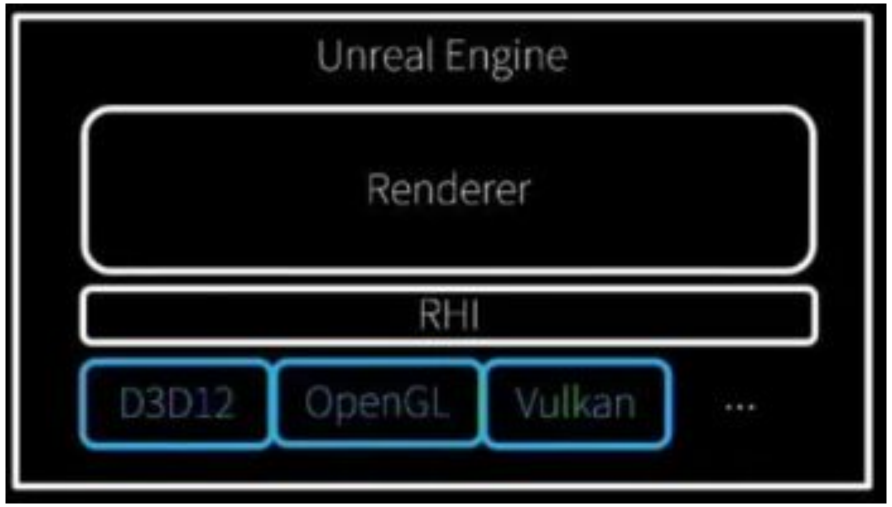
Pipeline State Object Caching



Pipeline :

1. Optimized hardware unit allocation for each stage.
2. It can be nested because it is divided into stages: maximum efficiency.

Pipeline State Object Caching

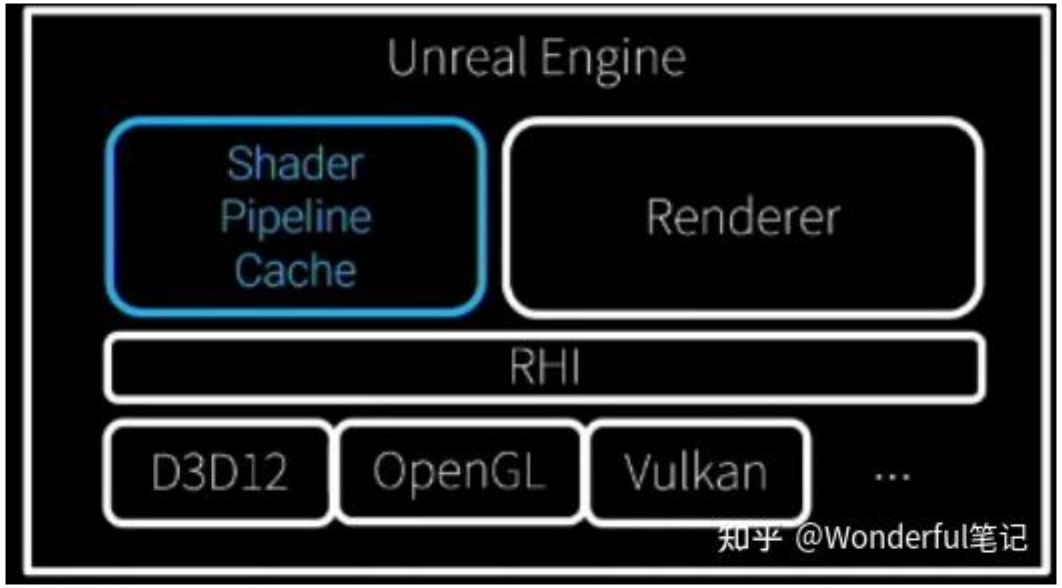


RHI
A thin layer on the platform-specific graphics API. Platform-independent code that handles all operations.

Low level cache:

D3D12 / Vulkan / Metal:
Cache support for <u>runtime generated PSO</u>.
D3D12 / Vulkan:
Create a <u>load-time PSO</u> by file out the PSO to disk.
OpenGL:
ProgramBinary Create a load-time PSO by file out the PSO to dis

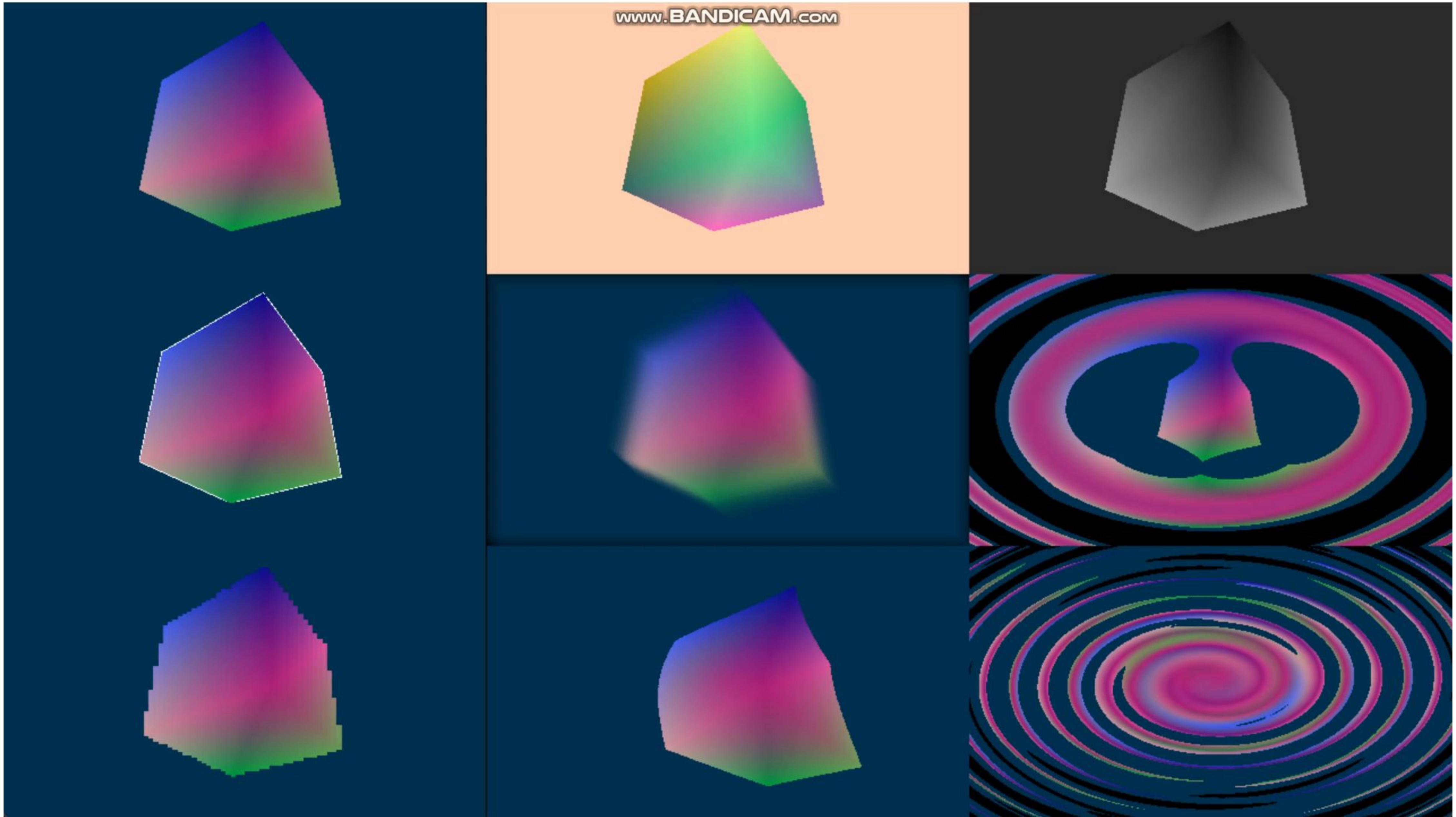
Pipeline State Object Caching



Shader Pipe-line cache.

1. An object that utilizes RHI level API to help low level digging.
2. Focus on when to generate/create PSO.
3. Replace Shader Cache that existed in the past.

Pipeline State Object Caching



Summary: Dx11 vs Dx12

- **Explicit Synchronization**
- **Physical Memory Residency Manager**
- **Pipeline state objects**
- **Command lists and bundles**
- **Descriptor heaps and tables**

What's more:

- **DXR (DirectX Ray Tracing)**
- **VRS (Variable Rate Shading)**
- **Mesh Shading**
- **Sampler Feedback**

Reference

- https://alain.xyz/blog/comparison-of-modern-graphics-apis
- https://en.wikipedia.org/wiki/DirectX
- https://learn.microsoft.com/en-us/windows/win32/direct3d12/
- https://developer.nvidia.com/sites/default/files/akamai/gameworks/blog/GDC16/GDC16_gthomas_adunn_Practical_DX12.pdf
- http://diligentgraphics.com/diligent-engine/architecture/d3d12/managing-descriptor-heaps/