Approaching Minimum Overhead with Direct3D12

Jerry Cao - DevTech Engineer

jecao@nvidia.com

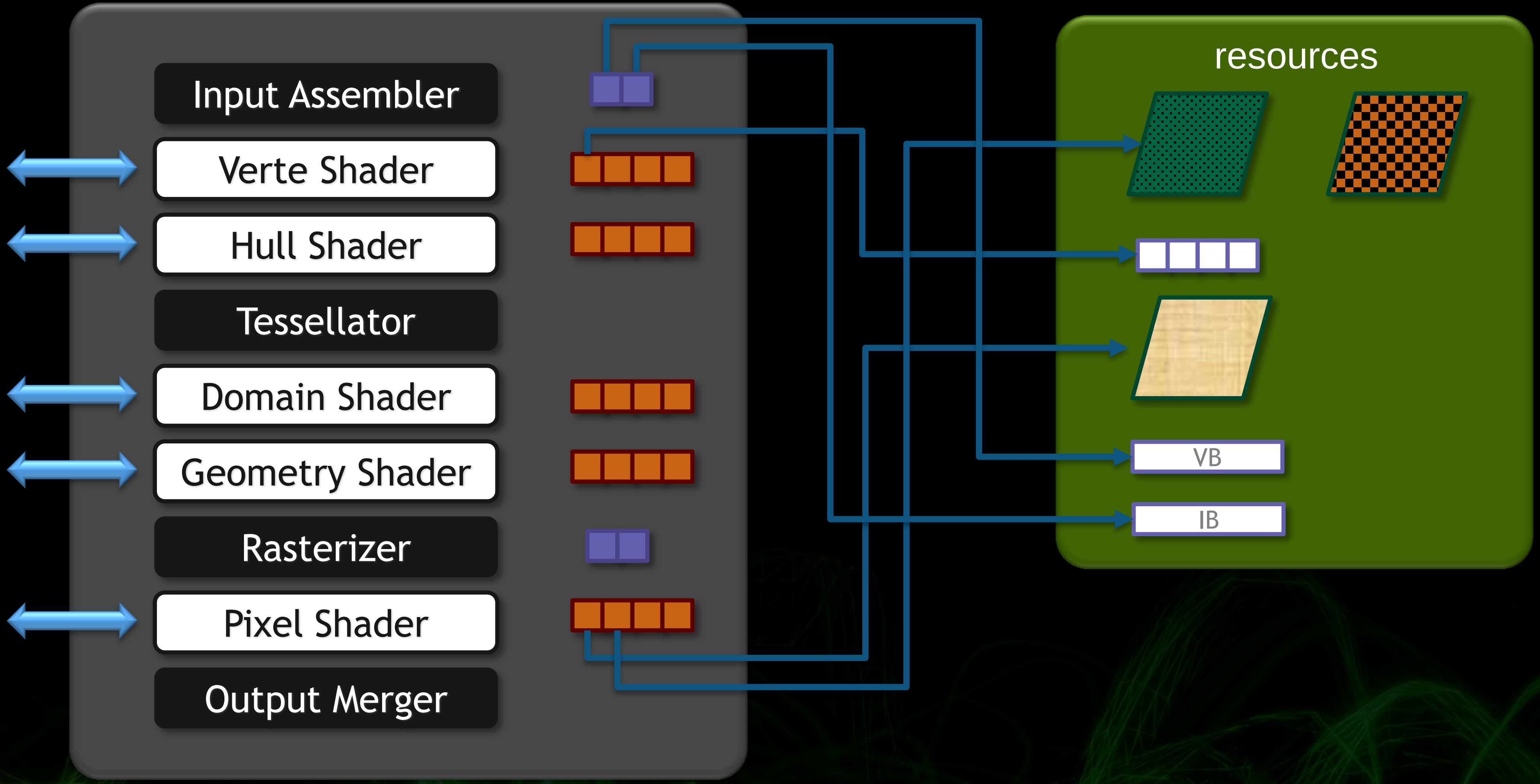
Overview

- D3D12 brief introduction
- Explicit memory management
- Reducing CPU overhead
  - CPU efficiency
  - CPU parallelism
- Improving GPU efficiency
- Performance Comparison with D3D11 and OpenGL 4.x
- New Graphics Features

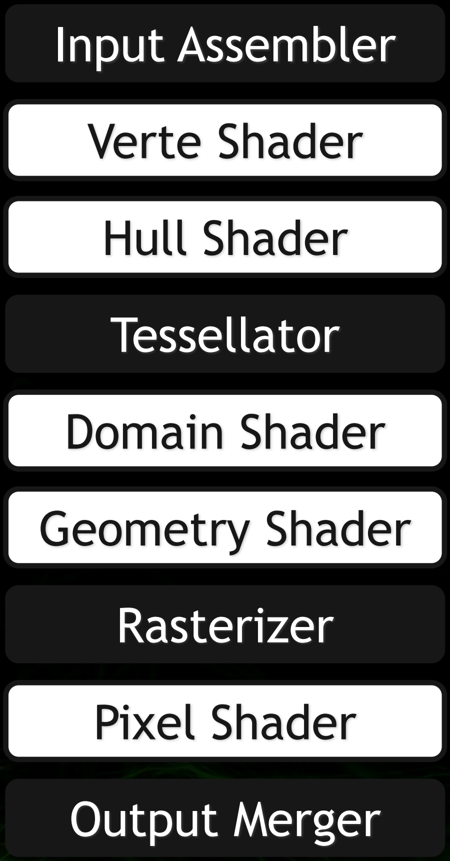
Direct3D12 Introduction

- Latest high-performance graphics API
- Low-level model, even more direct
- Works across all Microsoft Platforms

D3D11 Graphics Pipeline



Pipeline State Object



Input Assembler

Verte Shader

Hull Shader

Tessellator

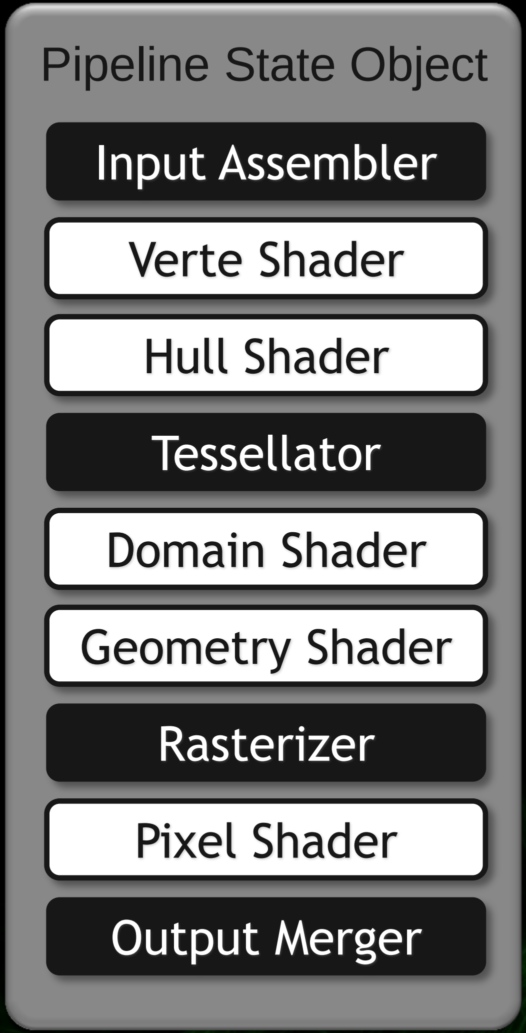
Domain Shader

Geometry Shader

Rasterizer

Pixel Shader

Output Merger



Pipeline State Object

Input Assembler

Verte Shader

Hull Shader

Tessellator

Domain Shader

Geometry Shader

Rasterizer

Pixel Shader

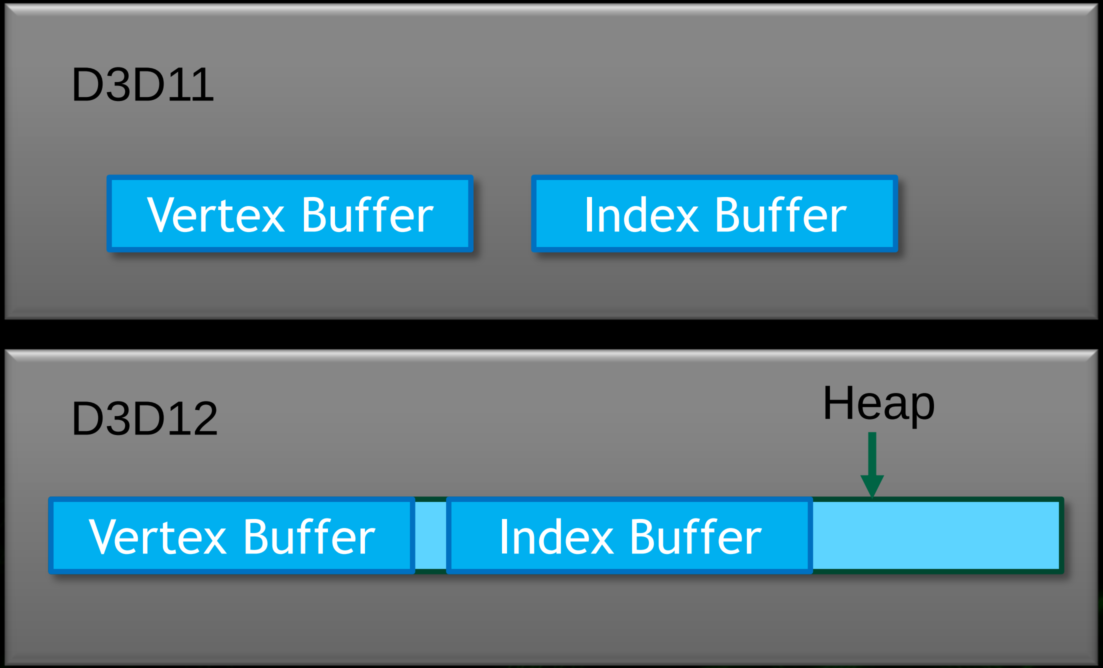
Output Merger

Pipeline State Object (cont)

- No implicit shader recompiling and linking during rendering.
- Resolve state to many hardware instructions earlier.
- PSO takes binary shader as output, shader cache friendly.
- Still need our attention:
  - Create a PSO in a separate thread
  - Use same values for don’t-care fields
  - Use similar PSOs among successive draw calls

Flexible Memory Allocation

- Heap based memory allocation
  - Texture
  - Buffer (VB/IB/CB)
  - Descriptors
  - Sampler



Resource Binding Model

- There are only four types of View in D3D11, there will be more in D3D12
  - Constant Buffer View
  - Vertex Buffer View
  - Index Buffer View
  - …
- And they are no longer D3D objects, you are in control of managing the memory directly

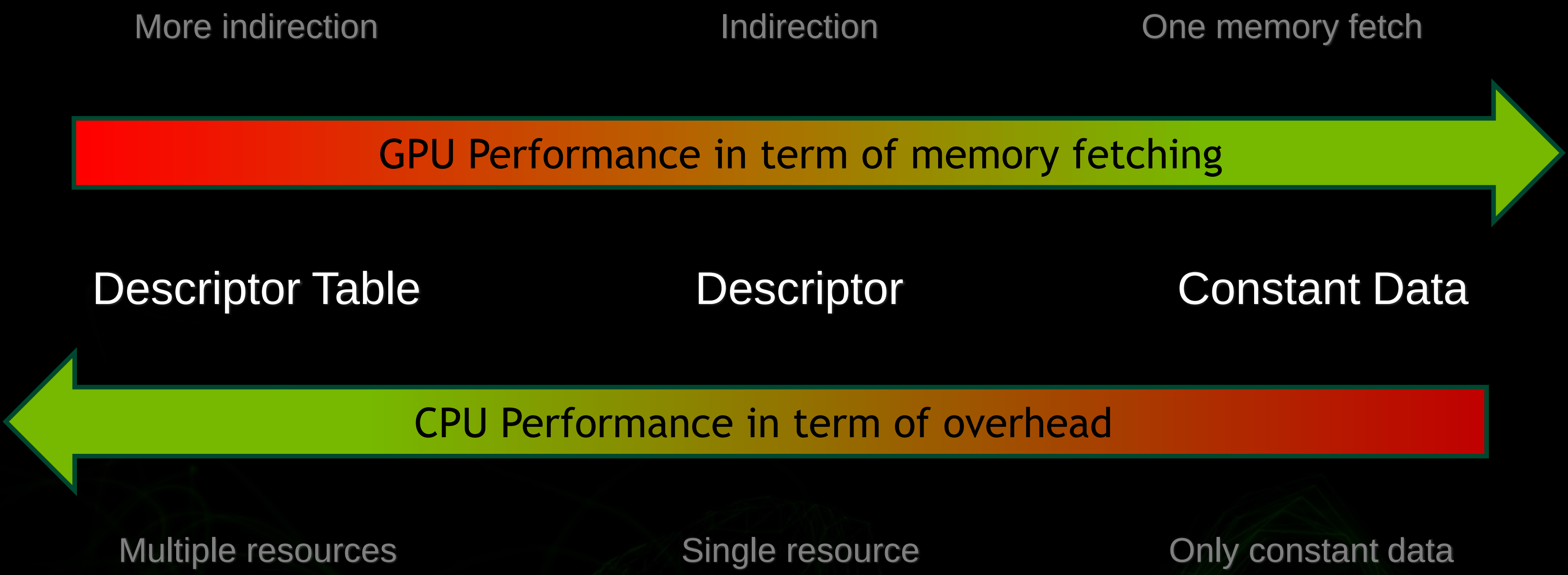
New Resource Binding model

- The following resources are set in a similar manner:
  - Render Target
  - Vertex/Index Buffer (through views, not resource handle)
  - Viewport/Scissor Rect
- There are dramatic changes for setting the following resources:
  - Texture
  - Constant Data
  - Sampler
- There are more to set in D3D12:
  - PSO
  - Root Signature
  - Heap

New Resource Binding Model (cont)

- D3D12 introduces a new type of object called “RootSignature”.
  - It is the only window for setting resources for shader stages.
- Three type of data:
  - Descriptor table
  - Descriptor
  - Constant Data

Balance Overhead in Your Case



More indirection

Indirection

One memory fetch

GPU Performance in term of memory fetching

Descriptor Table

Descriptor

Constant Data

CPU Performance in term of overhead

Multiple resources

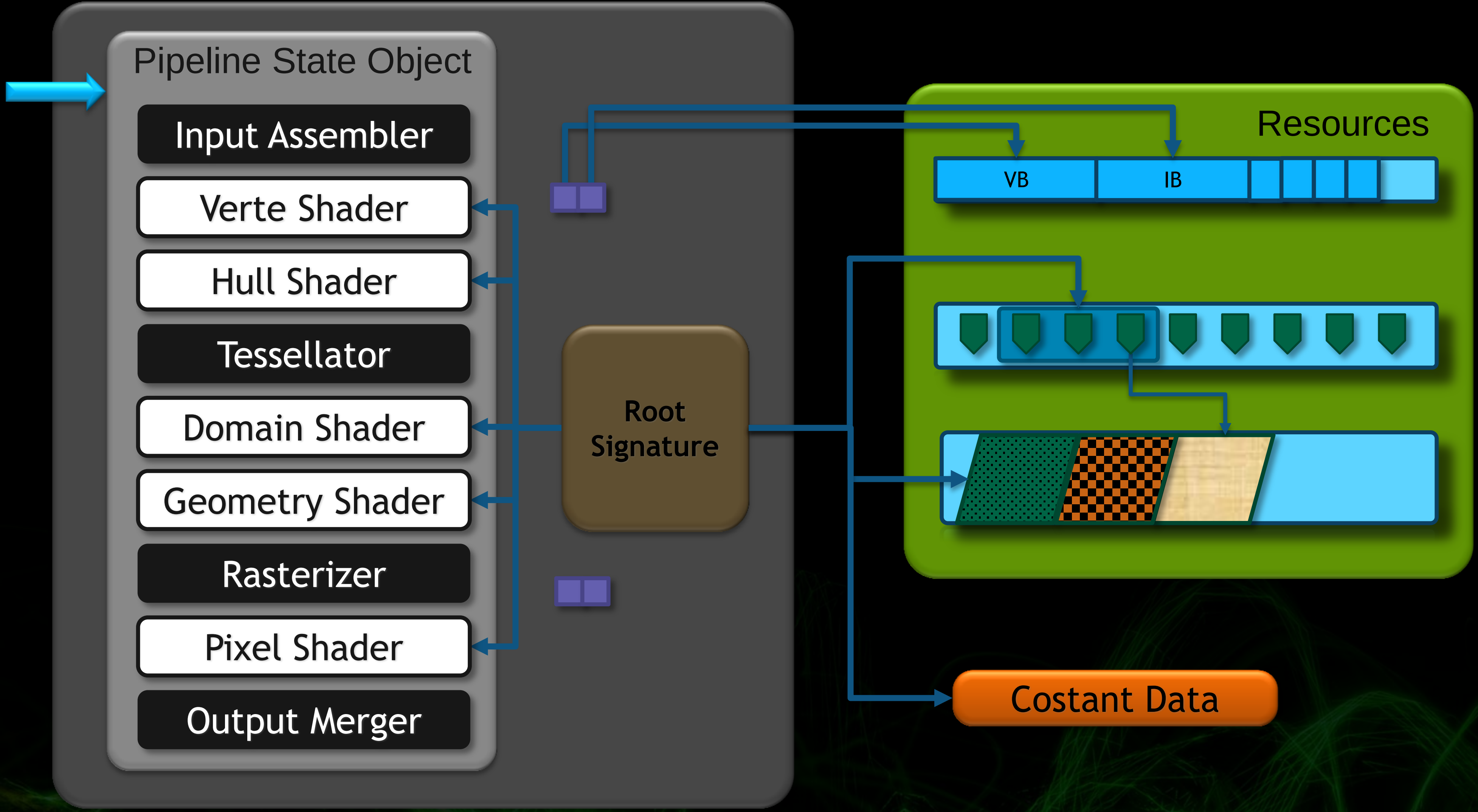
Single resource

Only constant data

Be Careful with your RootSignature

- Keep the size of your RootSignature smaller
- Limit shader visibility to a minimum set
- Only change data when necessary

The New D3D12 Pipeline



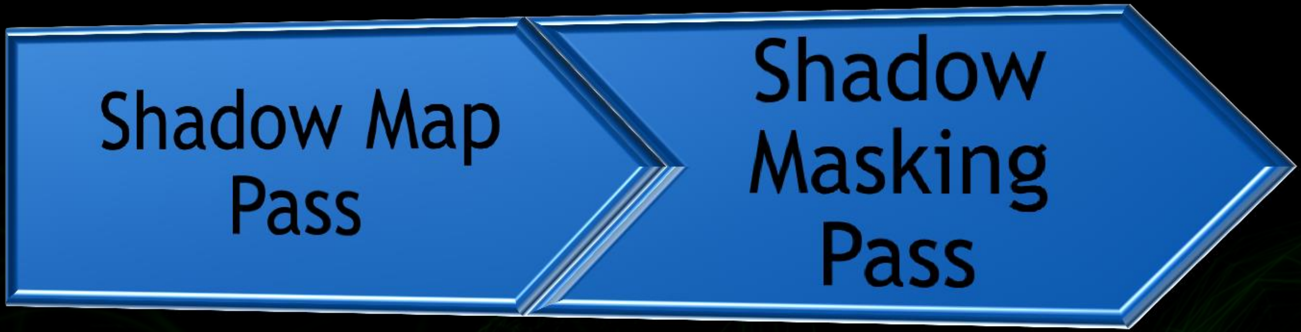
Issues of Resource Management

- Everything is deferred in D3D pipeline, make sure you don’t change anything that is already queued.

- Handle the following issues by yourself
  - Resource lifetime management
  - Resource residency management
  - Resource hazard

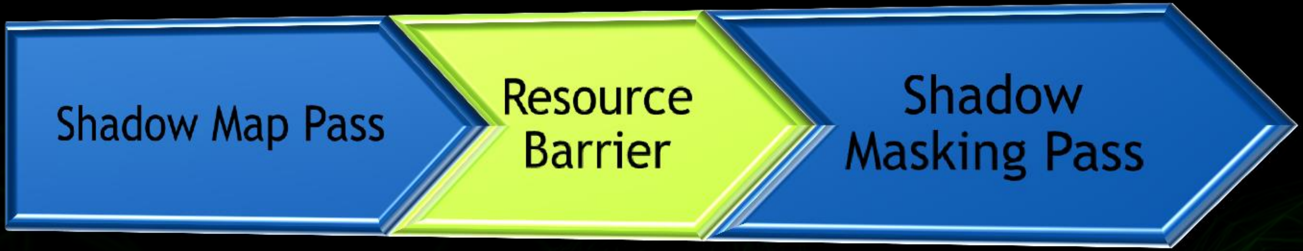
Avoid Resource Hazard

- State switching of D3D11 resources is implicit
- In D3D12, developer should take control of it
  - ResourceBarrier



Avoid Resource Hazard

- State switching of D3D11 resources is implicit
- In D3D12, developer should take control of it
  - ResourceBarrier



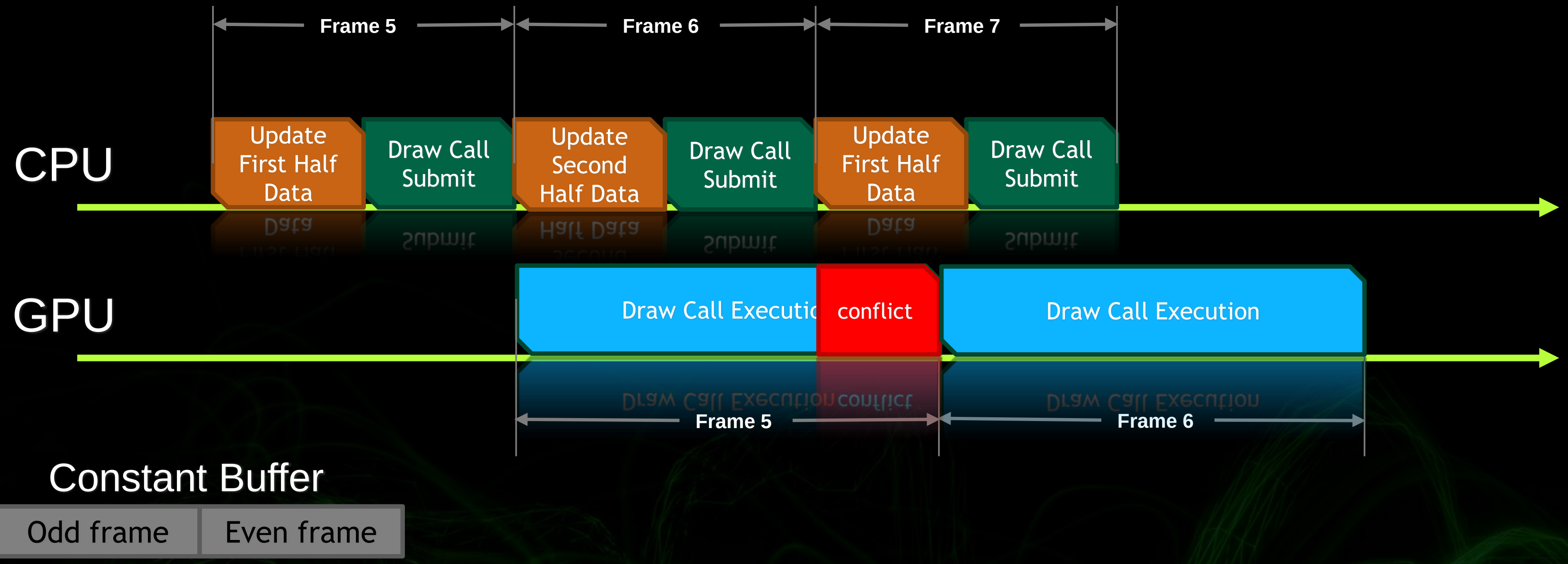
Shadow Map Pass

Resource Barrier

Shadow Masking Pass

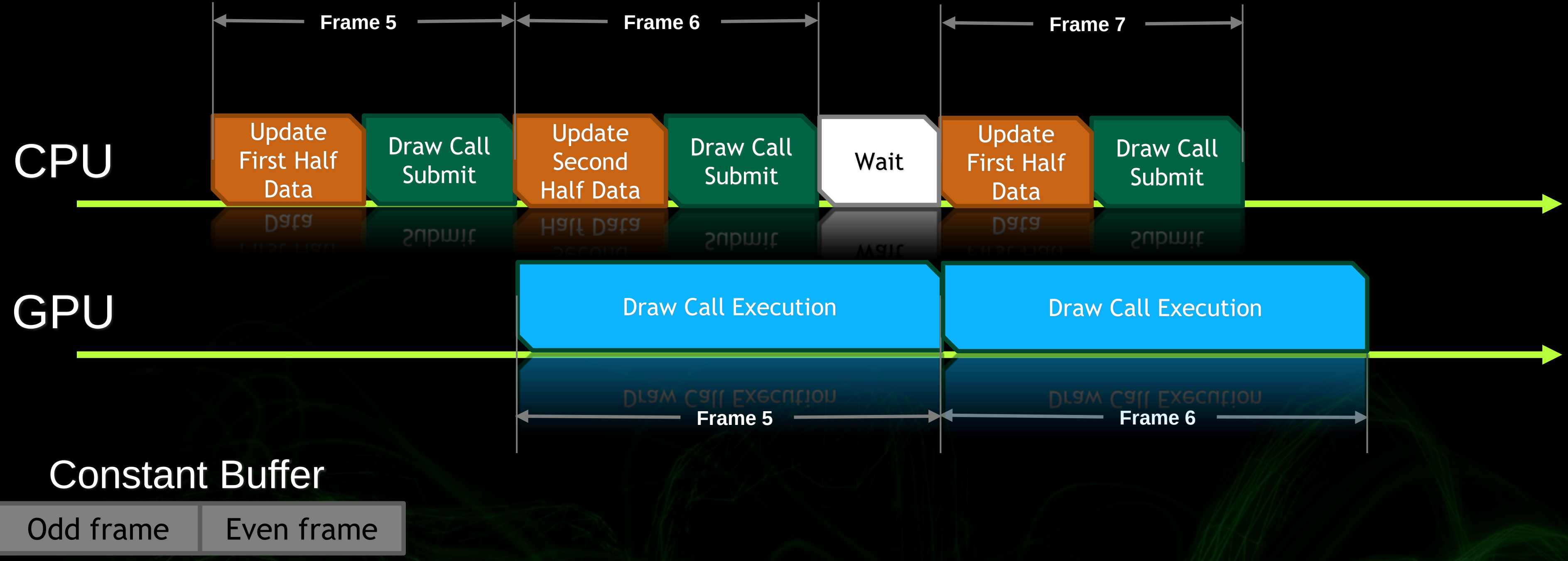
Another Example of Conflict

- Make sure you do not stamp on memory in use.



Another Example of Conflict

- Make sure you do not stamp on memory in use.

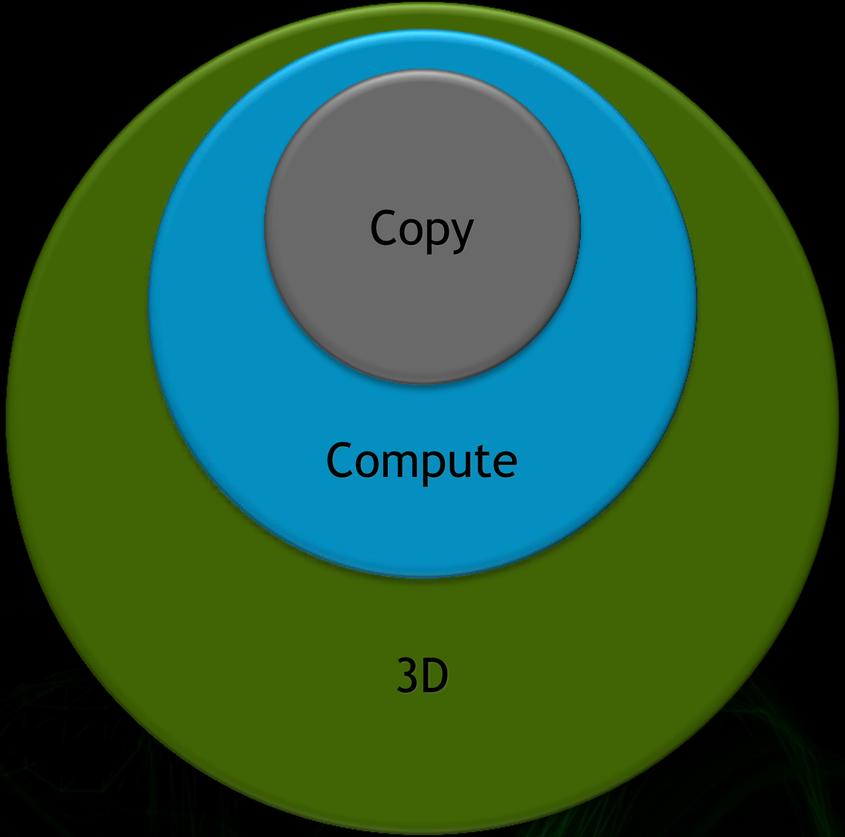


Typical Resource Hazard Scene

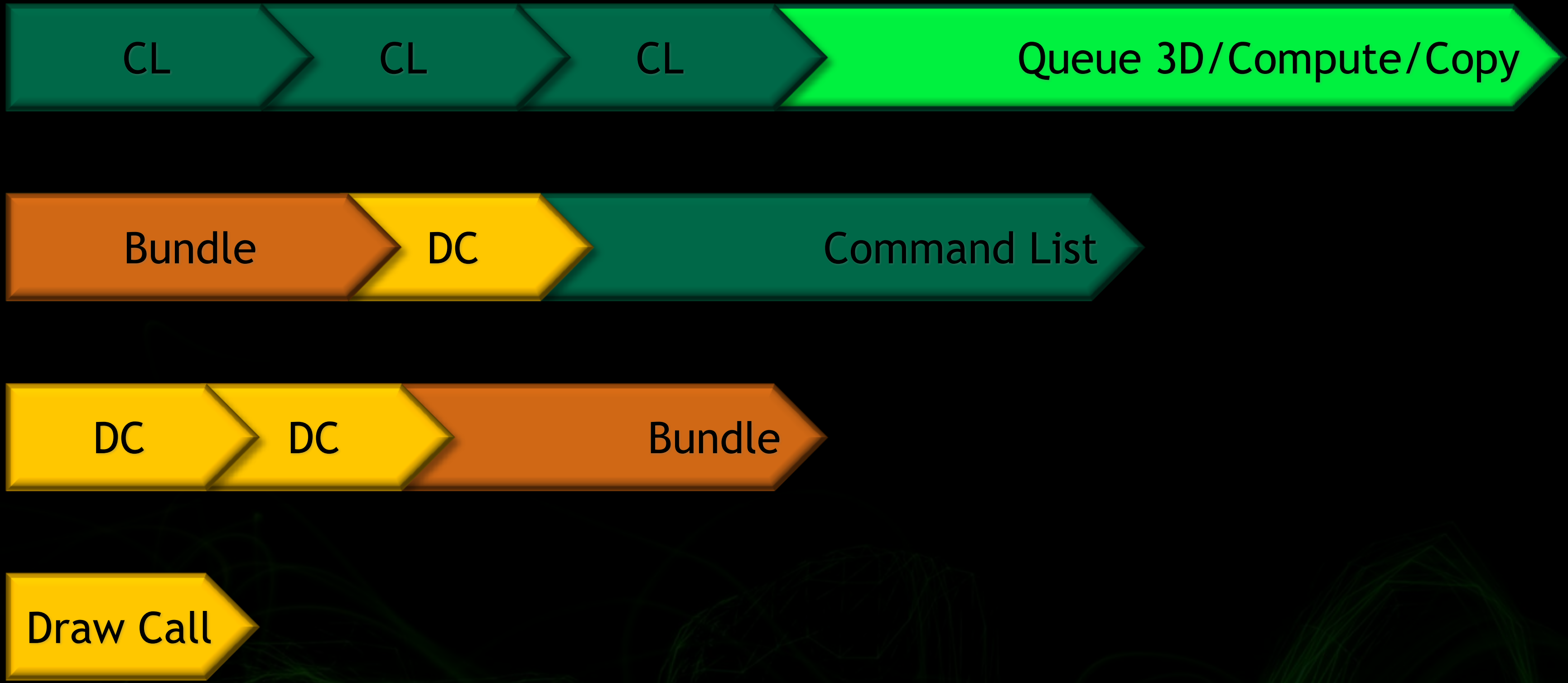
- Shadow map
- Deferred Shading/Lighting
- Real-time Reflection and Refraction
- …
- In any case that render target is used as texture in following draw calls

New Concepts in Execution Model

- Command Queue
  - 3D queue
  - Compute queue
  - Copy queue
- Command List
- Bundle



Execution Model



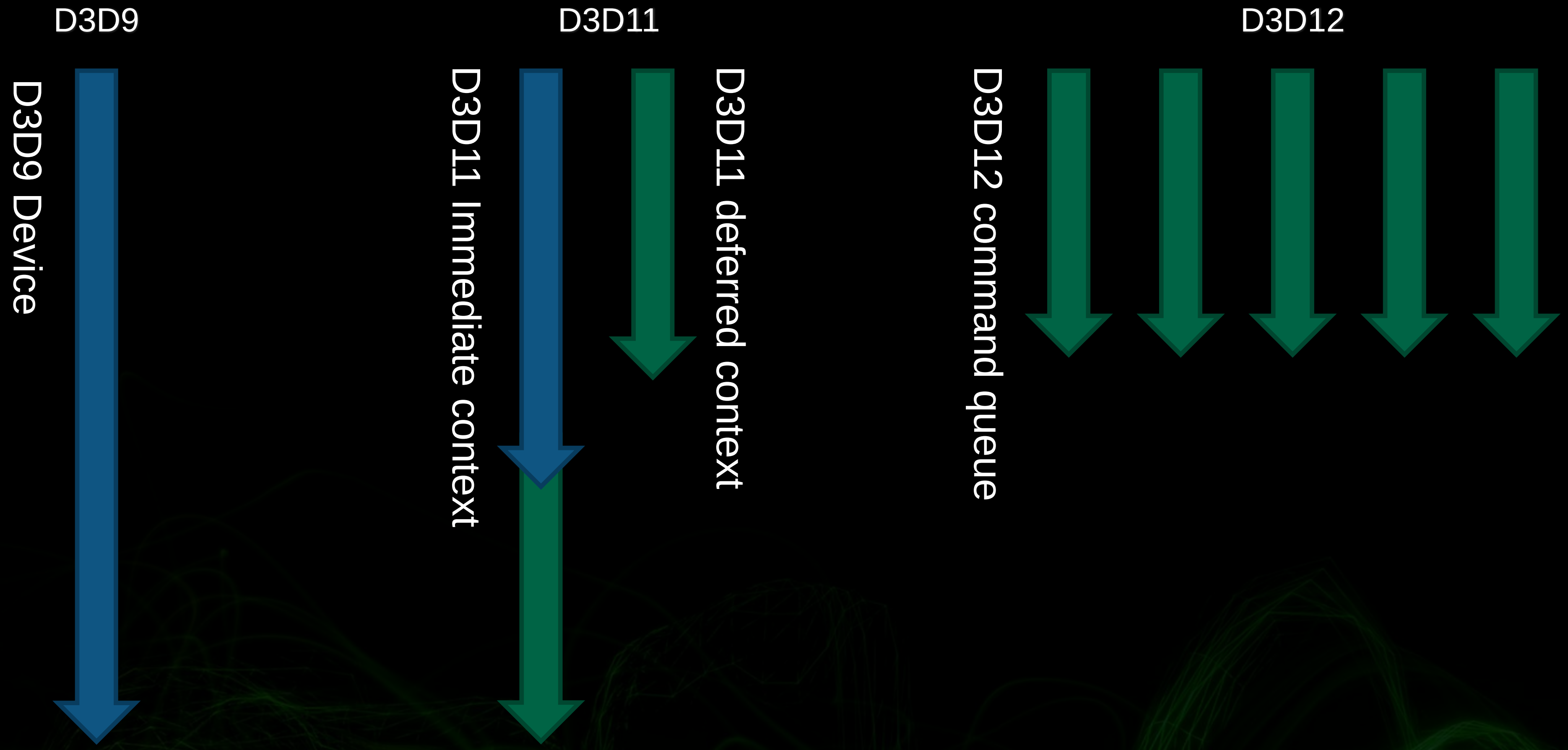
Steps to Issue Draw Calls

- No more immediate context.
- To issue a draw call
  1. Create a 3D queue
  2. Create a command list
  3. Record the draw call in the command list
  4. Execute command

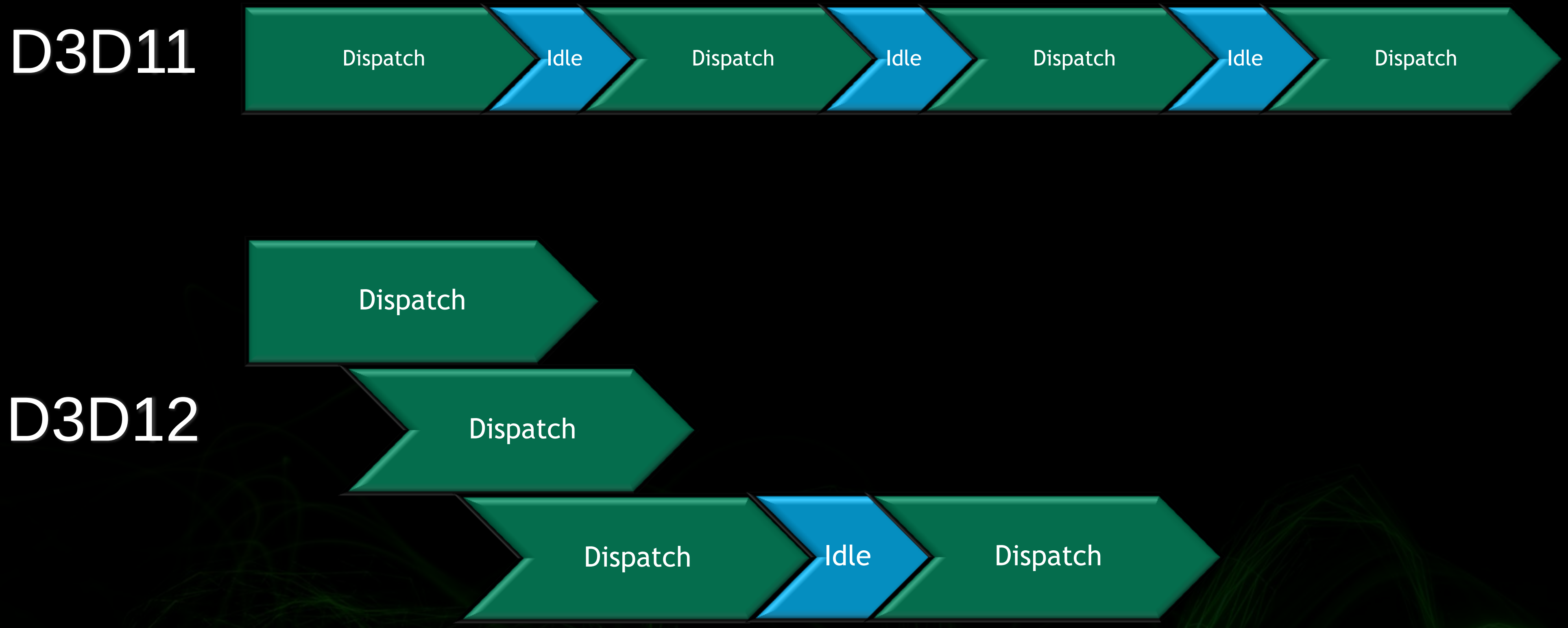
Multi-thread Rendering

- Old multi-thread rendering model
  - One dedicated thread for submitting draw/dispatch calls.
  - Several other thread for other things, like AI, visibility test.
- The new model
  - Several threads for anything

Multi-thread Rendering (cont)



Better GPU Efficiency



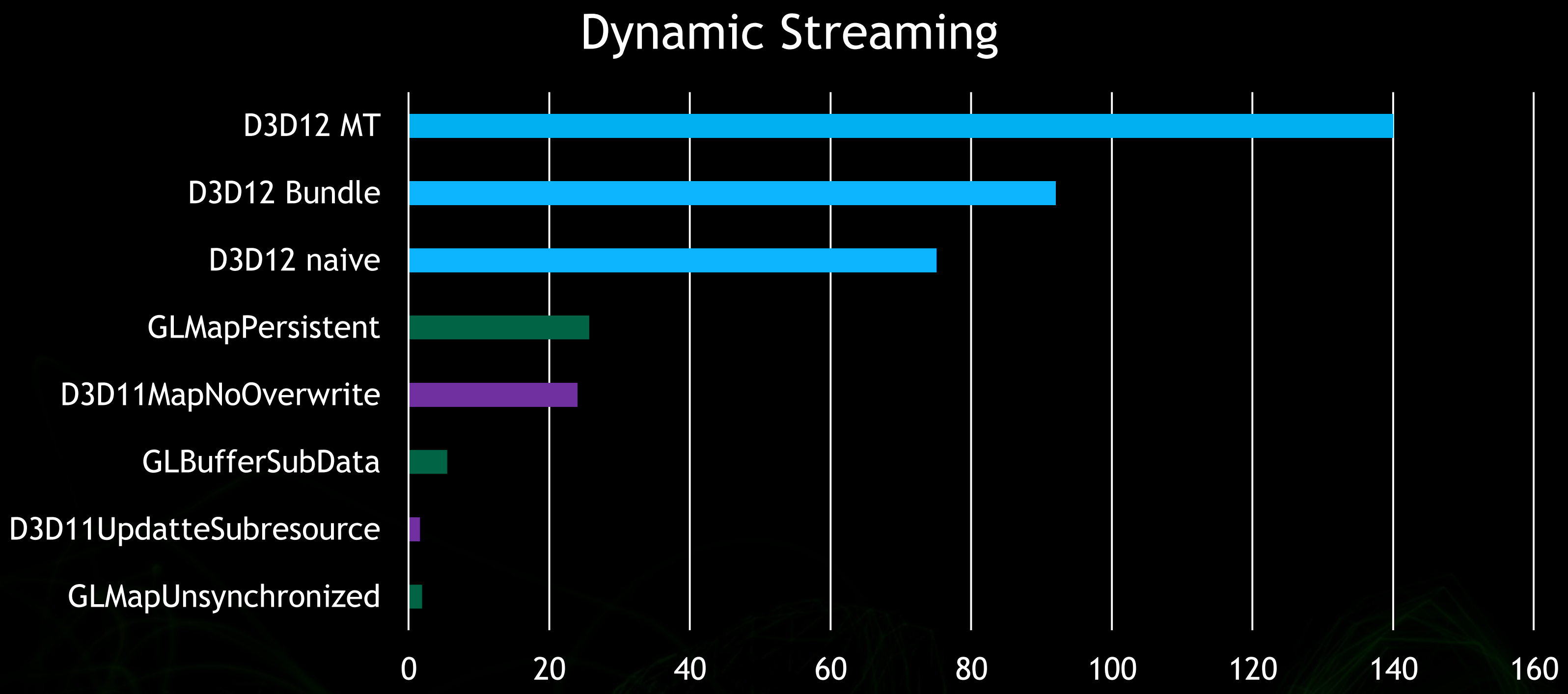
Porting from D3D11 to D3D12

- A low hanging fruit: D3D11on12
- Only minor changes in your D3D11 code:
  - Create D3D12 device
  - Create wrapped resource for back buffer
  - Manage render targets explicitly
  - Flush right before present
  - Fence your frame
- Performing a full porting is necessary, don’t expect too much on D3D11on12.

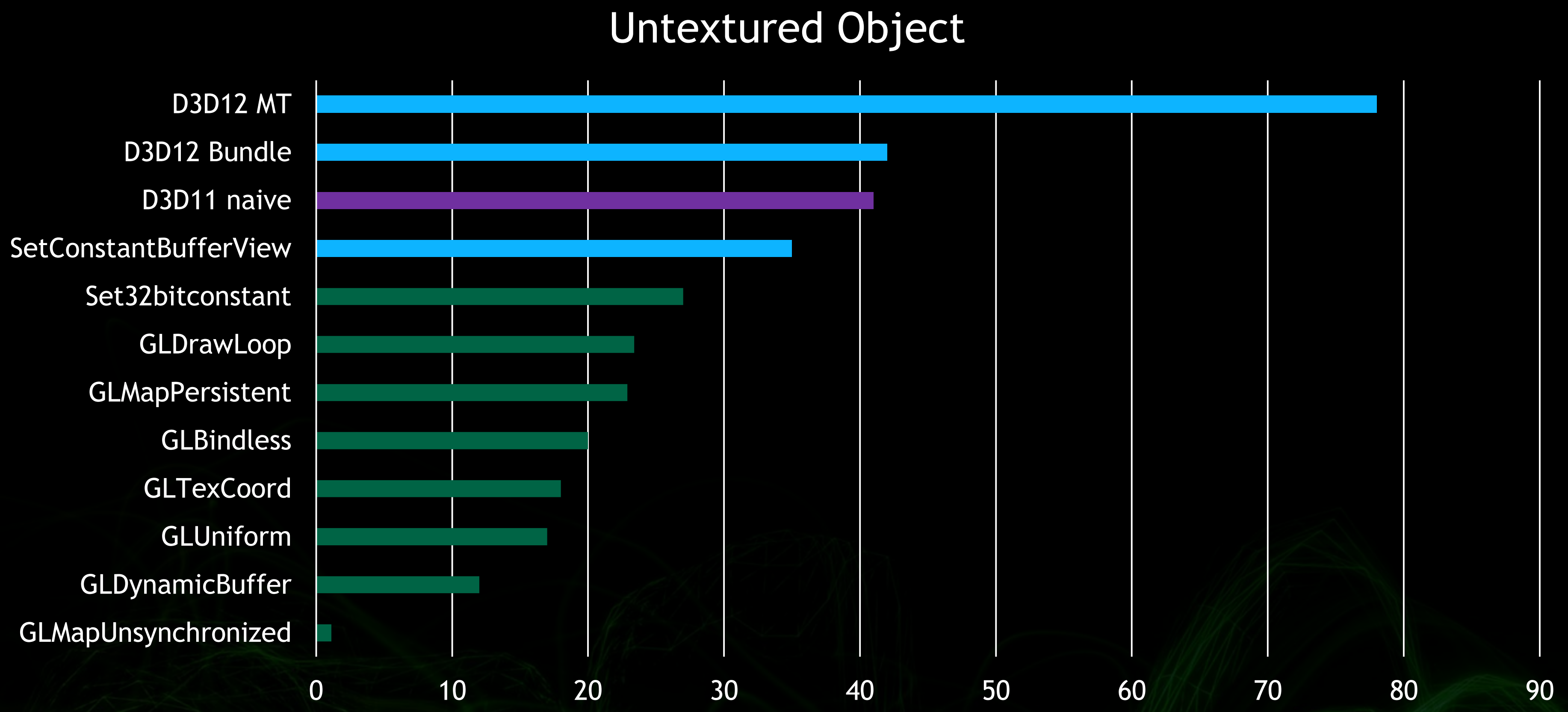
API test (Extended version)

- API test is a simple benchmark program for testing API performance.
- There are four problems:
  - Clear
  - Dynamic streaming, 250000 particles, each with different vertex buffer data
  - Untextured Objects, 64x64x64 objects, each with different constant data
  - Textured Objects, 160000 quads, each with different textures
- Get the source on github:
  - https://github.com/JerryCao1985/apitest

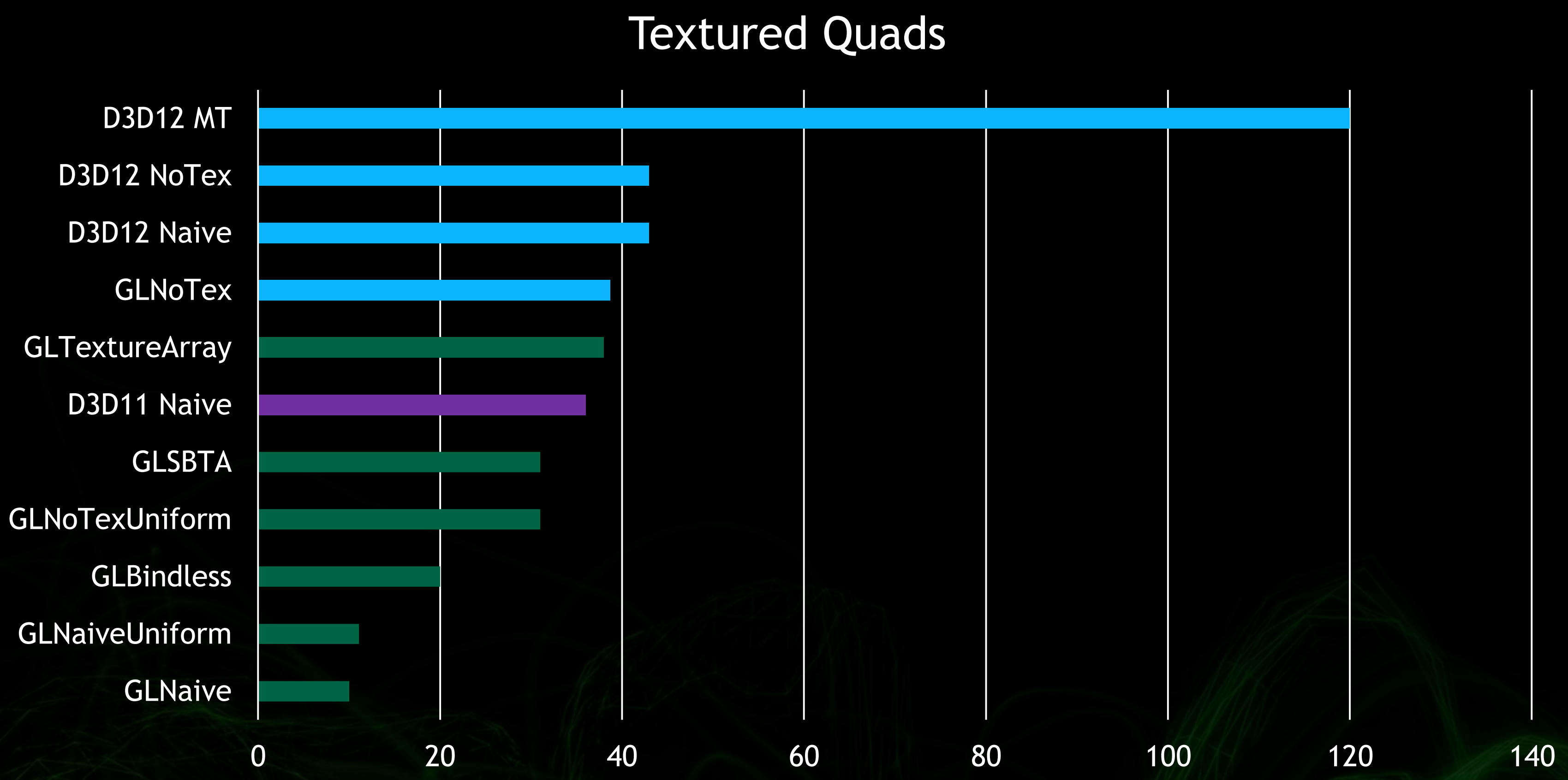
Performance



Performance



Performance



New Graphics Features

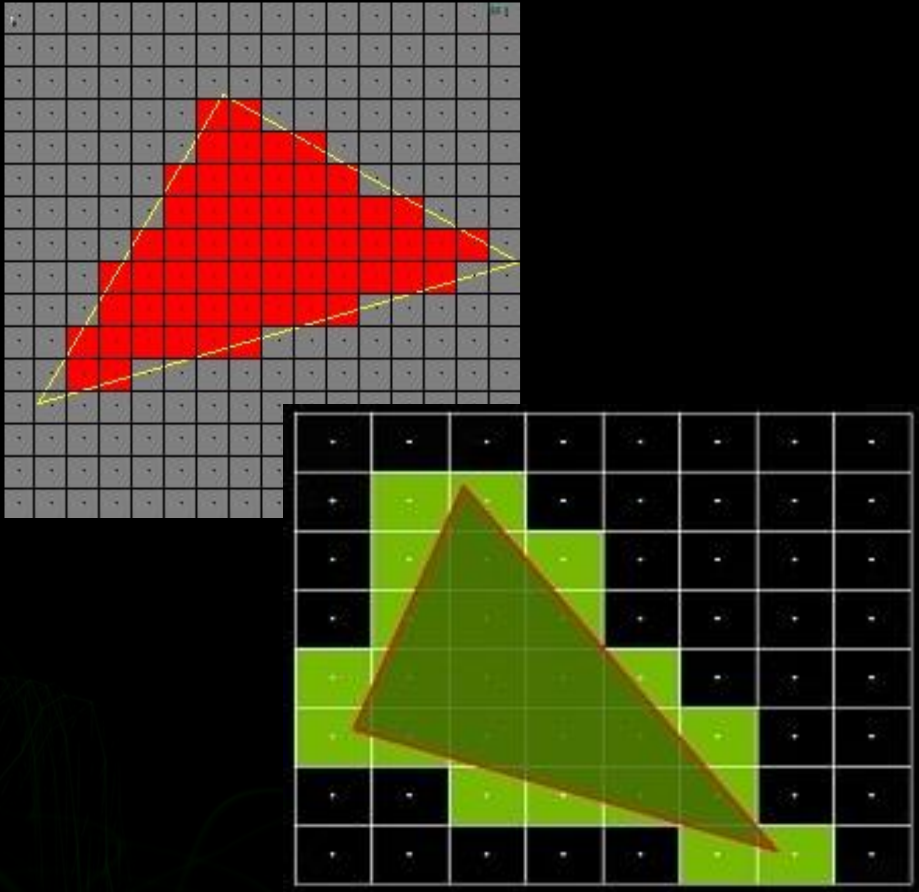
- Conservative Rasterization
- Raster Order View
- Tiled Resources (Volumes, 3D Texture)
- Typed UAV Load
- PS Specified Stencil Reference

Conservative Rasterization

- Draws all pixels a triangle touches
  - Different Tiers - see DX spec

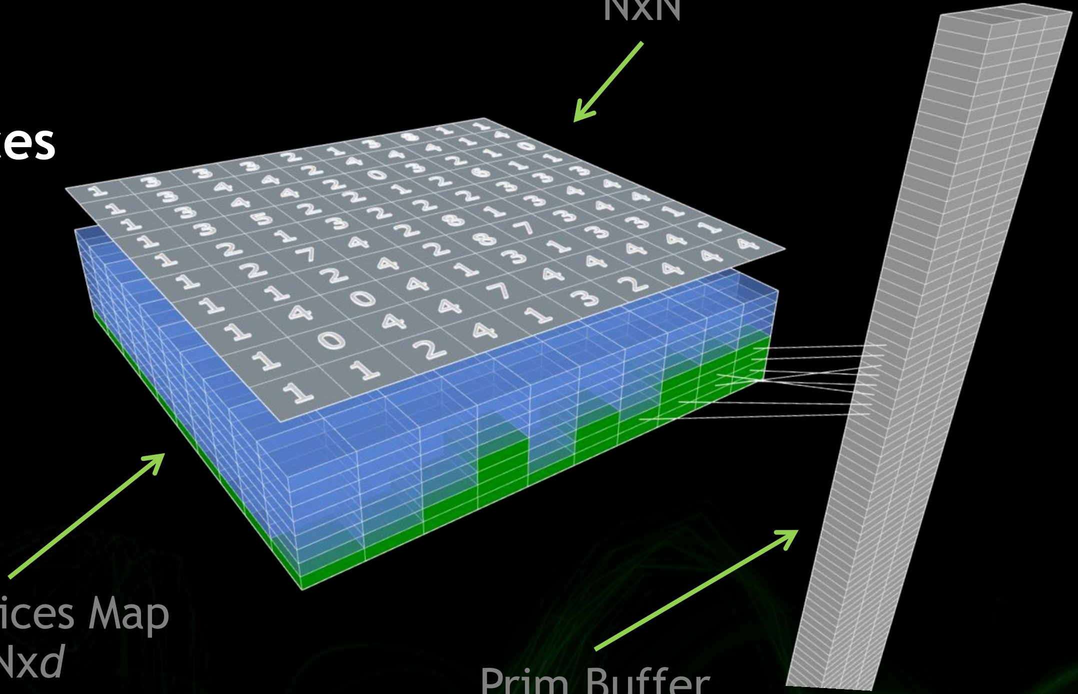
- Possible before through GS trick but relatively slow
  - See J. Hasselgren et. Al, “Conservative Rasterization“, GPU Gems 2

- Now we can use rasterization do implement some nice techniques!



Hybrid Raytraced Shadows

- Prim Buffer - Triangle vertices
- Prim Indices Map - Prim buffer indices of triangles
- Prim Count Map - # of tris per texel
- Raytrace triangles in a later pass

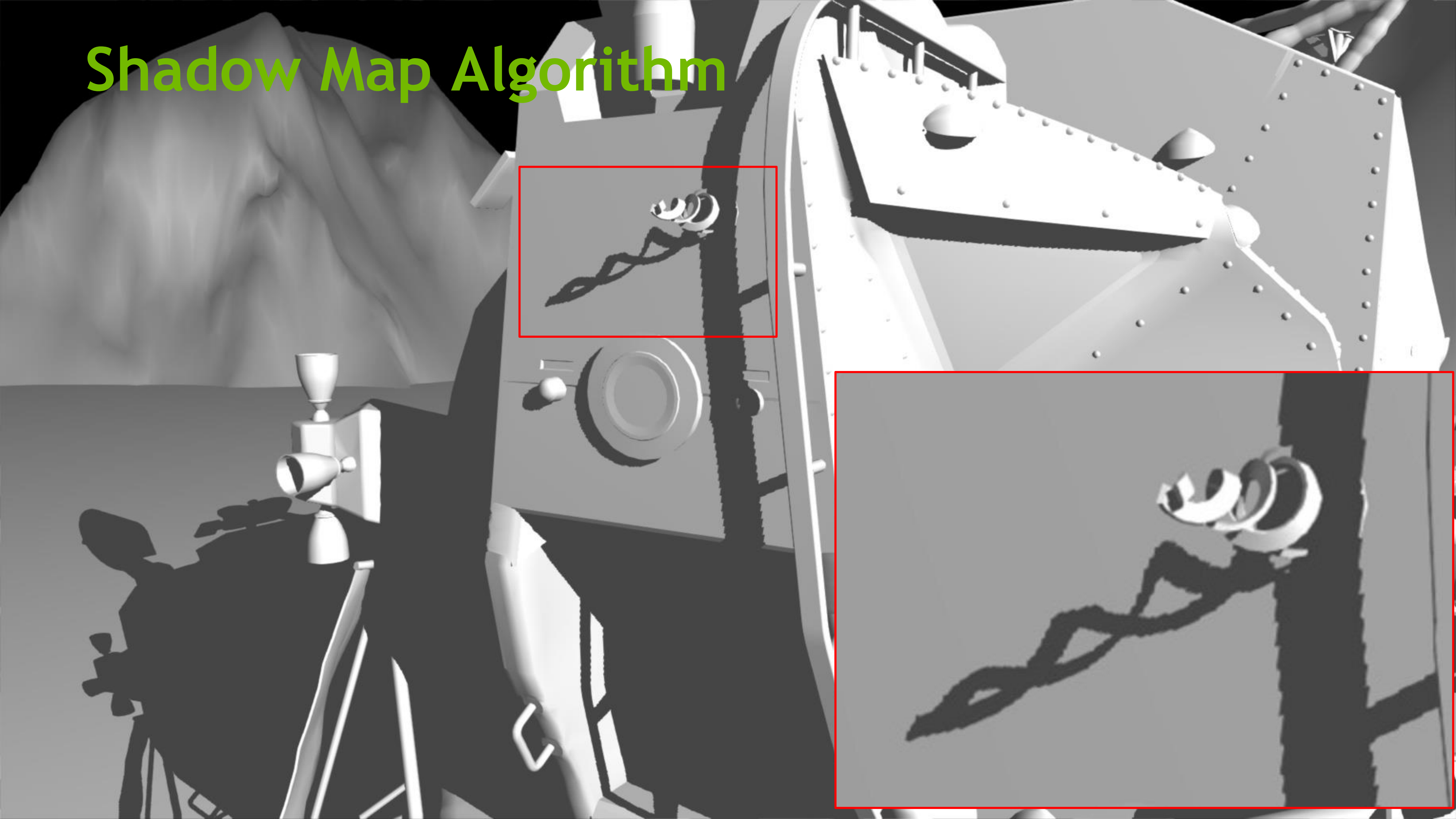


Prim Count Map
NxN

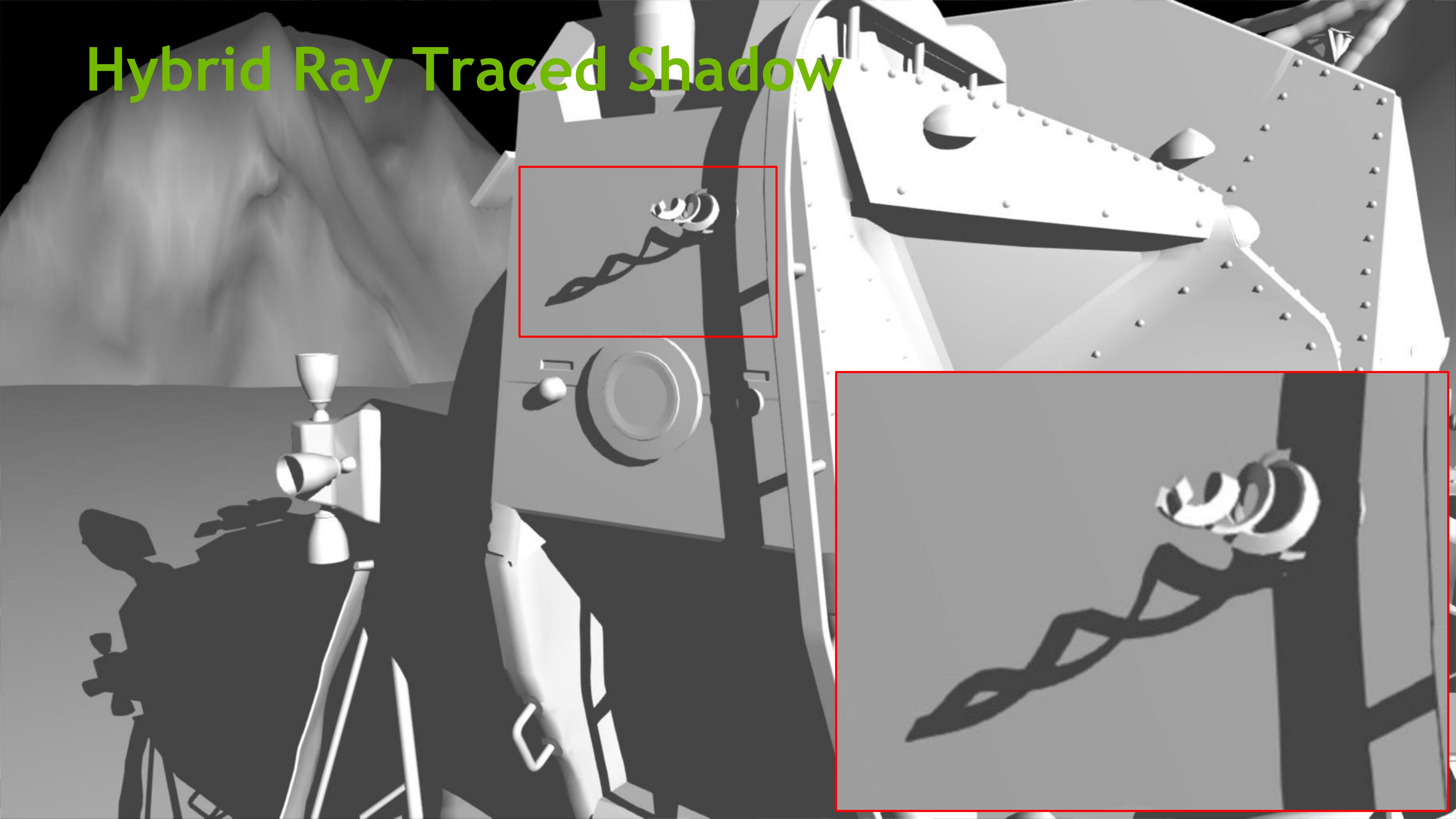
Prim Indices Map
NxNx*d*

Prim Buffer

Shadow Map Algorithm



Hybrid Ray Traced Shadow



Conclusion

- D3D12 better performance
  - Pipeline changes
  - Memory model changes
  - New model of issuing draw/dispatch calls
  - Less dummy wait
- D3D12 performance comparison with other APIs
- D3D12 new graphics features
  - Hybrid Ray traced shadow

Q&A