Advanced Rendering with DirectX 12®

Holger Gruen Senior Developer Technology Engineer, March 16th 2016

NVIDIA

Agenda

- DirectX 12: more control & responsibilities
- How to efficiently drive DirectX 12 on NVidia GPUs
- New DirectX 12 programming model use cases
- DirectX 12 & 11.1 new hardware feature use cases
- Q&A

Agenda

- **DirectX 12: more control** & responsibilities
- How to efficiently drive DirectX 12 on NVidia GPUs
- New DirectX 12 programming model use cases
- DirectX 12 & 11.1 new hardware feature use cases
- Q&A

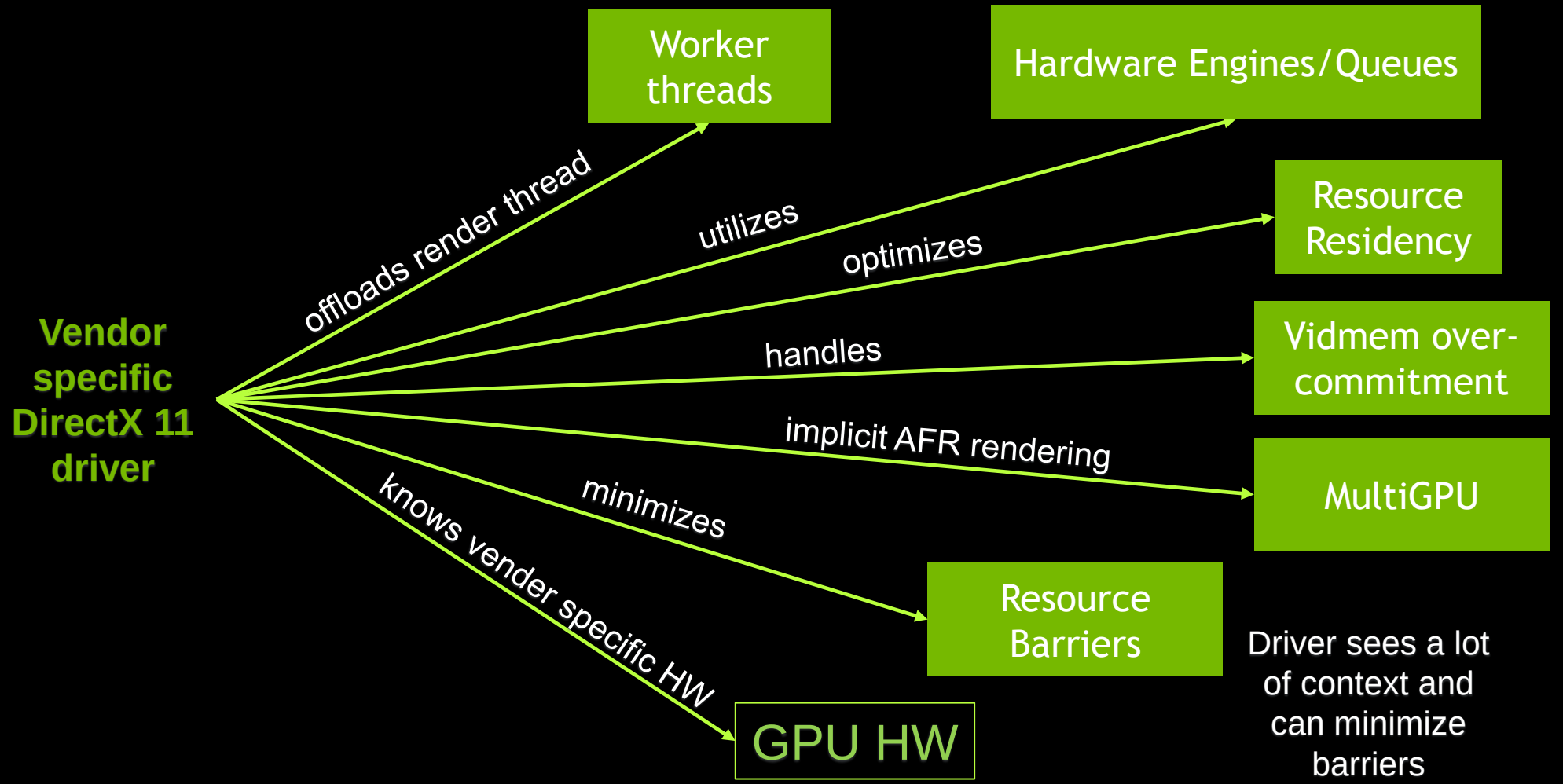
DirectX 12: More Control

- Gives expert programmers more explicit control over the GPU
  - Use multi-threading for faster draw call recording/submission
  - Manage resource residency
  - Explicit Multi-GPU access
  - In general lower level access to GPU HW (e.g. queues)

Agenda

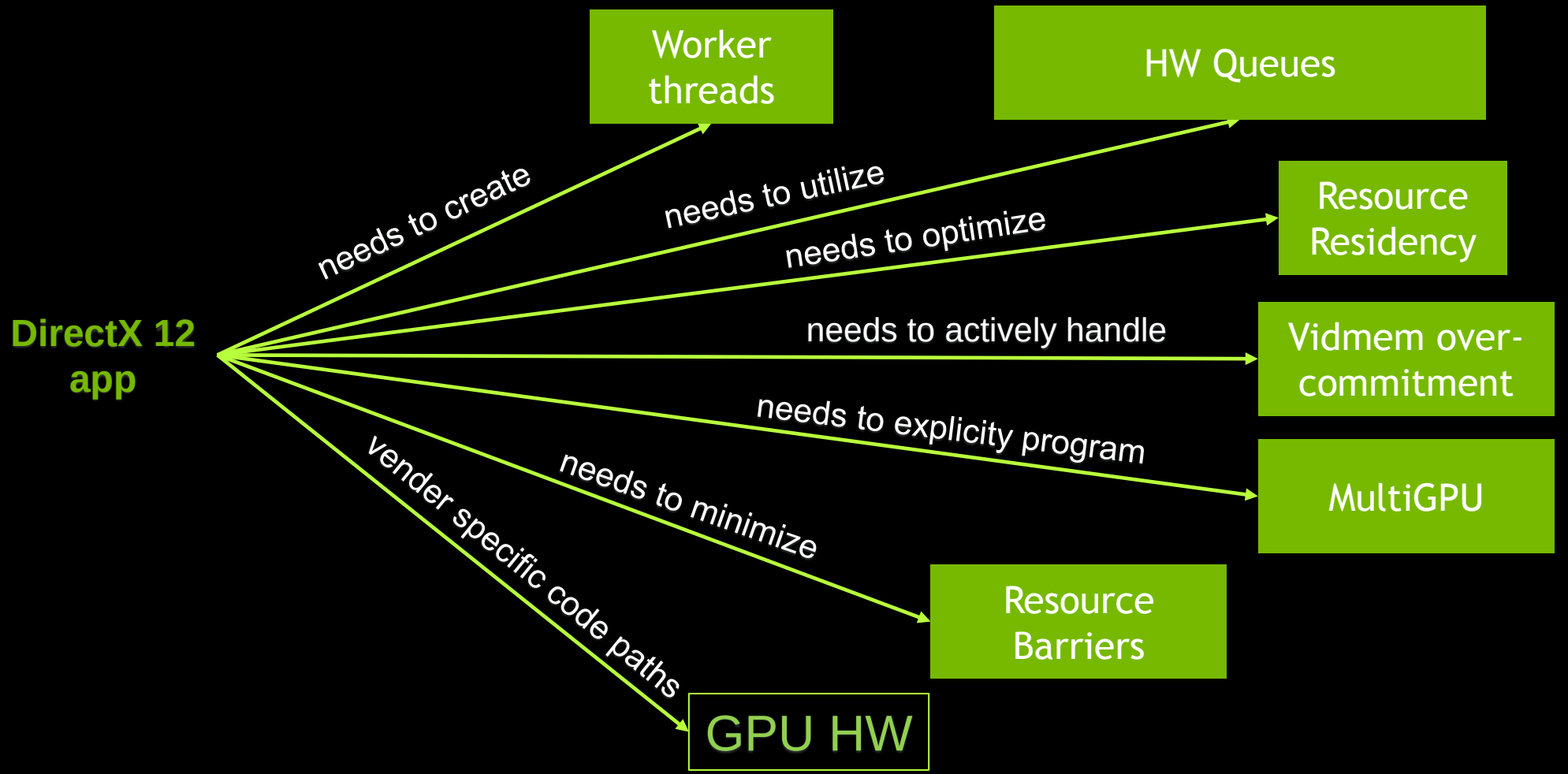
- **DirectX 12:** more control & **responsibilities**
- How to efficiently drive DirectX 12 on NVidia GPUs
- New DirectX 12 programming model use cases
- DirectX 12 & 11.1 new hardware feature use cases
- Q&A

Recap:What does the DirectX11 driver do for you?



Driver sees a lot of context and can minimize barriers

DirectX 12: more responsibilities



Agenda

- DirectX 12: more control & responsibilities
- **How to efficiently drive DirectX 12 on NVidia GPUs**
- New DirectX 12 programming model use cases
- DirectX 12 & 11.1 new hardware feature use cases
- Q&A

Efficient DirectX 12 on NVIDIA GPUs (1/2)

- Construct balanced number of Command Lists (CLs) in parallel
- Make sure barriers and fences are used optimally
- Efficiently handle resource residency
  - You can do a better job than the DX11 driver
- Make sensible use of HW queues

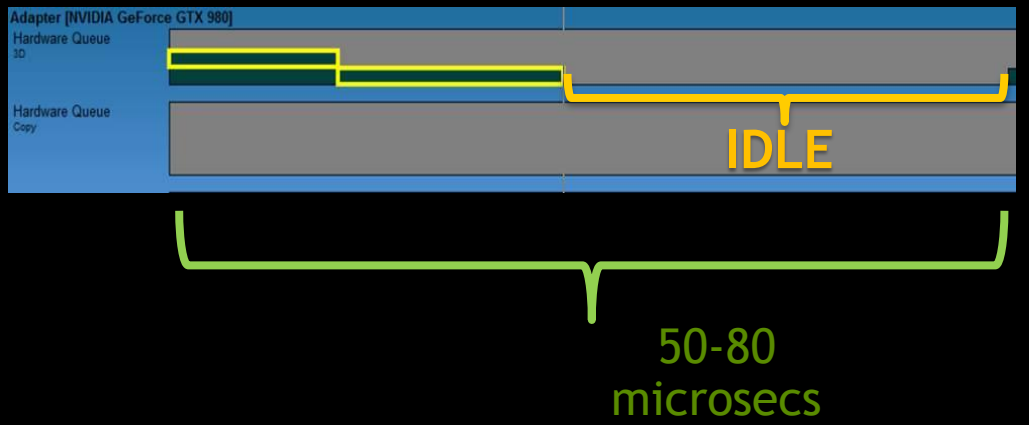
See also Gareth Thomas and Alex Dunns talk held at the **Advanced Graphics Techniques Tutorial Day :** 'Practical DirectX 12 – Programming Model and Hardware Capabilities'

Efficient DirectX 12 on NVIDIA GPUs (2/2)

- Gracefully deal with the hardware tiers of NVIDIA GPUs
- Use CBVs and constants in the root signature when possible
- Strategically flatten shader constants
- Never ever call SetStablePowerState() in shipping code

Command Lists

- Use multiple threads to construct CLs in parallel
- Don’t execute too many CLs per frame, aim for:
  - 15-30 Command Lists
  - 5-10 ‘ExecuteCommandLists’
- Avoid short CLs



Barriers

- You need to get the use of barriers right!
  - Avoid redundancy
- Use minimum set of resource usage flags to avoid redundant flushes
  - Don’t use D3D12_RESOURCE_USAGE_GENERIC_READ
- Use split barriers when possible
- Transition at the end of write
  - Avoid read-to-read barriers

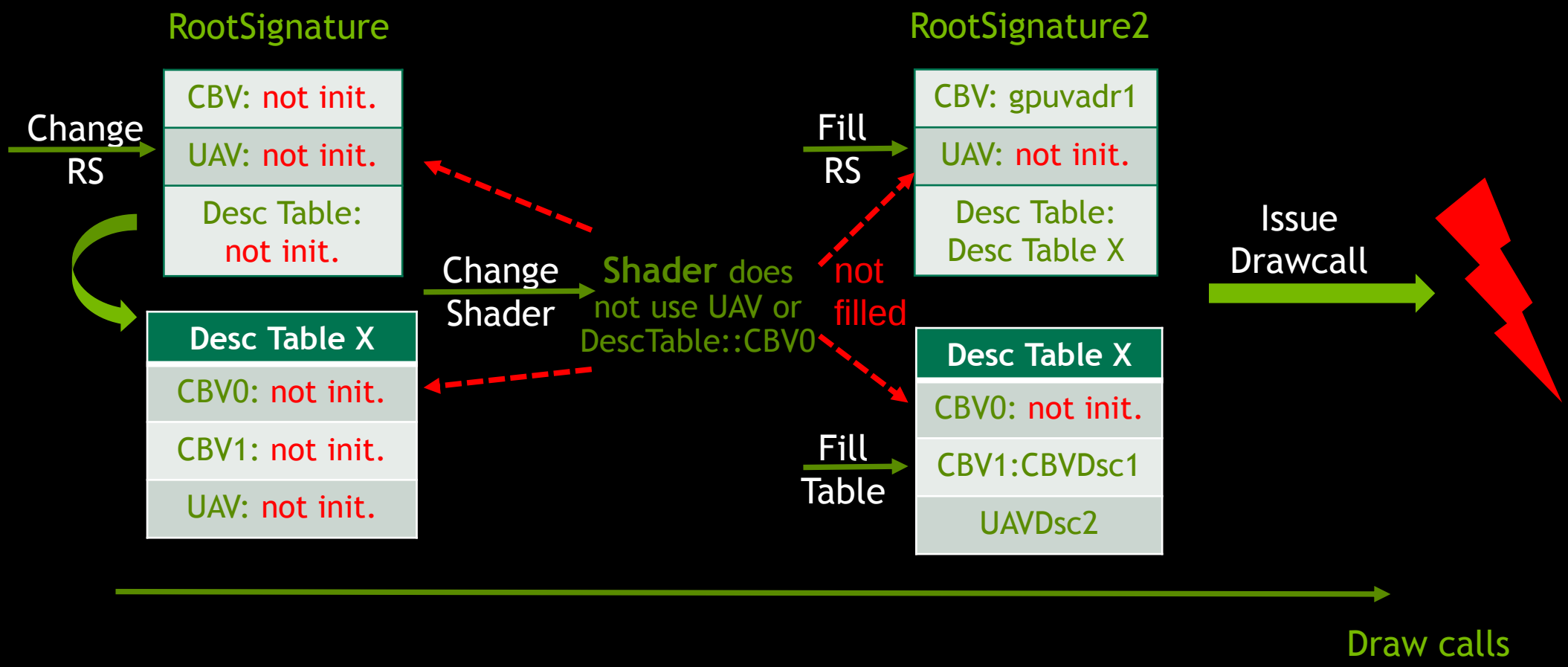
Root Signatures

- Don’t just use one RST
  - Use a reasonably small set of RSTs
- Keep RSTs small
- If possible place constants and CBVs in the RST
  - Constants/CBVs in the RST speeds up shaders - target PS first
- Limit resource visibility to the minimum set of stages
  - No D3D12_SHADER_VISIBILITY_ALL if not required
  - Use DENY_ROOT_SIGNATURE_*_ACCESS flags

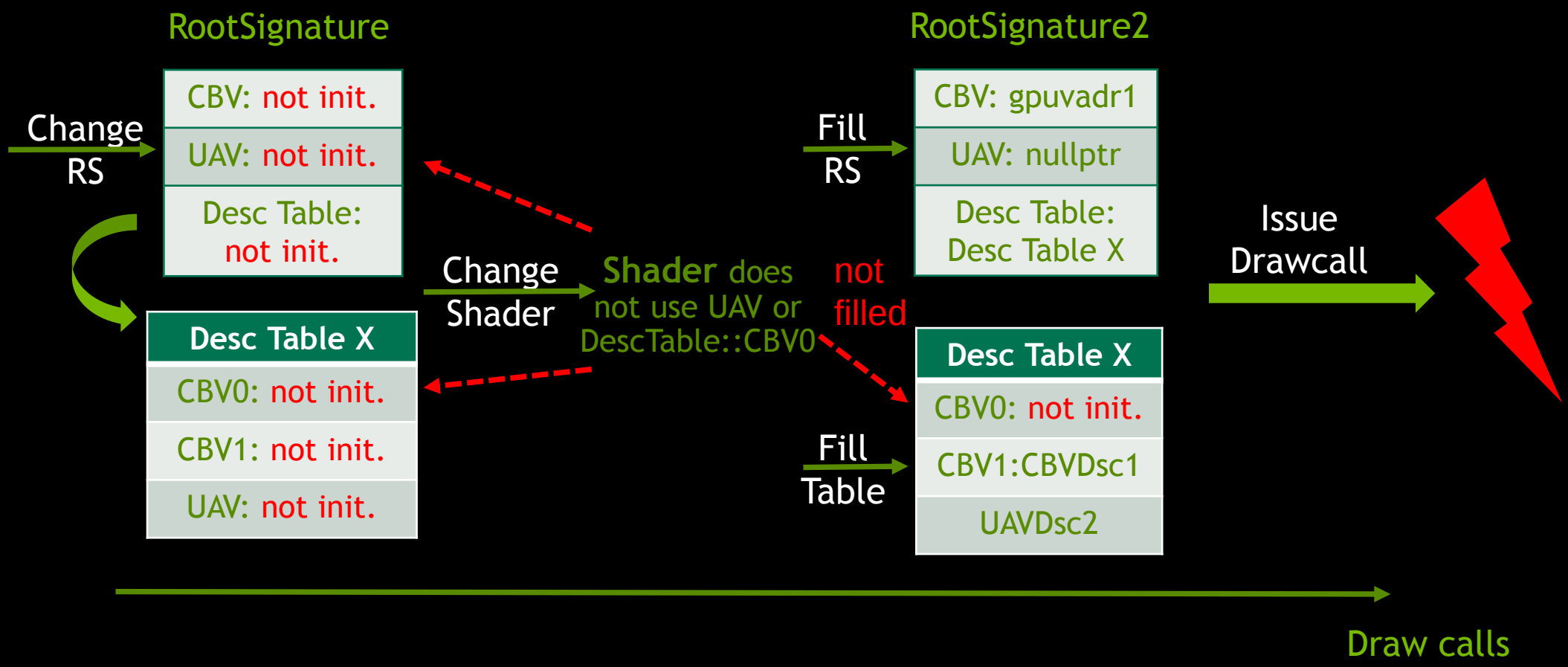
Resource Binding

- Current NVidia GPUs support **Resource Binding Tier 2**
- Gracefully handle CBV and UAV descriptors
  - Fill **all** of the RST (and descriptor tables) with sensible data before a CL executes
  - Even if the used shaders **do not** reference all descriptors
  - Use nullCBVs and nullUAVs in descriptor tables

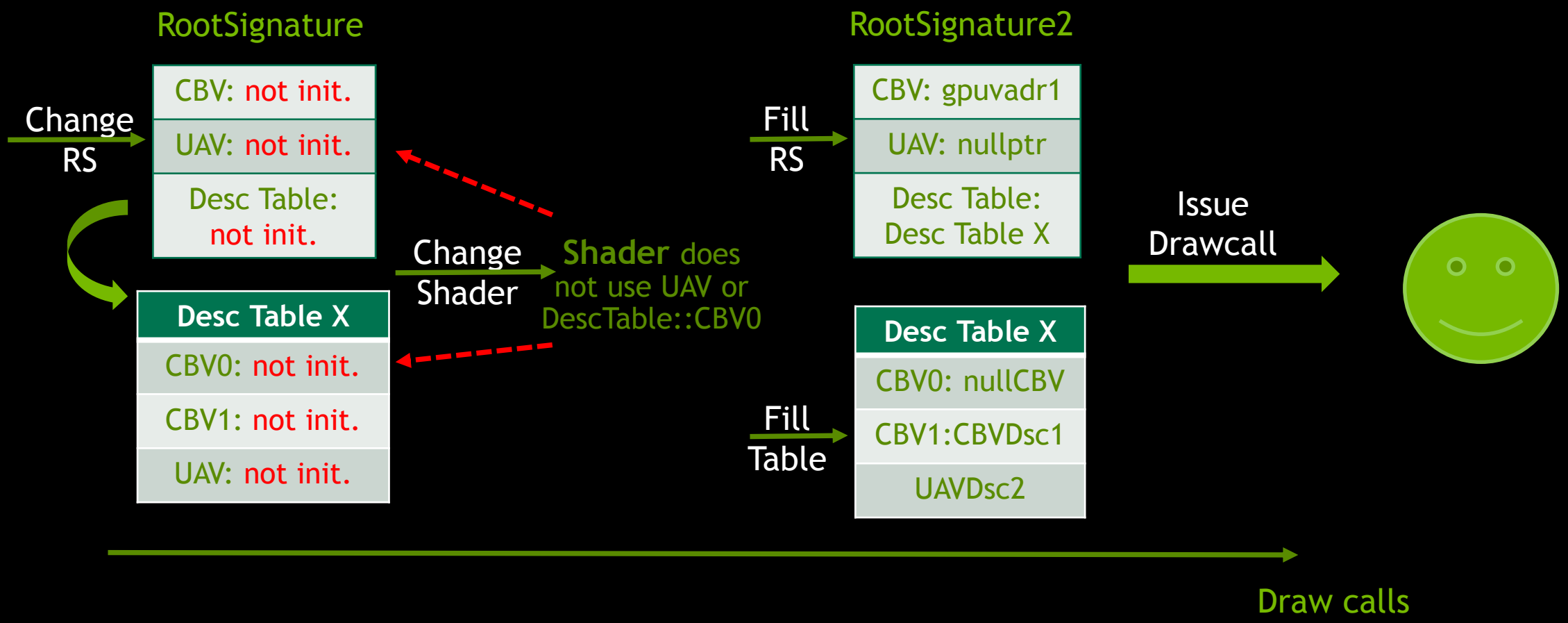
Resource Tier 2 binding gone wrong



Resource Tier 2 binding gone wrong



Resource Tier 2 binding done right



Resource Heaps

- Current NVidia GPUs support Resource Heap Tier 1
  - Max descriptors per heap ~55k
  - UAV count across all stages is limited to 64
  - CBV count is limited to 14 per stage
  - Sampler count is limited to 16 per stage

Strategic Constant Folding for Shaders

- DirectX 12 makes it harder for the driver to fold shader constants
- If you detect a big DX11 vs DX12 perf delta for key shaders
  - Try to strategically fold constants manually
- Generate shaders without folded constants first
  - Go for specialization later – use PSOs when they are ready

Shaders - fold key constants manually

```
cbuffer
{
    float cfSpecWeight;
    …
}

float4 computeLighting(…)
{
    …
    res=CalcLighting(cfSpecWeight);
    …
}
```

manual transform

cfSpecWeight == 0.0f

```
cbuffer
{
#ifdef FOLD_CBSWITCH
    float cfSpecWeightCB;
    #define cfSpecWeight 0.0f
#else
    float cfSpecWeight;
#endif
    …
}

float4 computeLighting(…)
{
    …
    res=CalcLighting(cfSpecWeight);
    …
}
```

Shaders - folding constants manually

```
cbuffer
{
    float cfSpecWeight;
    …
}

float4 computeLighting(…)
{
    …
    res=CalcLighting(cfSpecWeight);
    …
}
```

manual transform

cfSpecWeight == 0.0f

```
cbuffer
{
#ifdef FOLD_CBSWITCH
    float cfSpecWeightCB;
    #define cfSpecWeight 0.0f
#else
    float cfSpecWeight;
#endif
    …
}

float4 computeLighting(…)
{
    …
    res=CalcLighting(0.0f);
    …
}
```

Resource Residency

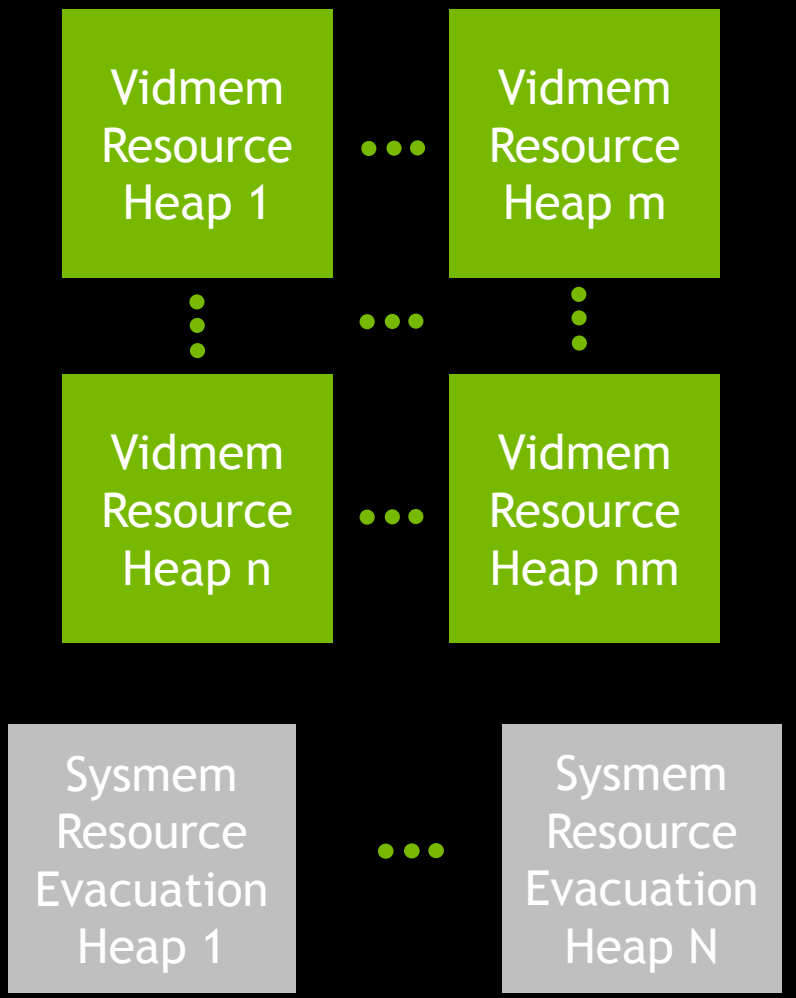
- IDXGIAdapter3::QueryVideoMemoryInfo:How much vid-mem do I have?
  - Foreground app is guaranteed a subset of total vidmem – this is your budget
  - App needs to deal with changes in available mem and Evict() resources
- Use committed resources for RTVs, DSVs, UAVs
- Consider placing small resources in larger committed heaps
- Call MakeResident() on worker threads as it may take some time
  - App must handle MakeResident failure

Video Memory Over-commitment

- DX12 gives user a real advantage over the DX11 driver
  - You what's more important to have in vidmem
- Try to repurpose vidmem heaps
  - Temporarily evacuate vidmem heaps to 'overflow' sysmem heaps
  - Try to repurpose ('older') vidmem heaps
  - Move textures from upload heaps to repurposed vidmem heaps
- Cap graphics settings/resolution based on memory available

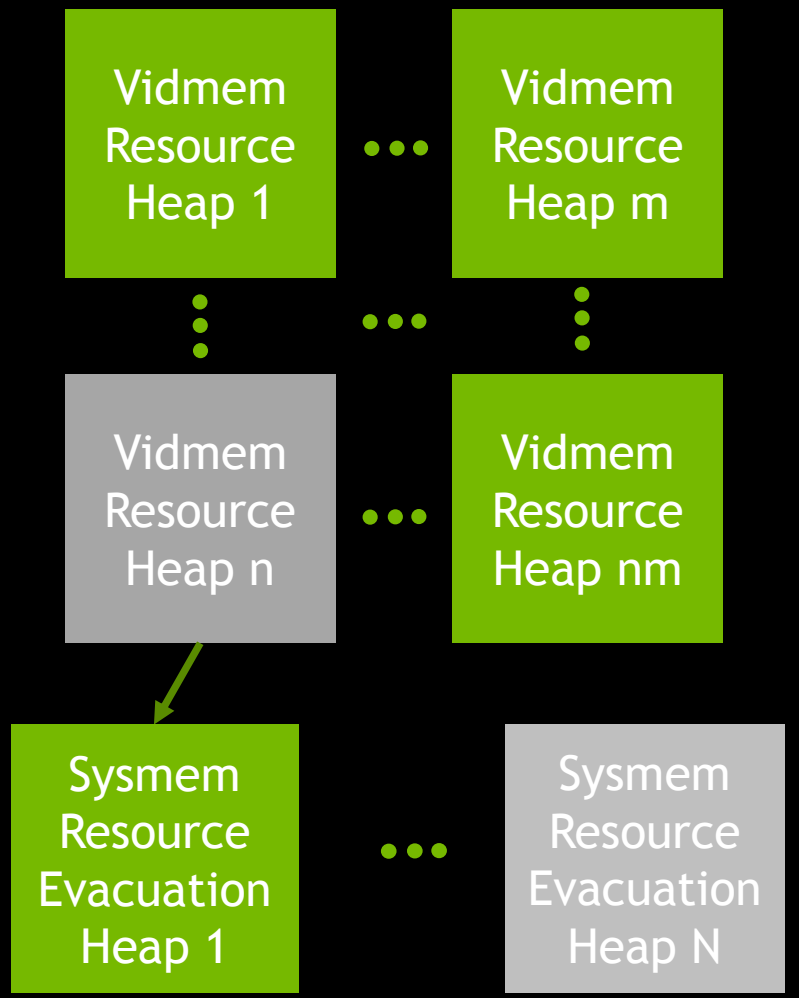
Handling Video Memory Over-commitment

App detects that the next CL needs more committed vidmem than is currently available



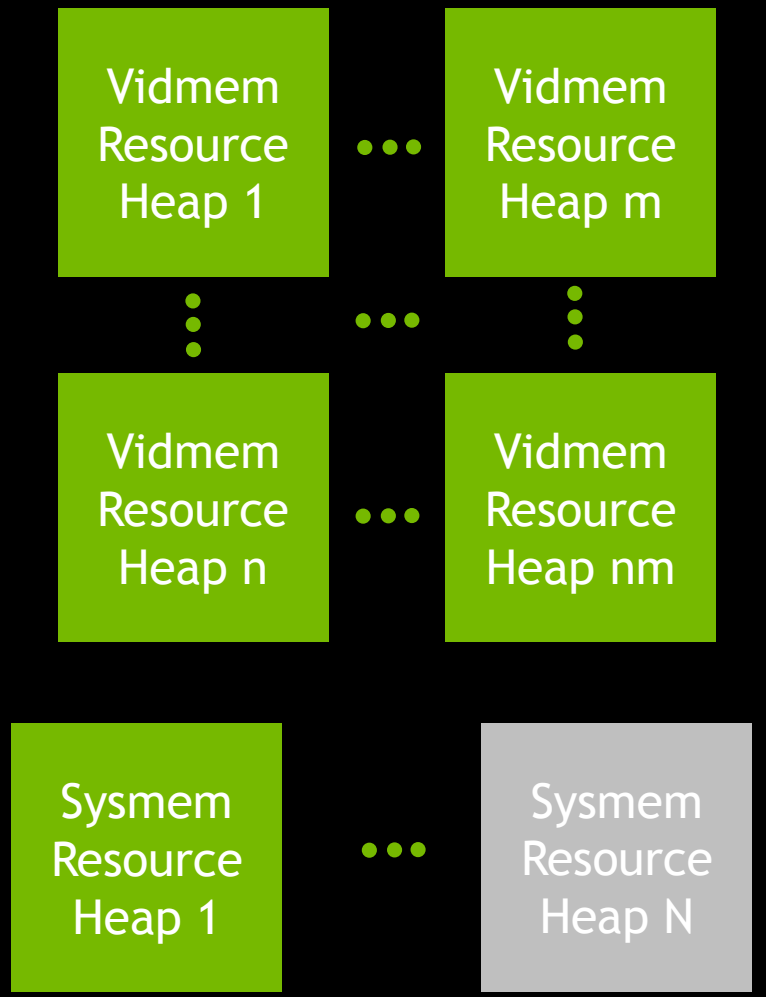
Handling Video Memory Over-commitment

Temporarily evacuate some vidmem resources to a sysmem heap



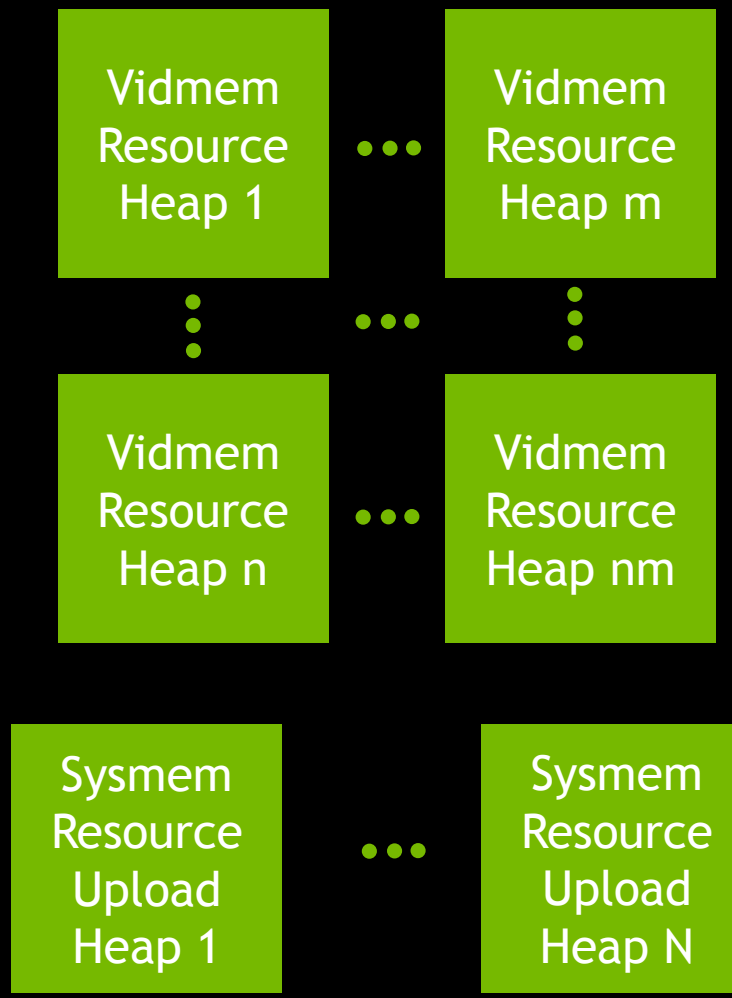
Handling Video Memory Over-commitment

Now reuse vidmem heap for some temp resource requirements



Handling Video Memory Over-commitment

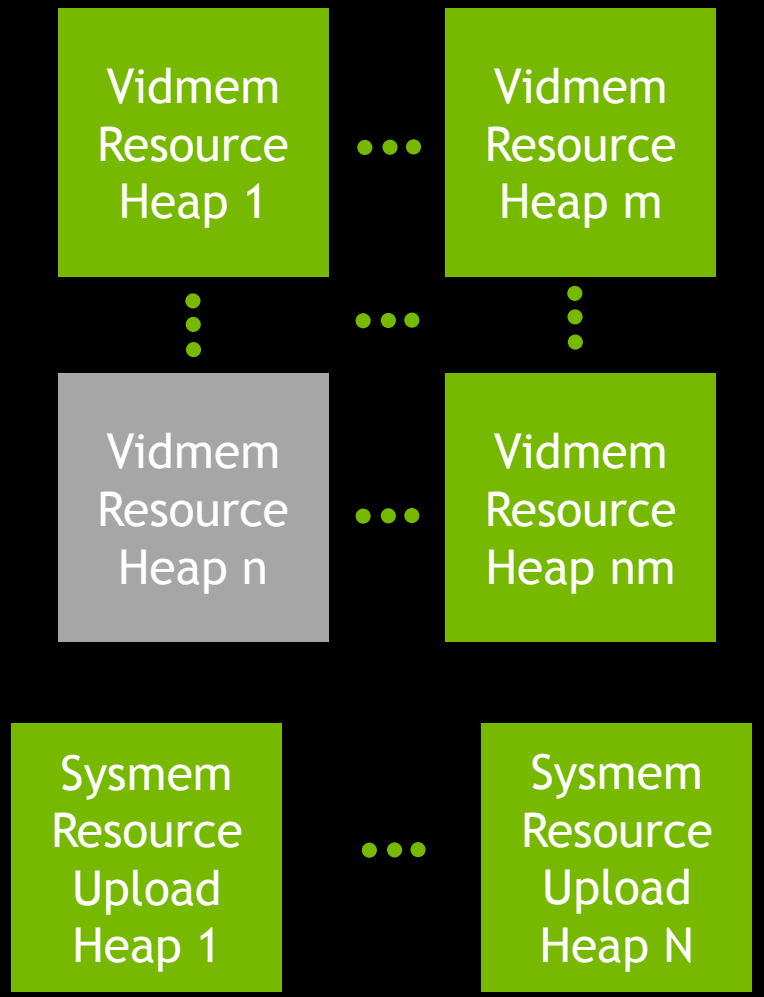
App detects that the next CL needs more additional texture vidmem than is currently available



Assume we got sysmem copies for all our textures in upload heaps

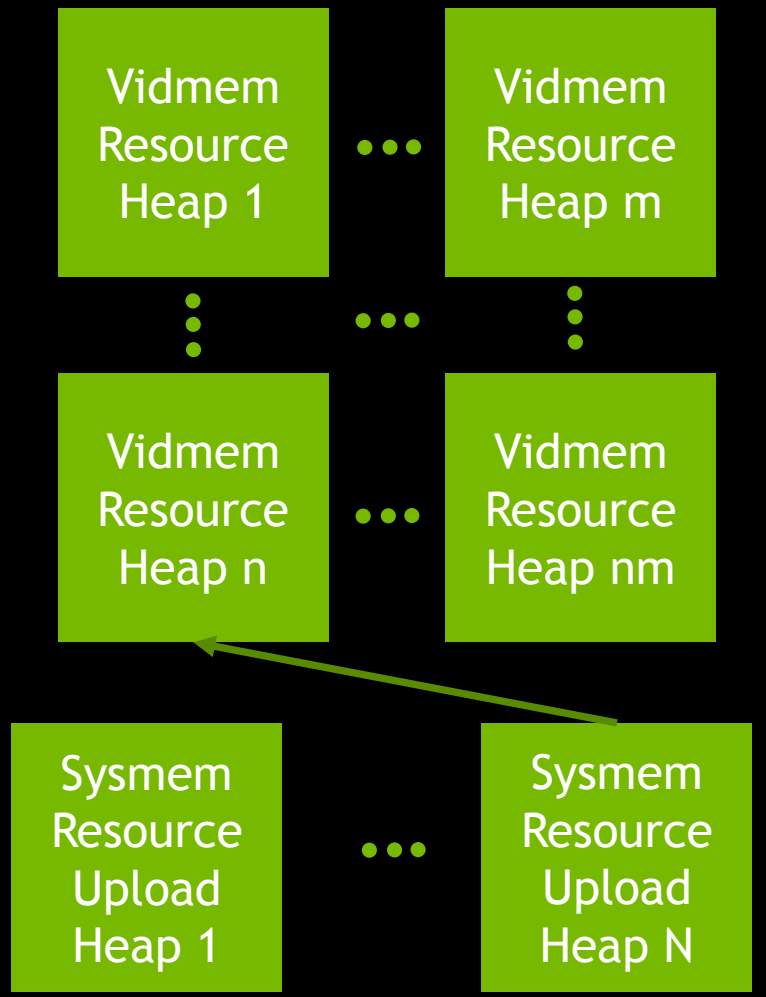
Handling Video Memory Over-commitment

‘Free’ old vidmem heap



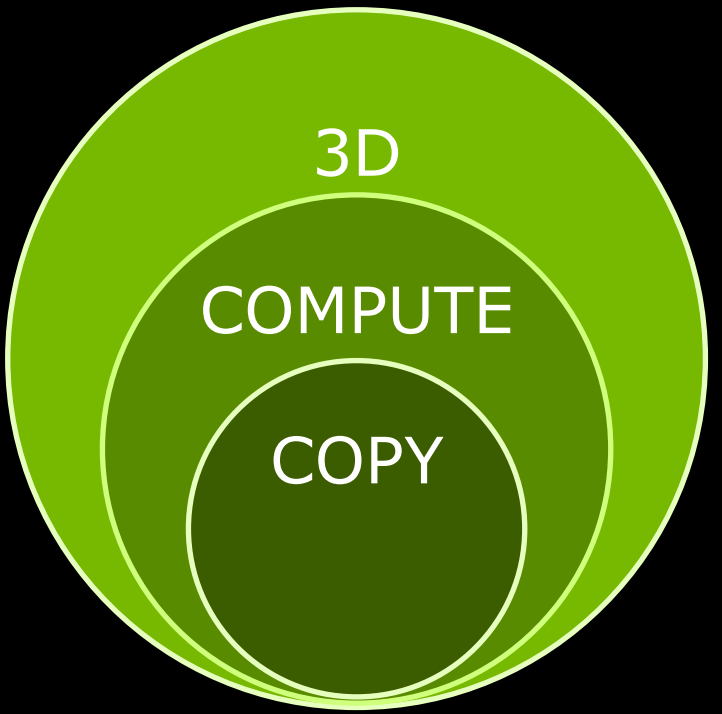
Handling Video Memory Over-commitment

Move data from sysmem copy heap of resource



Command Queues

- Use copy queues for async transfer operations
  - Especially important for MultiGPU transfers
- Use compute queues with care
  - Not all workloads pair up nicely
  - Remember IHV specific path for DX12!
  - Come and talk to us about getting this right



Agenda

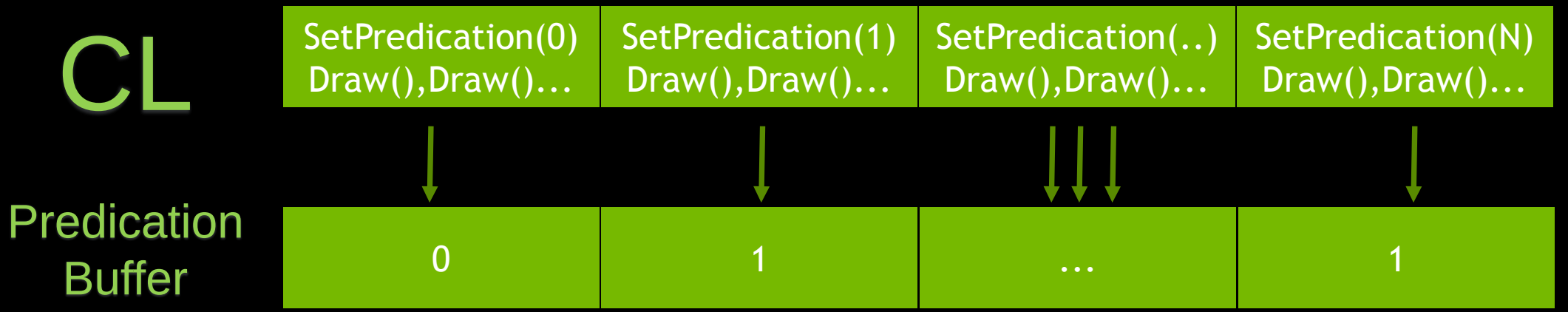
- DirectX 12: more control & responsibilities
- How to efficiently drive DirectX 12 on NVidia GPUs
- **New DirectX 12 programming model use cases**
- DirectX 12 & 11.1 new hardware feature use cases
- Q&A

New DirectX 12 programming model use cases

- Predication
  - Offers more flexibility than DirectX 11
- ExecuteIndirect
  - More powerful than DirectX 11 DrawIndirect() or DispatchIndirect()
- Explicit multi GPU support
  - Full control over where resources go and where execution happen

New DirectX 12 Predication Model

- Now fully decoupled from queries
- Predication on the value at a location in a buffer
- GPU reads buffer value when executing SetPredication



Just FYI : Calls that can be Predicated

DrawInstanced, DrawIndexedInstanced, Dispatch, CopyTextureRegion, CopyBufferRegion, CopyResource, CopyTiles, ResolveSubresource, ClearDepthStencilView, ClearRenderTargetView, ClearUnorderedAccessViewUint, ClearUnorderedAccessViewFloat, ExecuteIndirect

Usecase: Asynchronous CPU based occlusion

- CPU threads set 1: record command lists for objects

CL

SetPredication(0) DrawObj(0)	SetPredication(1) DrawObj(1)	SetPredication(..) DrawObj(..)	SetPredication(N) DrawObj(N)

- CPU threads set 2: perform software occlusion queries and fill in buf

Predication Buffer

0	1	…	1

- Excute the CL once the software occlusion is done

Execute Indirect (1/2)

- Execute several Draw, DrawIndexed or Dispatch calls in one go
  - It‘s more a MultiExecuteIndirect()
- Inbetween Draws/Dispatches:
  - Change Vertex and/or Index Buffer (also prim count)
  - Change root constants and root CBVs
  - Change root SRVs and UAVs

Execute Indirect API

```
void ExecuteIndirect(
ID3D12CommandSignature*  pCommandSignature,
UINT                     (Max)CommandCount,
ID3D12Resource*          pArgumentBuffer,
UINT64                   ArgumentBufferOffset,
ID3D12Resource*          pCountBuffer,
UINT64                   CountBufferOffset
);
```

Defines the operations to be carried out repeatedly

Max count of repetitions

Array of arguments that conform to the signature

Optional – buffer that overrides MaxCommandCount

Execute Indirect (2/2)

- Draw thousands of different objects in one ExecuteIndirect
  - Saves significant CPU time even for hundreds of objects
- Indirect compute work
  - For ideal perf use NULL counter buffer arg
- Graphics draw calls
  - For ideal perf keep counter buffer count ~= ArgMaxCount calls

Execute Indirect – Drawing Simulated Trees

- Imagine large set of physically simulated unique trees
  - Perhaps even broken up into tree parts by destruction
- For simplicity : All trees share the same texture atlas or texture array
- Each tree has a unique mesh and unique vertex and index buffer
  - This also means vertex count and topolgy are unique as well

Execute Indirect – Drawing Simulated Trees

DirectX 11

Solution 1:

```
foreach(tree)
      SetupMesh(VB,IB);
      DrawTree();
```

Slow – too many API call

Solution 2:

```
SetupMeshForAllTrees(VB,IB);
DrawTreesInstanced();
```

Needs to draw each tree with the same numbers of vertices/ topology for instancing to work

Execute Indirect – Drawing Simulated Trees

DirectX 12

CommandSignature

Argument Type	Data
VertexBufferView	VirtualAddressVB Size Stride
IndexBufferView	VirtualAddressIB Size Type
DrawIndexed	IndexCount InstanceCount StartIndexLocation BaseLocation StartInstanceLocation

Execute Indirect – Drawing Simulated Trees

DirectX 11

Solution 1:

```
foreach(tree)
      SetupMesh(VB,IB);
      DrawTree();
```

Slow – too many API call

Solution 2:

```
SetupMeshForAllTrees(VB,IB);
DrawTreesInstanced();
```

Needs to draw each tree with the same numbers of vertices/ topology for instancing to work

DirectX 12

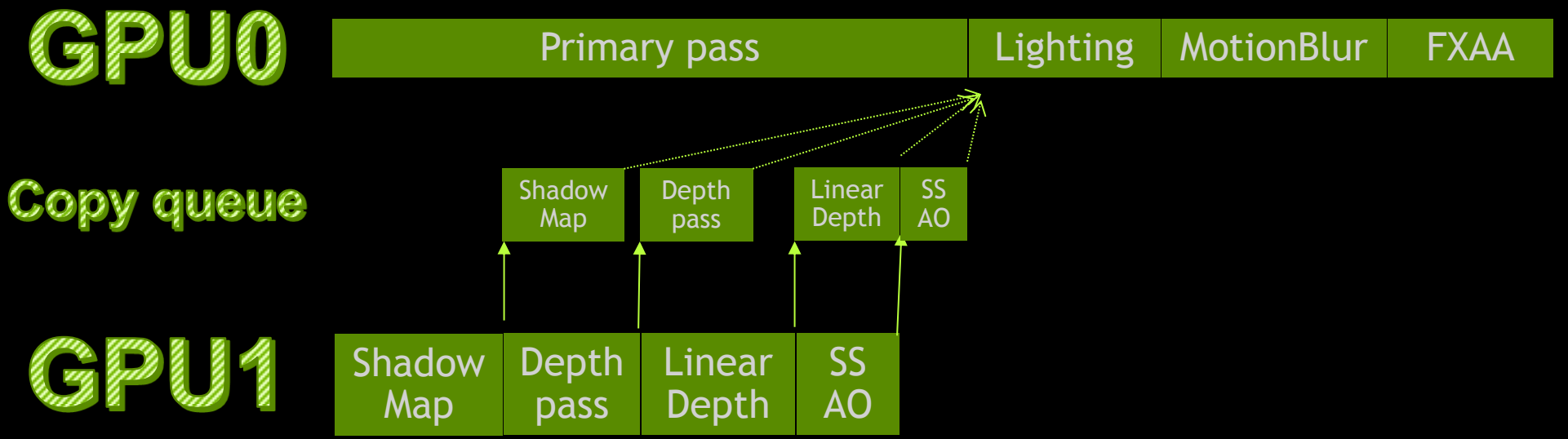
```
CreateTreeCommandSignature();

foreach(tree)
   appendDrawArgsAndVB(tree,argbuffer);

ExcuteIndirect(...,argbuffer,..)
```

One ExecuteIndirect() call efficiently draws all trees whilst using the right VB and IB using the optimal vertex count for the tree

Explicit multi GPU

- Finally full control over what goes on each GPU
- Create resources on specific GPUs
- Execute command lists on specific GPUs
- Explicitly copy resources between GPUs
  - Perfect usecase for DirectX 12 copy queues
- Distribute workloads between between GPUs
  - Not restricted to AFR

MultiGPU work distribution sample



Check Juha Sjöholms talk from the ‘Advanced Graphics Techniques Tutorial Day’ : ‚Explicit Multi GPU Programming with DirectX 12’

Agenda

- DirectX 12: more control & responsibilities
- How to efficiently drive DirectX 12 on NVidia GPUs
- New DirectX 12 programming model use cases
- **DirectX 12 & 11.1 new hardware feature use cases**
- Q&A

Conservative Raster

- Door opener to advanced AA techniques
- Enables the rasterizer to be used to do triangle binning
- See Jon Story‘s presentation:

  ‚Advanced Geometrically Correct Shadows for Modern Game Engines’

  directly after this talk!

DX12&11.1 FL3 hardware features use cases

- Volume Tiled Resources
  - Store sparse volumetric data
  - Run sparse volumetric simulations

see ‚Latency Resistant Sparse Fluid Simulation': [Alex Dunn, D3D Day – GDC 2015]

Q&A

Holger Gruen : hgruen@nvidia.com

	NVIDIA GeForce	
	6xx series and above	9xx series and above
Feature Level	11_0	12_1
Resource Binding	Tier 2	
Tiled Resources	Tier 1	Tier 3
Typed UAV Loads	No	Yes
Conservative Rasterization	No	Tier 1
Rasterizer-Ordered Views	No	Yes
Stencil Reference Output	No	
UAV Slots	64	
Resource Heap	Tier 1	