Tiled shading: light culling – reaching the speed of light

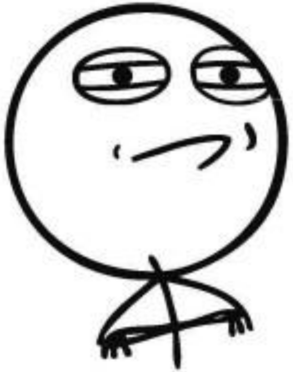
Dmitry Zhdan
Developer Technology Engineer, NVIDIA

Agenda

- Über Goal
- Classic deferred vs tiled shading
- How to improve culling in tiled shading?
- New culling method overview
- Cool results!

Über Goal

Improve overall lighting performance in tiled shading

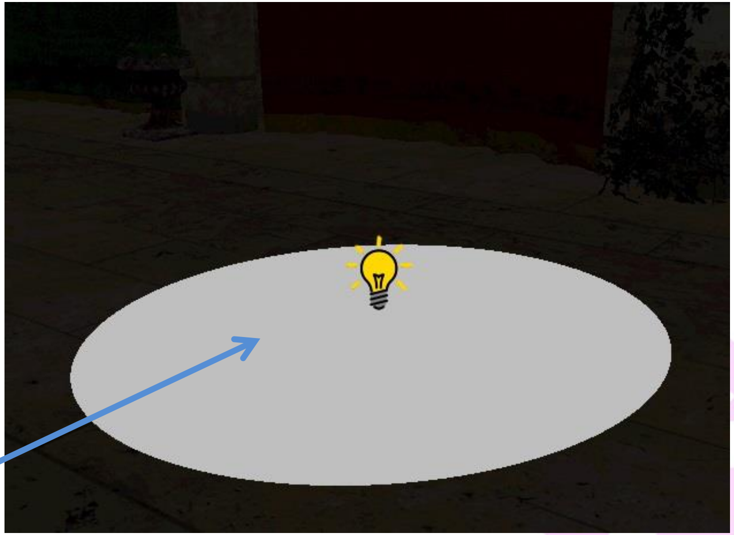


Takeaway

You'll know how to speed up light culling in 10x times and more!

Classic deferred: overview

- For each light:
  - Render proxy geometry to mark pixels inside the light volume



Pixels where light will be processed

Classic deferred: overview

- For each light:
  - Render proxy geometry to mark pixels inside the light volume
  - Shade only marked pixels
  - Blend to output

Classic deferred: pros and cons

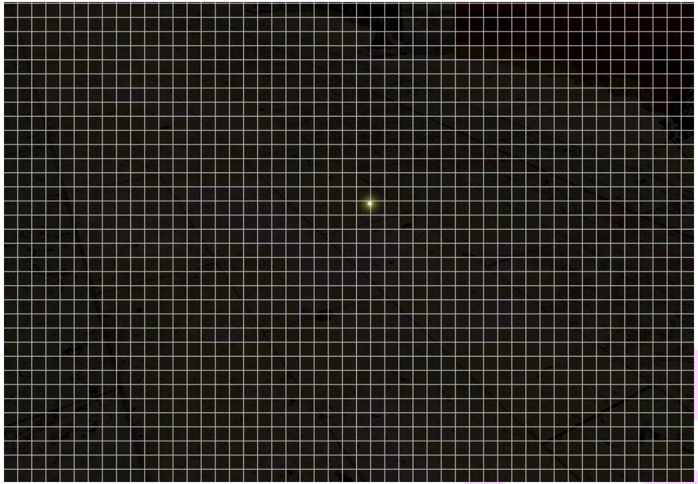
- **Pros ☺**
  - Precise per-pixel light culling
  - A lot of work is done outside of the shader
- **Cons ☹**
  - Lighting is likely to become bandwidth limited
  - Culling is ROP limited

What we want to avoid?

- Blending
- G-buffer data reloading
- Per light state switching

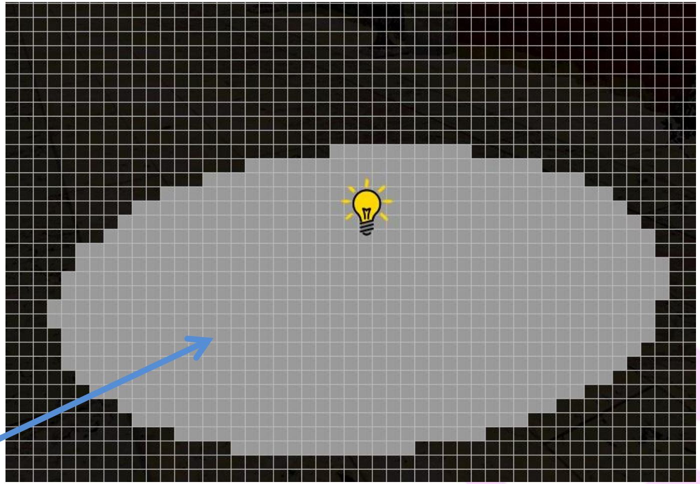
Tiled shading: overview

- Divide screen into tiles
- For each tile:
  - Find min-max z



Tiled shading: overview

- Divide screen into tiles
- For each tile:
  - Find min-max z
  - Cull light sources against tile frustum



Tiles where light will be processed

Tiled shading: overview

- Divide screen into tiles
- For each tile:
  - Find min-max z
  - Cull light sources against tile frustum
  - Shade tile using given light list

Tiled shading: pros and cons

- **Pros ☺**
  - Lighting phase takes all visible lights in one go
- **Cons ☹**
  - Less accurate culling with tile granularity
  - Frustum-primitive tests are either too coarse or too slow

Why care about culling?

- Culling itself can be a costly operation
- Accurate culling speeds up lighting

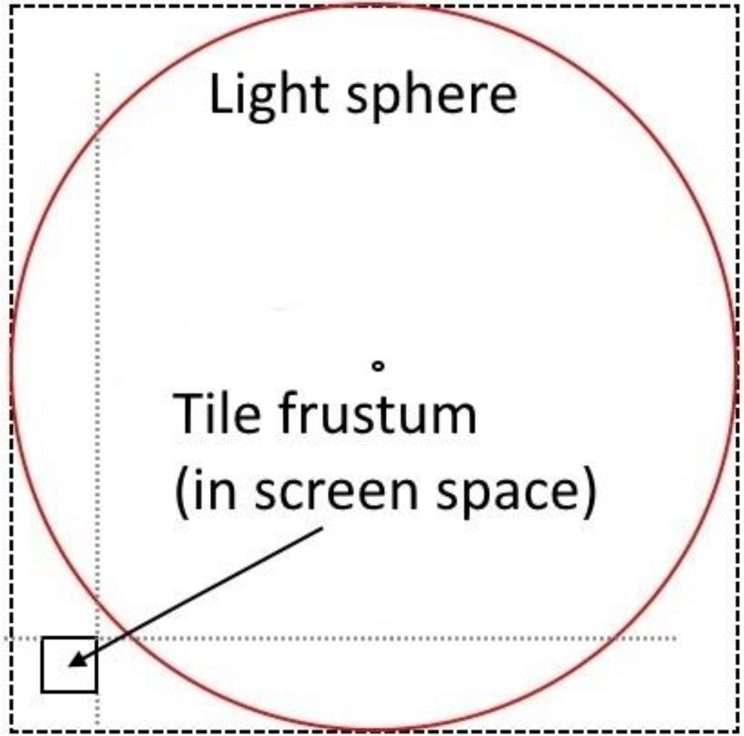
Adding “false positives” can dramatically reduce lighting performance!

Culling challenges

- Minimize the number of “false positive” lights obtained in culling phase
- Improve light culling performance in tiled shading rendering

Sphere vs frustum planes: never ever!

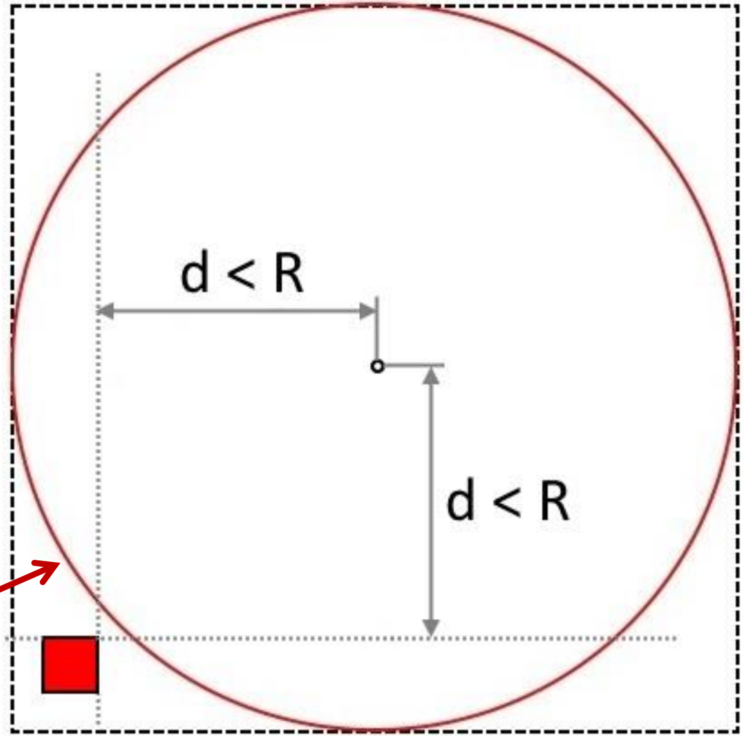
- Most commonly used test
- In fact, it is a frustum-box test
- Extremely inaccurate with large spheres



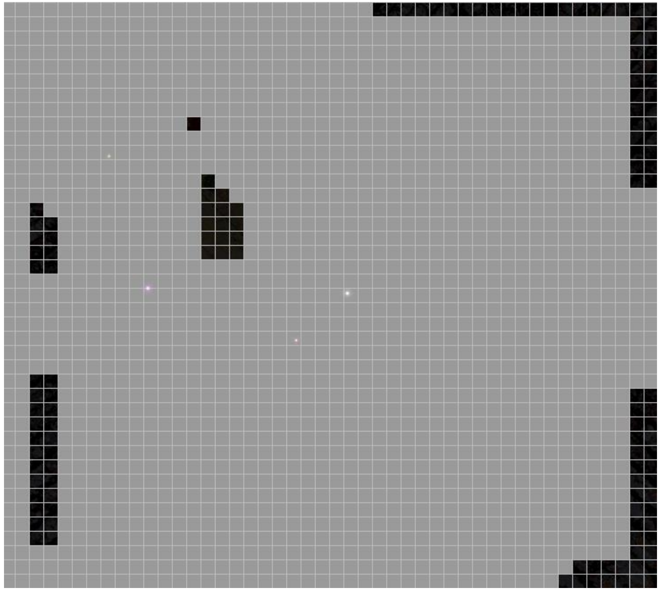
Sphere vs frustum planes: never ever!

- Most commonly used test
- In fact, it is a frustum-box test
- Extremely inaccurate with large spheres

False positive ☹

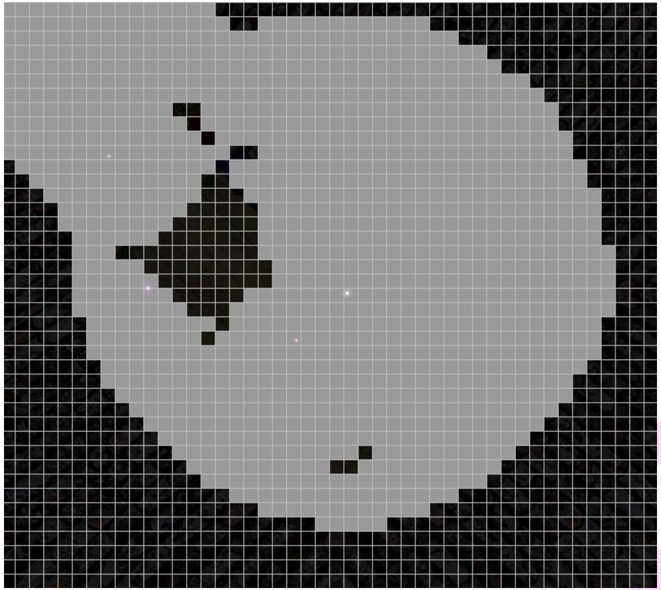


Frustum planes



No ☹

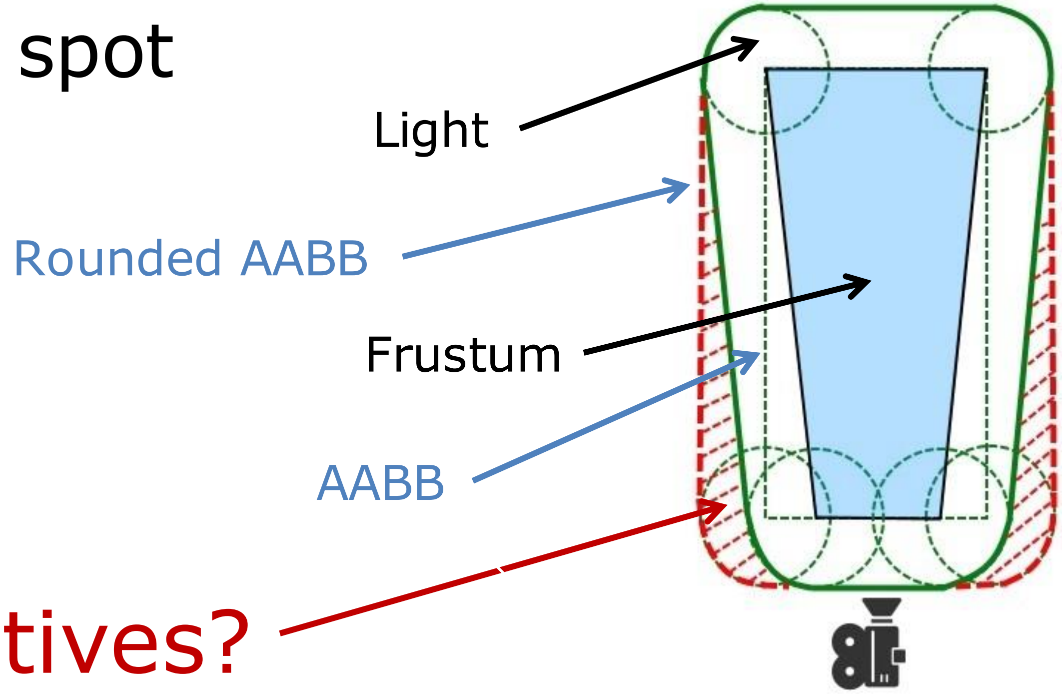
Reference



Does “is point inside volume” test for each pixel in a tile

Rounded AABB isn’t an option too…

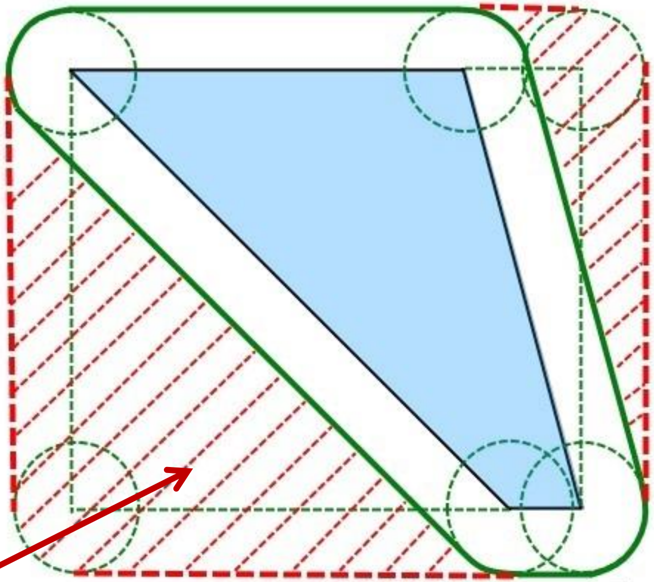
- Doesn’t suit for spot lights!

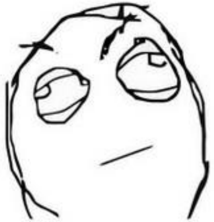


False positives?

Rounded AABB isn't an option too…

- Doesn't suit for spot lights!
- Works badly for very long frustums

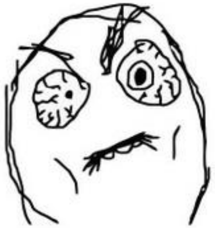




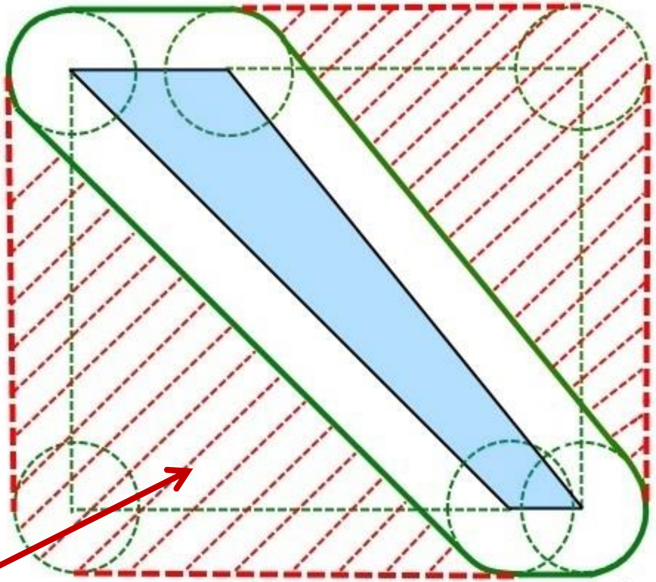
False positives

Rounded AABB isn't an option too…

- Doesn't suit for spot lights!
- Works badly for very long frustums
- Problematic for wide FOV



False positives!



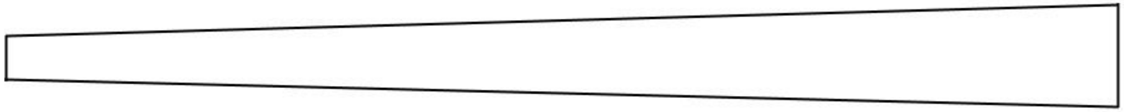
Can we get away from frustums?

- Average tile frustum angle is small:

  FOV = 100^0, Tile size = 16x16 pixels

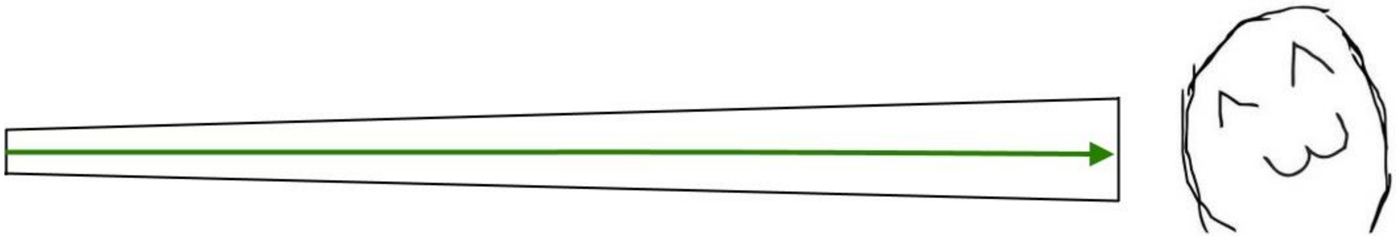
  Angle = FOV • (tile_size / screen_height) = **$\mathbf{0.8^0}$** (at 1080p)

This one is only 2.5^0



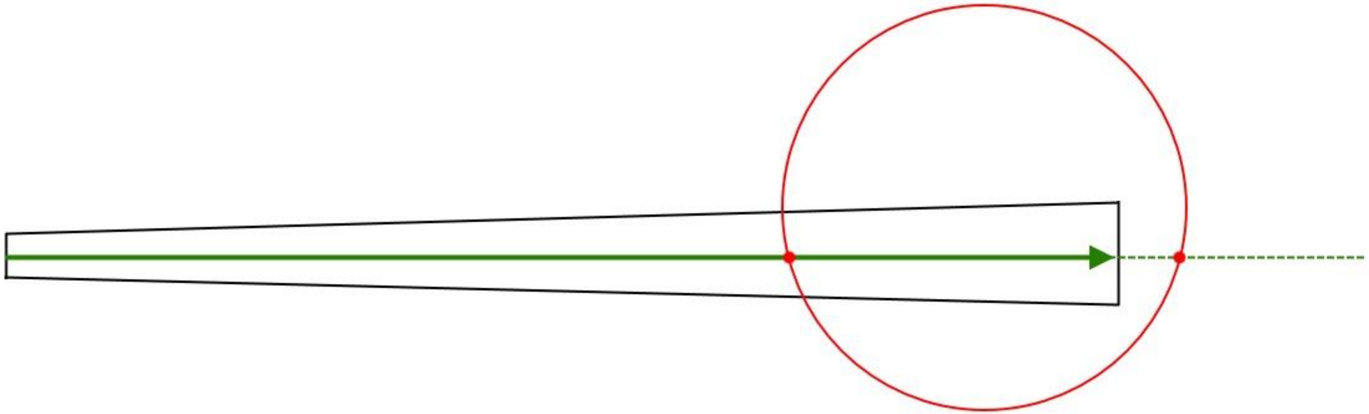
Can we get away from frustums?

- Frustum can be represented as a single ray at tile center
  - Or 4 rays at tile corners



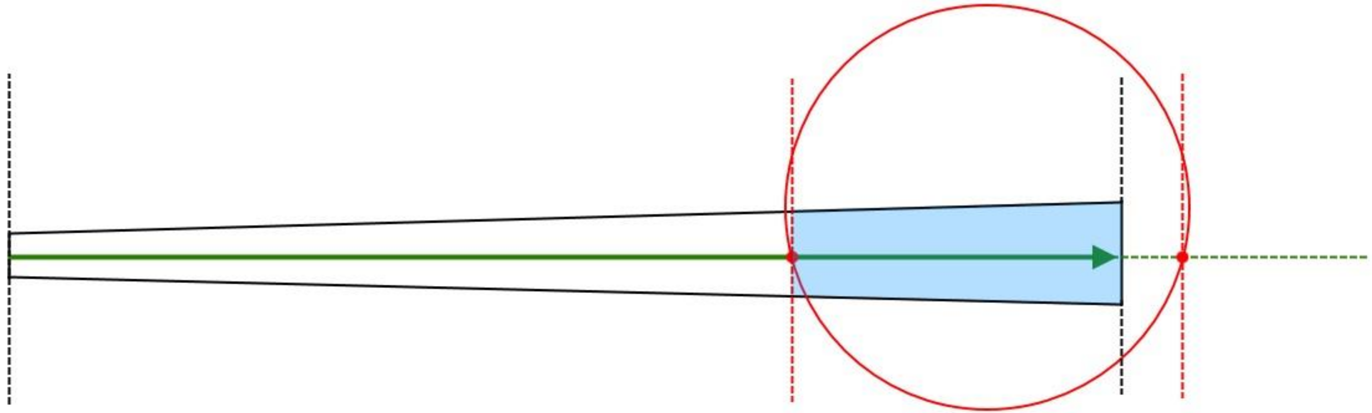
How to improve culling accuracy?

- Replace frustum test with ray intersection test:
    - Ray-sphere, ray-cone, …



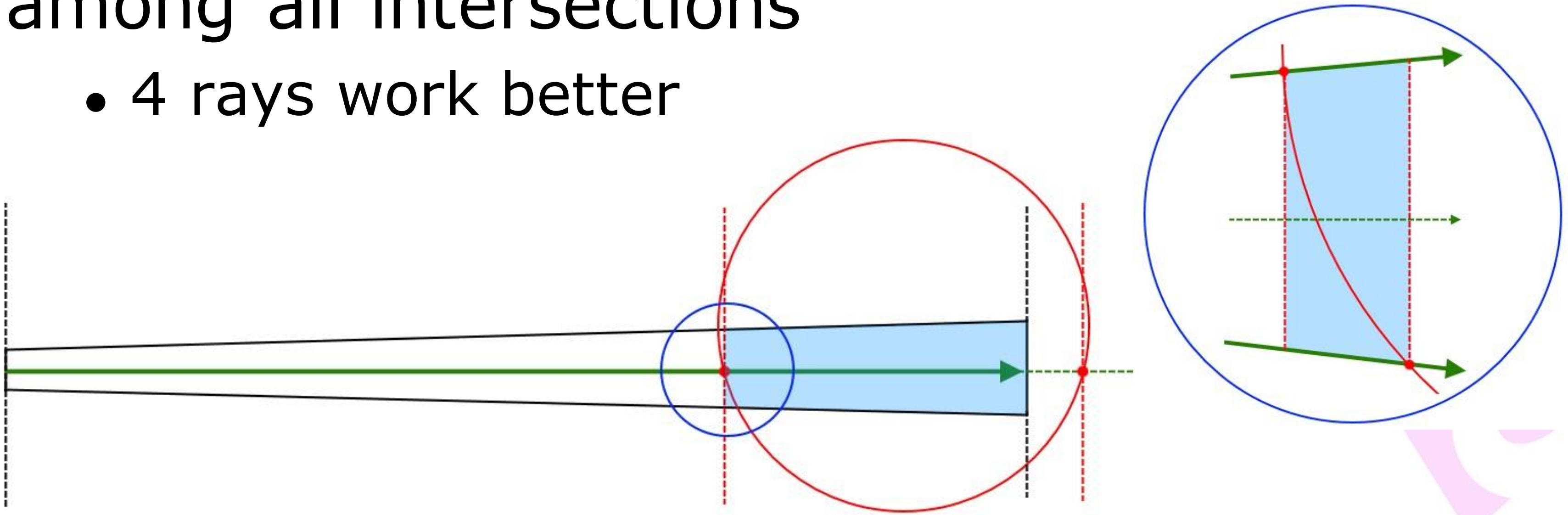
How to improve culling accuracy?

- Compare tile min-max z with min-max among all intersections

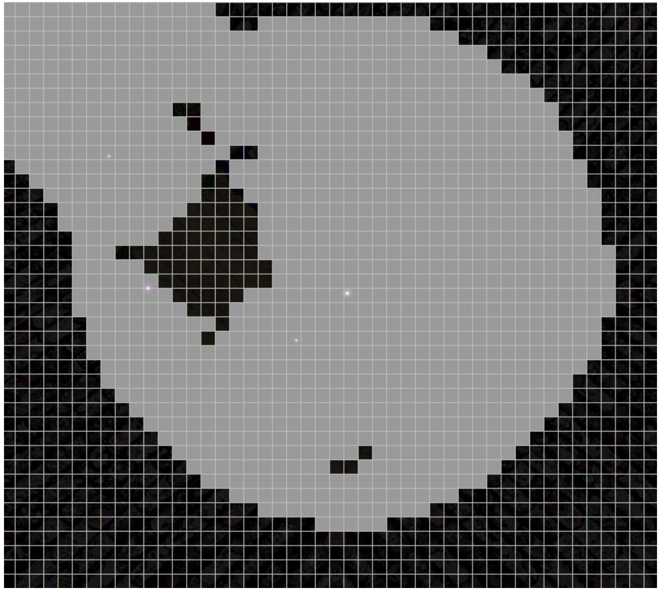


How to improve culling accuracy?

- Compare tile min-max z with min-max among all intersections
    - 4 rays work better

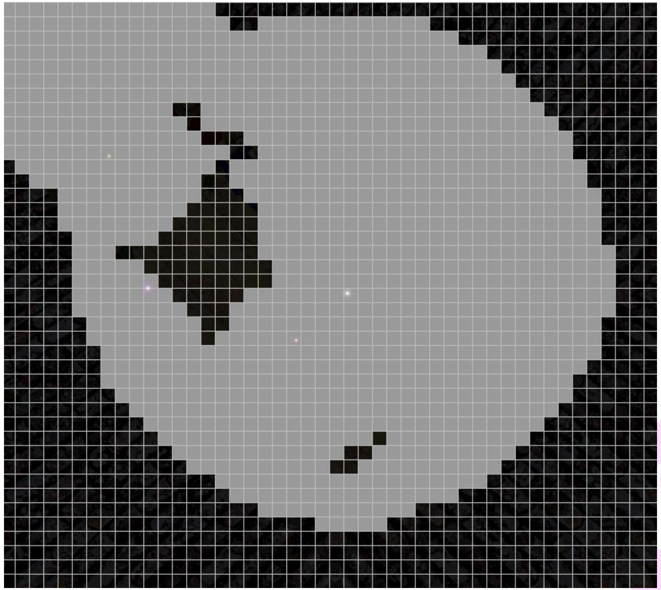


Ray-primitive



Yes ☺

Reference



But culling on compute sucks

- It is a straightforward enumeration ☹

  total operations = X • Y • N

  X – tile grid width

  Y – tile grid height

  N – number of lights

How to improve culling performance?

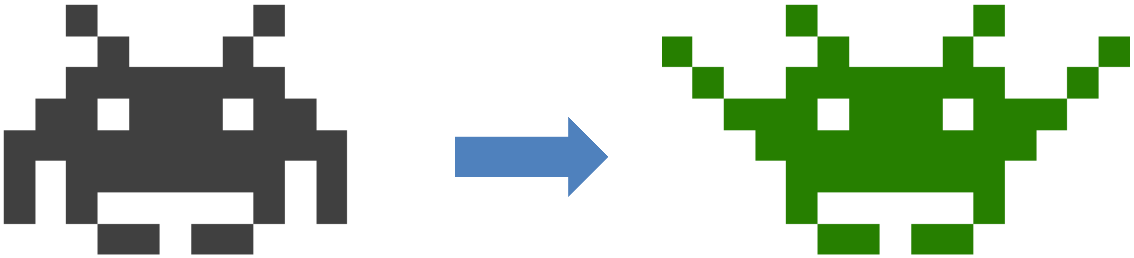
- Reduce the order of enumeration
  - Subdivide screen into 4-8 sub-screens
  - Coarsely cull lights against sub-screen frustums
  - Select corresponding sub-screen during culling phase
- Up to 2x boost with small lights, but we want more!

How to improve culling performance?

- We are limited by the compute power ☹
- Let’s try to offload some work from shader to special HW units!

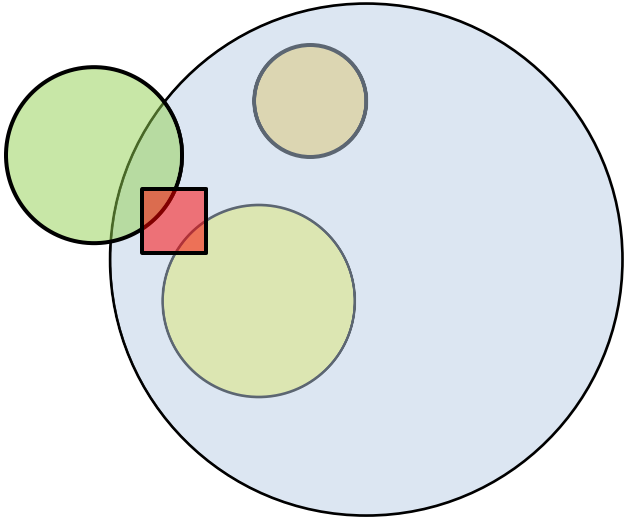
How to improve culling performance?

- Let's switch from compute to graphics pipeline! Like in the good old times! ☺



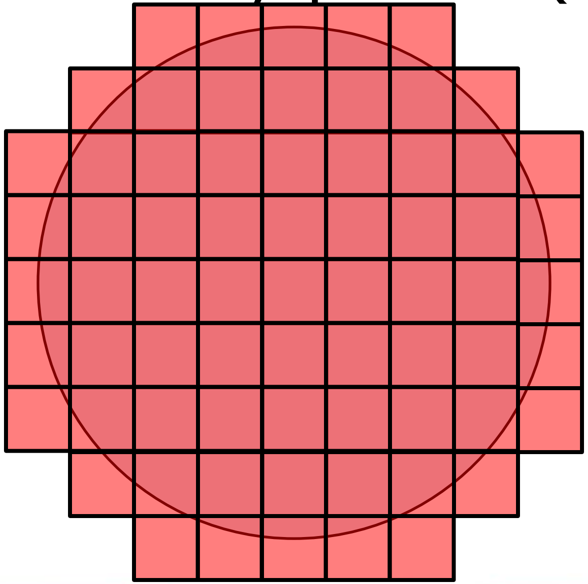
Take the best from classic and tiled!

- Migrate from compute idiom:
  - “one tile - many lights”



Take the best from classic and tiled!

- To classic deferred idiom:
  - "one light - many pixels" (1 pixel = 1 tile)



Light culling using graphics

- Use rasterizer to generate light fragments
  - Empty tiles will be natively skipped
- Use depth test to account for occlusion
  - Useless work for occluded tiles will be skipped
- Use primitive-ray intersection in PS for fine culling and light list updating

The Idea: overview

- Culling phase tile → 1 pixel
- Light volume → proxy geometry
- Coarse XY-culling → rasterization
- Coarse Z-culling → depth test
- Precise culling → pixel shader

How to integrate?

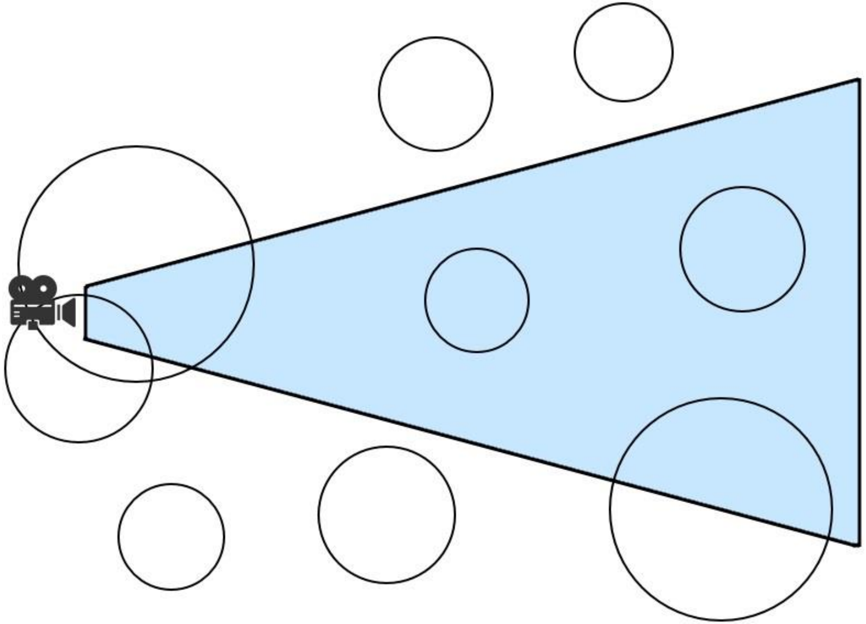
- Don’t use über shaders
- Always break tiled shading into 3 phases:
    - Reduction
    - Culling → **new method**
    - Lighting

New Culling: Bird's-eye view

- Camera frustum culling
- Depth buffers creation
- Rasterization & classification

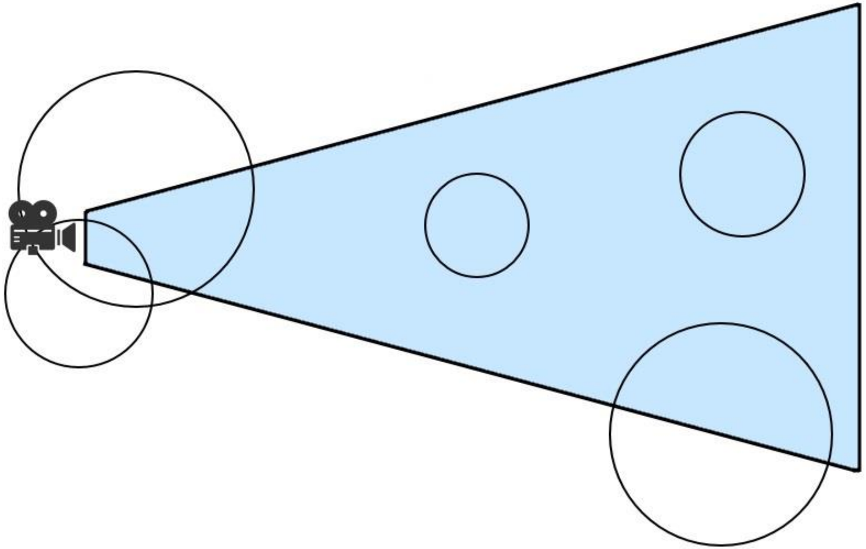
Step 1: Camera frustum culling

- Cull lights against camera frustum



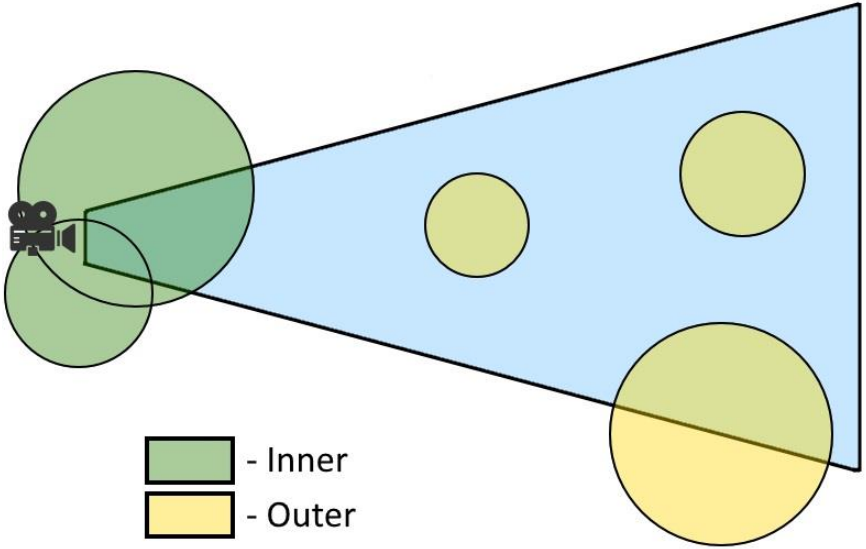
Step 1: Camera frustum culling

- Cull lights against camera frustum



Step 1: Camera frustum culling

- Cull lights against camera frustum
- Split visible lights into "outer" and "inner"



Step 2: Depth buffers creation

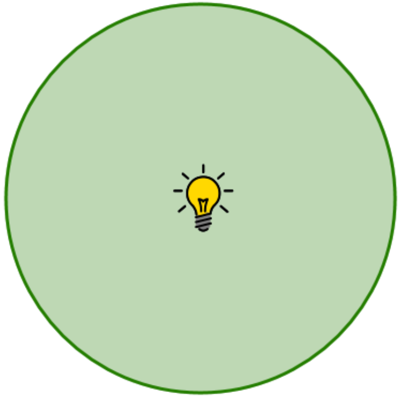
- For each tile:
  - Find and copy max depth for “outer” lights
  - Find and copy min depth for “inner” lights

- Depth test is a key to high performance!
  - Use [earlydepthstencil] in shader

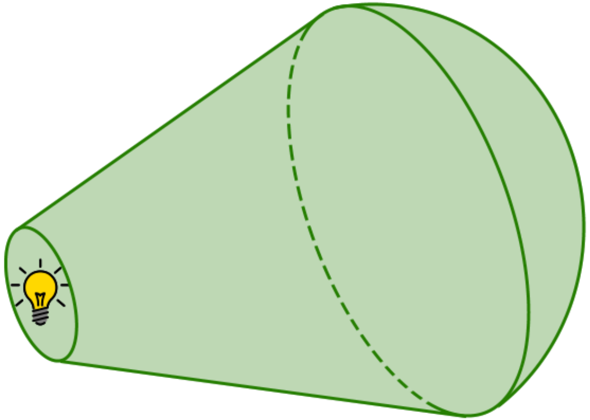
Step 3: Rasterization & Classification

- Render light geometry with depth test
  - “outer” – max depth buffer
    - Front faces with direct depth test
  - “inner” - min depth buffer
    - Back faces with inverted depth test
- Use PS for precise culling and per-tile light list creation

Common light types



Point light (omni)

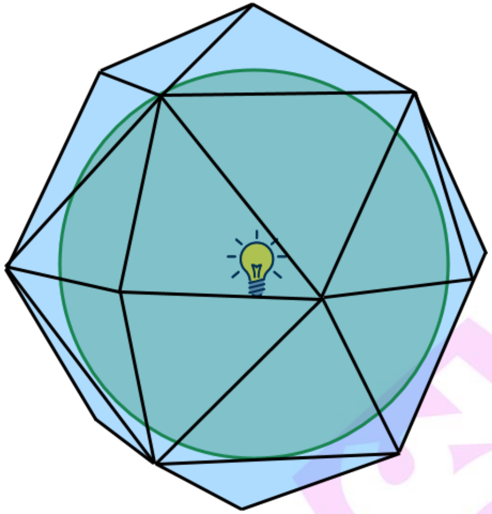


Directional light (spot)

Light geometry can be replaced with proxy geometry

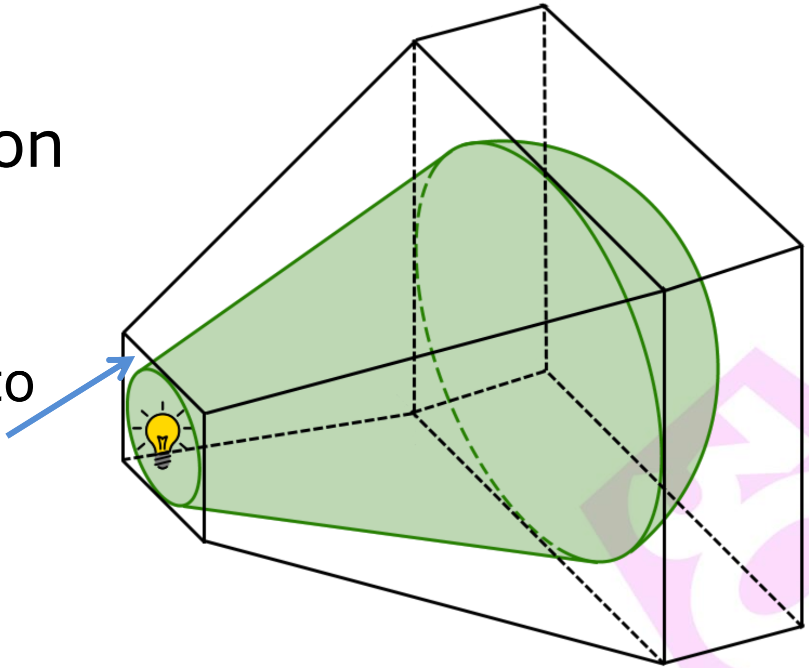
Proxy geometry for point lights

- Geosphere (2 subdivisions, octa-based)
- Close enough to sphere
  - Low poly works well at low resolution
  - Equilateral triangles can ease rasterizer's life



Proxy geometry for spot lights

- Why so simple?
  - Easy for parametrization
    - From a searchlight
    - To a hemisphere
    - Plane part can be used to handle area lights



Light culling via rasterization

- **Advantages ☺**
  - No work for tiles without lights and for occluded lights
  - Coarse culling is almost free!
  - Incredible speed up with small lights
  - Complex proxy models can be used!
  - Mathematically it is a branch-and-bound procedure!

Culling perf: long-ranged lights

GPU	CS, ms	Raster, ms	Boost
GTX 970 - 19x12	0.55	0.15	**x4**
R9 390 - 19x12	0.60	0.25	**x3**
GTX 970 - 4K	2.00	0.35	**x6**
R9 390 - 4K	2.15	0.65	**x3**

400 lights (200 omnis, 200 spots)
20 lights per tile on average
CS: ray-primitive based (same culling precision as using raster)

Culling perf: medium-ranged lights

GPU	CS, ms	Raster, ms	Boost
GTX 970 - 19x12	7.30	0.45	**x17**
R9 390 - 19x12	6.90	0.45	**x15**
GTX 970 - 4K	25.35	1.10	**x23**
R9 390 - 4K	23.75	1.30	**x18**

10000 lights (5000 omnis, 5000 spots)
70 lights per tile on average
CS: ray-primitive based (same culling precision as using raster)

Culling perf: fast CS vs Raster

GPU	CS fast, ms	Raster, ms	Boost
GTX 970 - 19x12	1.60	0.45	**x3.5**
R9 390 - 19x12	1.30	0.45	**x3.0**
GTX 970 - 4K	5.45	1.10	**x5.0**
R9 390 - 4K	4.55	1.30	**x3.5**

10000 lights (5000 omnis, 5000 spots)
70 lights per tile on average
CS fast: rounded AABB, sub-screens partitioning (less accurate culling)

Lighting perf: accurate vs fast culling

GPU	Fast, ms	Accurate, ms	Boost
GTX 970 - 19x12	6.50	4.85	**25%**
R9 390 - 19x12	3.55	2.75	**22%**
GTX 970 - 4K	22.20	16.45	**26%**
R9 390 - 4K	12.00	9.25	**23%**

10000 lights (5000 omnis, 5000 spots)
70 lights per tile on average
Fast: CS with rounded AABB, sub-screens partitioning
Accurate: fine CS or raster

Culling perf: HD vs 4K

GPU	HD (ms)	4K (ms)	4K / HD
GTX 970 – CSopt	1.45	5.45	**3.8**
GTX 970 - Raster	0.40	1.10	**2.7**
R9 390 - CSopt	1.15	4.55	**4.0**
R9 390 - Raster	0.40	1.30	**3.2**

Raster leads to less performance drop compared with optimized CS version at 4K

Culling via rasterization: conclusion

- ☺ 3x-20x times faster than the same CS version
- ☺ Produces less “false-positives” at a small cost
- ☺ Has better resolution scaling
- ☺ Raster allows us to use complex light volumes

References

- "Advancements in Tiled-Based Compute Rendering" – GDC 2015, Gareth Thomas
- "Parallel Graphics in Frostbite –Current & Future" - SIGGRAPH 2009, Johan Andersson
- Jim Arvo, "A simple method for box-sphere intersection testing", Graphics Gems 1990

Thanks!

dzhdan@nvidia.com

Bonus slides

But the devil is in the details…

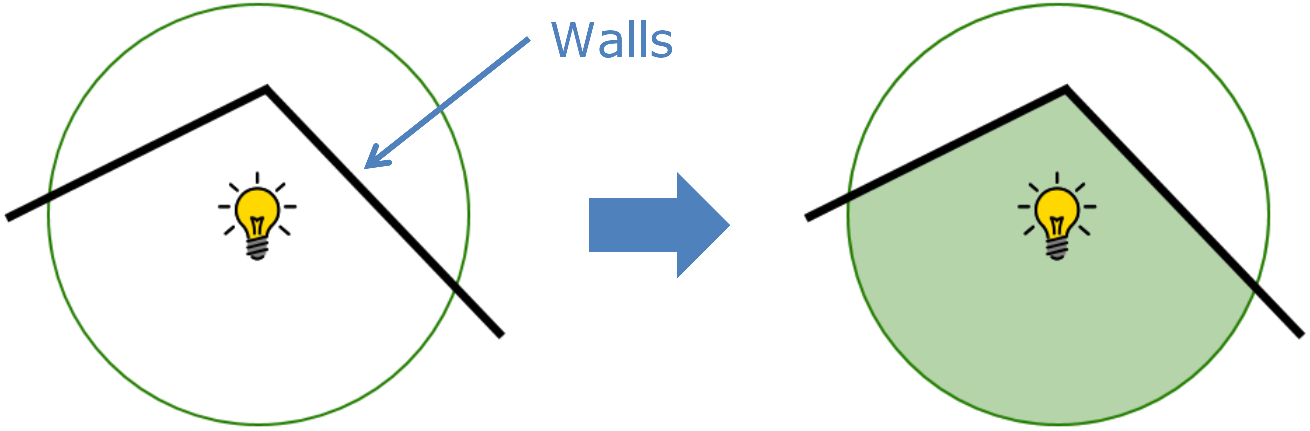
BONUS SLIDES!

Camera frustum culling

- Suits well for CPU
- It is always better to not only compute index list of visible lights but tightly pack light data too!
    - Better cache locality
    - Boosts culling and lighting phases

Proxy geometry ideas

- We can integrate clip planes into proxy models to avoid light leaking

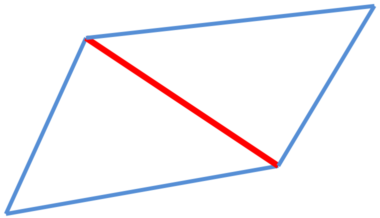


Proxy geometry ideas

- We can use even coarse shadow volumes to avoid lighting in shadows! ☺

Rasterization tips

- Conservative raster is not applicable here!
  - Fragments on shared edges will be added twice, thus light will be added twice at some tiles



- Enlarge geometry in VS instead!

Omni rasterization tips

- Reproject half tile size back to view space
- Use closest to the camera value for reprojection:
    - z = light_view.z – light_range
- Add it to light range

Spot rasterization tips

- Reproject half tile size back to view space
- Use closest to the camera z value for reprojection
- Enlarge geometry in all directions!
  - This is why plane part in the spot proxy is important

Explicit Multi GPU Programming with DirectX 12

Juha Sjöholm
Developer Technology Engineer
NVIDIA

Agenda

- What is explicit Multi GPU
- API Introduction
- Engine Requirements
- Frame Pipelining – Case Study

Problem With Implicit Multi GPU

Ideal situation

- Driver does its magic
- Developer doesn‘t have to care
- It just works

Reality

- Driver needs lots of hints
  - Clears, discards
  - Vendor specific APIs
- Developer needs to understand what driver is trying to do
- It still doesn’t always fly

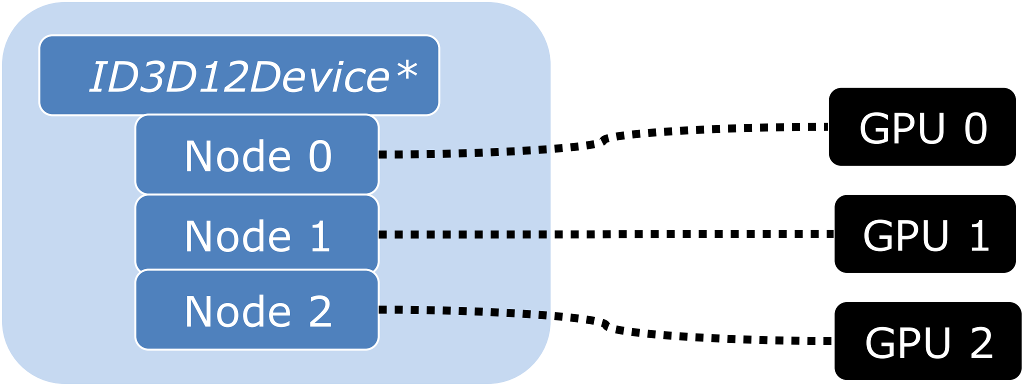
What is Explicit Multi-GPU?

- Control cross GPU transfers
  - No unintended implicit transfers
- Control what work is done on each GPU
- Not just Alternate Frame Rendering (AFR)

DX12 Explicit Multi GPU

- No more driver magic
- There is no driver level support for AFR
- Now you can do it better yourself, and much more!
- No vendor specific APIs needed

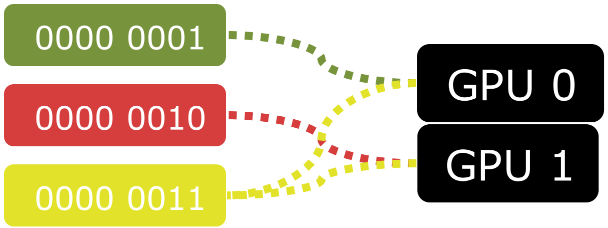
Adapters – Linked Node Adapter



Adapters – Multiple Adapters

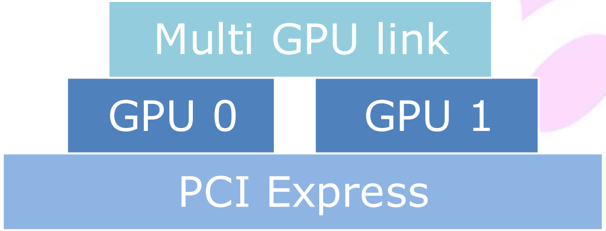
Linked Node Adapter

- When user has enabled use of multiple GPUs in display driver, linked node mode is enabled
  - *IDXGIFactory::EnumAdapters1()* sees one adapter
  - *ID3D12Device::GetNodeCount()* tells node count
- Nodes (GPUs) are referenced with affinity masks
  - Node 0 = 0x1
  - Node 1 = 0x2
  - Node 1 and 2 = 0x3



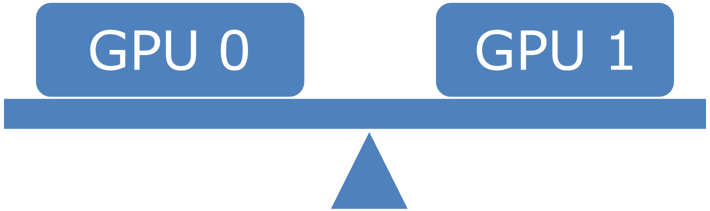
Linked Node Features

- Resource copies directly from discrete GPU to discrete GPU – not through system memory
- Special support for AFR *IDXGISwapChain3::ResizeBuffers1()* allows utilization of other connections than PCIe when presenting frames
- Good for multiple discrete GPUs!



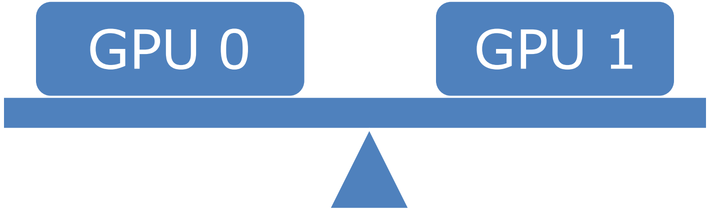
Linked Node Load Balancing

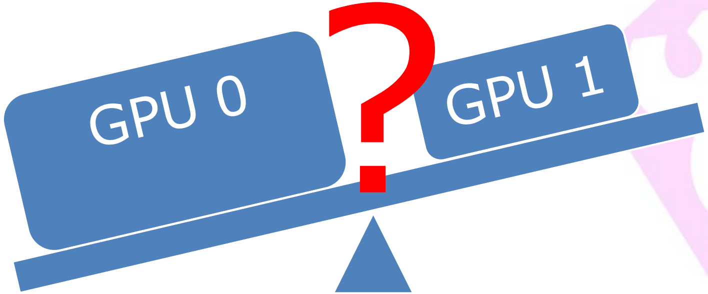
- It’s safe to assume that nodes are balanced for foreseeable future
    - Life is easy



Linked Node Load Balancing

- It's safe to assume that nodes are balanced for foreseeable future
    - Life is easy
- Heterogeneous nodes may be available some day





Infrastructure For Explicit M-GPU

- Renderer has to be aware of multiple GPUs
  - Expose multiple GPUs at right level
  - Wrap command queues, resources, descriptors, gpu virtual addresses etc. for multiple GPUs
- This can actually be the part that requires most effort
  - Once infrastructure exists, it’s easier to experiment

Multi Node APIs

- With linked nodes, some things are very easy
- Some interfaces are omni node (no node mask)
  - Starting with *ID3D12Device*
- Some interfaces are multi node
  - Affinity mask can have more than one bit set
  - Root signatures, pipeline states and command signatures can be often just shared for all nodes

*ID3D12RootSignature**
NodeMask 0x3

*ID3D12PipelineState**
NodeMask 0x3

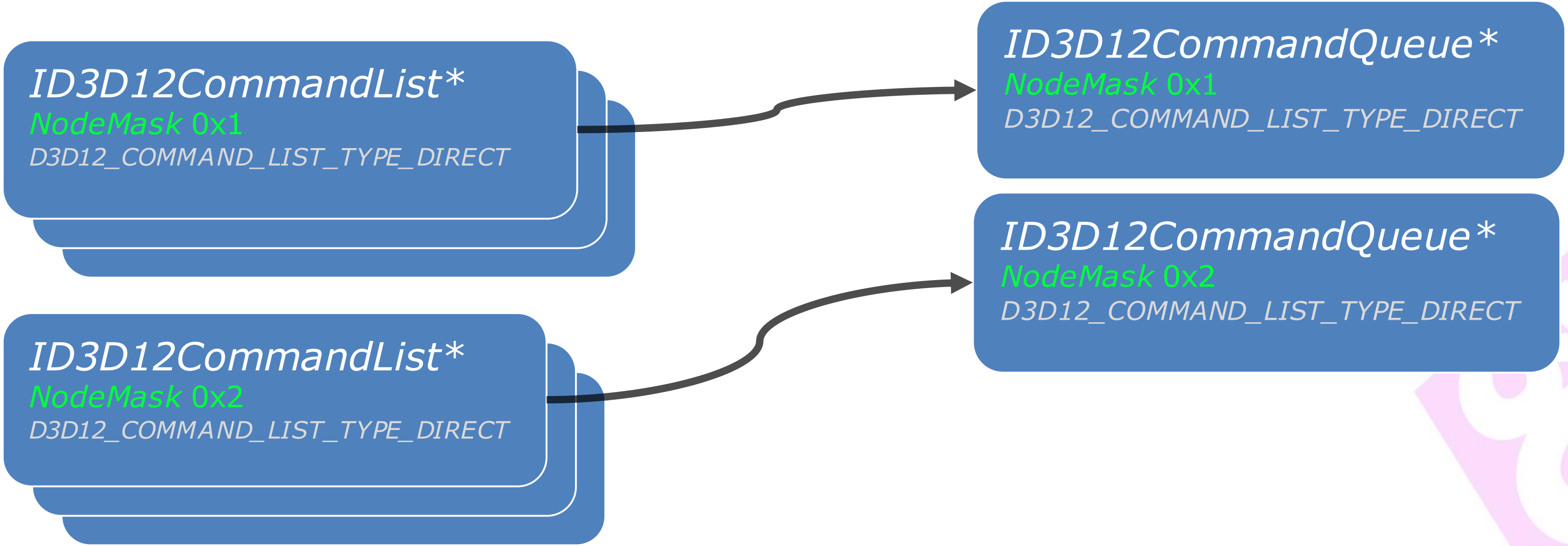
*ID3D12CommandSignature**
NodeMask 0x3

Command Queues And Lists

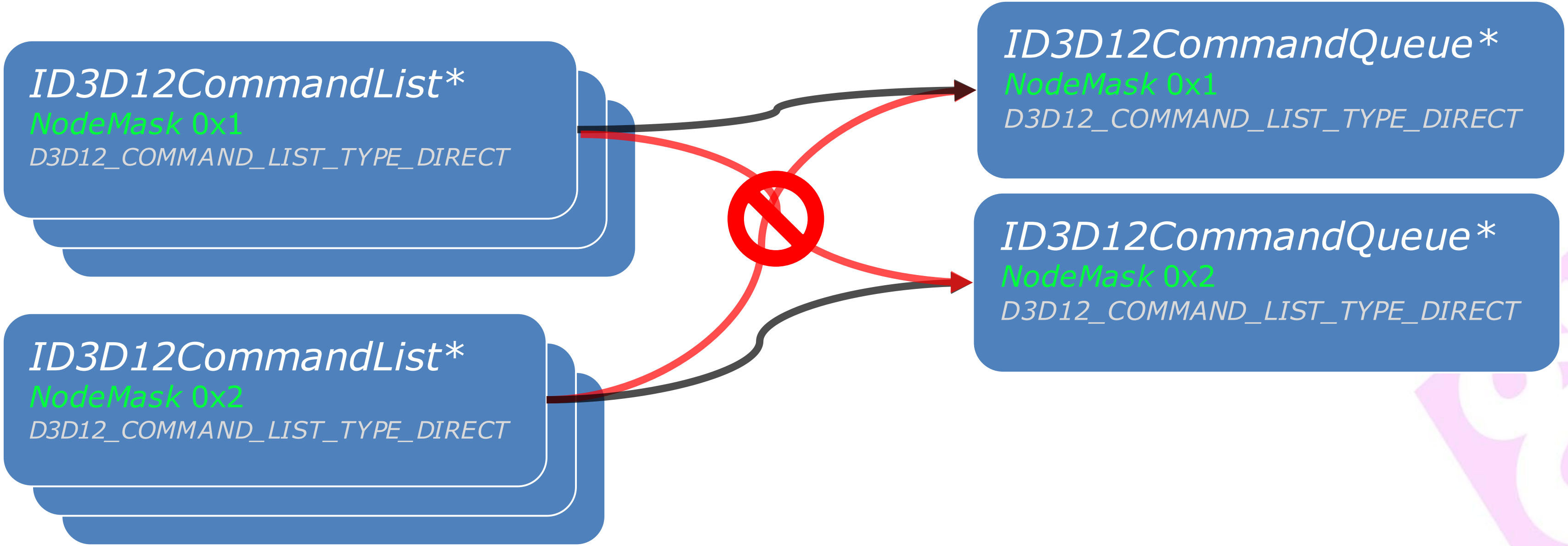
*ID3D12CommandQueue**
NodeMask 0x1
D3D12_COMMAND_LIST_TYPE_DIRECT

- Each node has its own *ID3D12CommandQueue, i.e. "engine"*
- *ID3D12CommandLists* are also exclusive to single node
  - Command list pooling for each node is needed

Command List Pooling



Command List Pooling

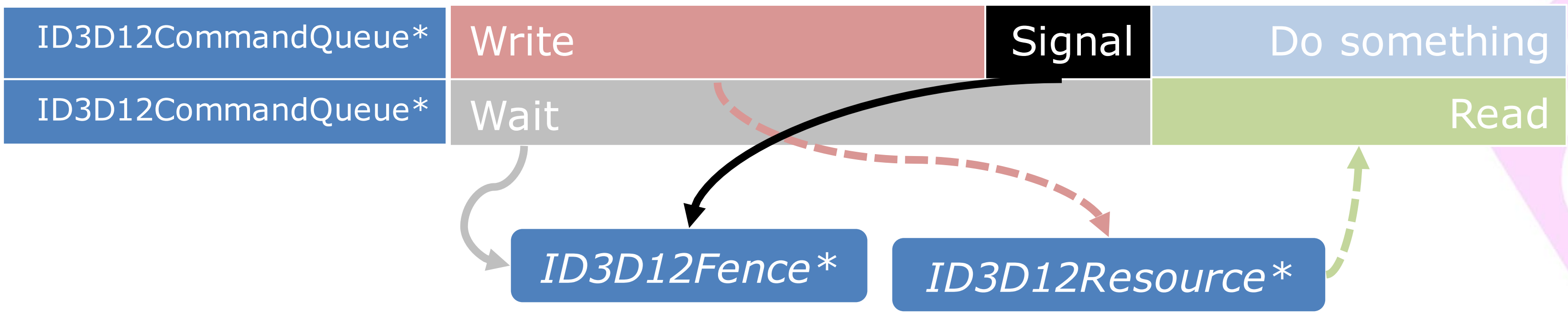


Synchronization - Fences

- Different command queues need to be synchronized when sharing resources
- *ID3D12Fence* is the synchronization tool

Fences

- Application must avoid access conflicts
- Application must ensure that all engines see shared resources in same state



Copy Engine(s)

- *ID3D12CommandQueue* with *D3D12_COMMAND_LIST_TYPE_COPY*
- Cross GPU copies *parallel* to other processing
- Remember to double buffer the resources

GPU 1	Graphics	Frame 0	Frame 1	Frame 2	Frame 3	Frame 4	Frame 5				
	Copy	*Idle*	F0	*Idle*	F1	*Idle*	F2	*Idle*	F3	*Idle*	F4
GPU 0	Graphics	(F-2)	(F-1)	F0	F1	F2	F3	F4			

Cross Node Sharing Tiers

- *ID3D12Device* has tiers for cross node sharing
- Tier 1 supports only cross node copy operations
  - *ID3D12GraphicsCommandList::CopyResource()* etc
- Tier 2 supports cross node SRV/CBV/UAV access

- While SRV/CBV/UAV access may seem convenient, try whether using parallel copy engines would be more efficient

Resources

- Resources and descriptors need most attention
- Resources/heaps have two separate node masks
  - *CreationNodeMask* is single node mask
  - *VisibleNodeMask* is multi node mask
- Descriptor heap is exclusive to single node

Resources - Visibility

Node 0x1 memory

*ID3D12DescriptorHeap**
NodeMask 0x1

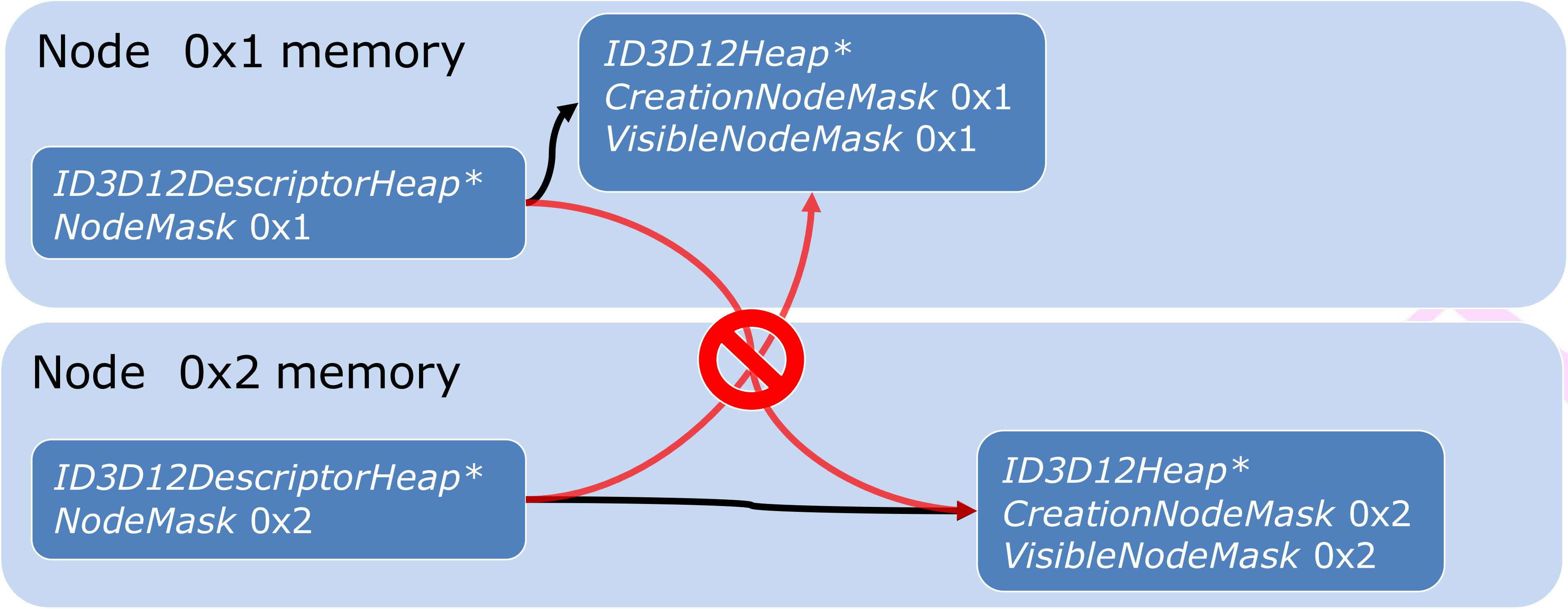
*ID3D12Heap**
CreationNodeMask 0x1
VisibleNodeMask 0x1

Node 0x2 memory

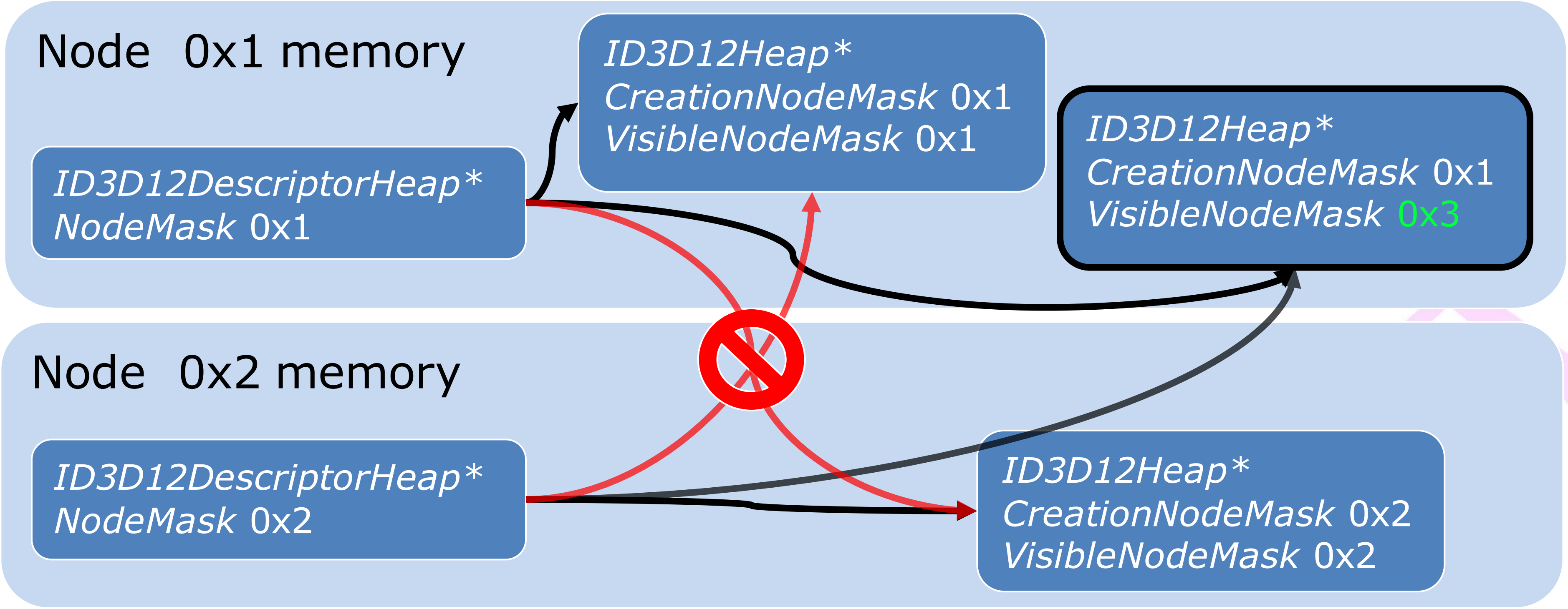
*ID3D12DescriptorHeap**
NodeMask 0x2

*ID3D12Heap**
CreationNodeMask 0x2
VisibleNodeMask 0x2

Resources - Visibility



Resources - Visibility



Resources - Assets

- Upload art assets (vertex data, textures etc.) to nodes that need them
    - It’s often convenient to upload your assets to all nodes for easy experimentation
    - AFR needs assets on all nodes
- Create a unique resource for each node, not just one that would be visible to others (with proper *VisibleNodeMask*)

Resources - AFR Targets

- AFR requires all render targets be duplicated for each node
  - Need robust cycling mechanism
- Again, a unique resource for each node, not one resource visible to all nodes

AFR Isn't For Everyone…

- Temporal techniques make AFR difficult
  - Too many inter-frame dependencies can kill the performance
  - Explicit or implicit

AFR Workflow Problem

Ideal

GPU 1	Frame 0	Frame 2	Frame 4	Frame 6	Frame 8						
GPU 0		*Frame 1*	*Frame 3*	Frame 5	Frame 7	Frame 9					
Screen	(F-2)	(F-1)	F0	*F1*	F2	*F3*	*F4*	*F5*	*F6*	*F7*	*F8*

AFR Workflow Problem

Ideal

GPU 1	Frame 0		Frame 2		Frame 4		Frame 6		Frame 8		
GPU 0		*Frame 1*		*Frame 3*		Frame 5		Frame 7		Frame 9	
Screen	(F-2)	(F-1)	F0	*F1*	F2	*F3*	*F4*	*F5*	*F6*	*F7*	*F8*

Dependencies between frames

GPU 1	Graphics	Frame 0			*Idle*	Frame 2			*Idle*	Frame 4			*Idle*	Frame 6		
	Copy		F0->F1			*Idle*	F2->F3			*Idle*	F4->F5			*Idle*	F6->F7	
GPU 0	Graphics		*Idle*	Frame 1			*Idle*	Frame 3			*Idle*	Frame 5			*Idle*	
	Copy			*Idle*	F1->F2		*Idle*		F3->F4		*Idle*		F5->F6		*Idle*	
Screen			*(F-1)*		*F0*		*F1*		*F2*		*F3*		*F4*		*F5*	

AFR Workflow Problem

Ideal

GPU 1	Frame 0		Frame 2		Frame 4		Frame 6		Frame 8		
GPU 0		Frame 1		Frame 3		Frame 5		Frame 7		Frame 9	
Screen	(F-2)	(F-1)	F0	F1	F2	F3	F4	F5	F6	F7	F8

Dependencies between frames

GPU 1	**Graphics**	Frame 0		Idle	Frame 2		Idle	Frame 4		Idle	Frame 6	
	Copy		F0->F1		Idle	F2->F3		Idle	F4->F5		Idle	F6->F7
GPU 0	**Graphics**		Idle	Frame 1		Idle	Frame 3		Idle	Frame 5		Idle
	Copy			Idle	F1->F2	Idle	F3->F4		Idle	F5->F6		Idle
Screen			(F-1)	F0	F1		F2		F3	F4		F5

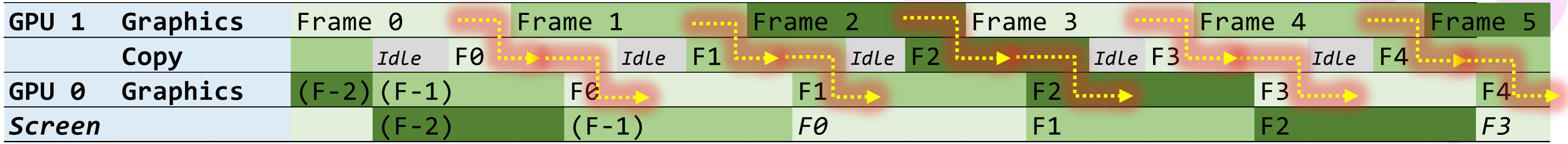
New Possibility - Frame Pipelining

- Pipeline rendering of frames
  - Begin frame on one GPU
  - Transfer work to next GPU to finish rendering and present
  - The GPUs and copy engines form a pipeline

GPU 1	**Graphics**	Frame 0	Frame 1	Frame 2	Frame 3	Frame 4	Frame 5					
	Copy		*Idle*	F0	*Idle*	F1	*Idle*	F2	*Idle*	F3	*Idle*	F4
GPU 0	**Graphics**	(F-2)	(F-1)	F0	F1	F2	F3	F4				
Screen			(F-2)	(F-1)	*F0*	F1	F2	*F3*				

New Possibility - Frame Pipelining

- Pipeline rendering of frames
  - Begin frame on one GPU
  - Transfer work to next GPU to finish rendering and present
  - The GPUs and copy engines form a pipeline

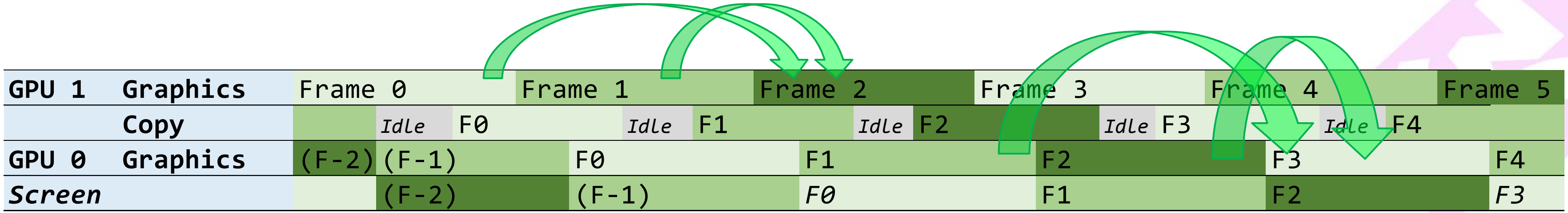


Pipelining – Simple Dependencies

- No back and forth dependencies between GPUs
  - Helps to minimize waits
  - Easier to do large cross GPU data transfers without reducing frame rate
  - Unless copying takes longer than actual work, it affects only latency, not frame rate

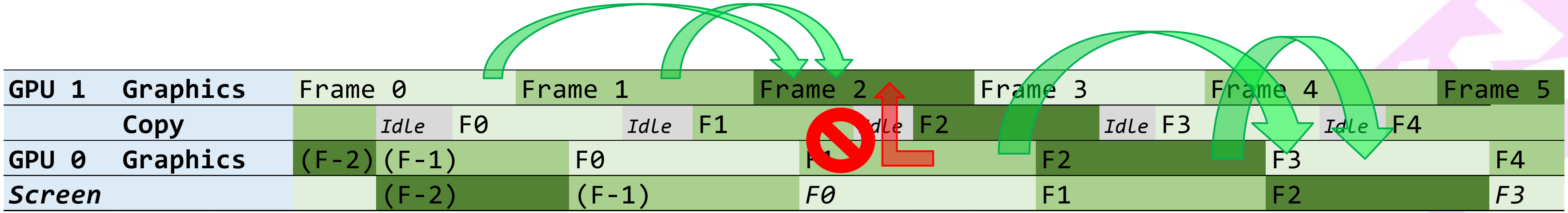
Pipelining – Temporal techniques

- Temporal techniques allowed without penalties



Pipelining – Temporal techniques

- Temporal techniques allowed without penalties
- Limitation: GPUs at beginning of pipeline cannot use resources produced further down the pipeline



Pipelining – Something More

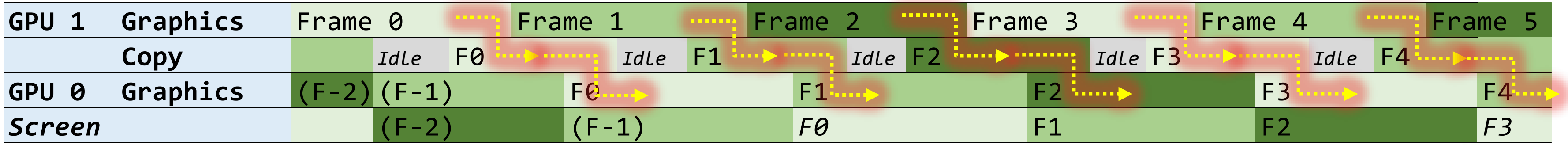
- Instead doing the same faster, do something more
  - GI
  - Ray tracing
  - Physics
  - Etc.

Pipelining – Workload Distribution

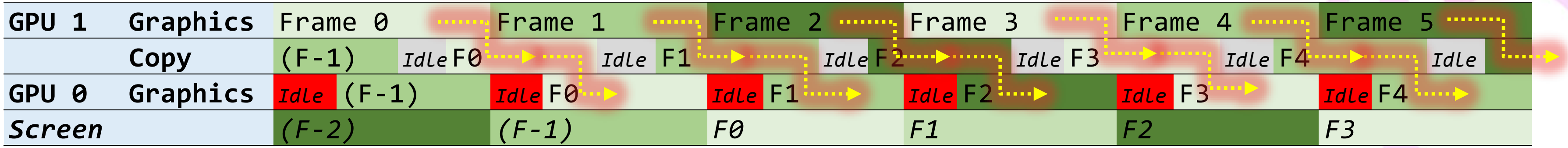
- Needs a good point to split the frame
  - Cross GPU copies are slow regardless of parallel copy engines
    - <8 GB/s on 8xPCIe3, 64 MB consumes at least 8 ms
- Doing some passes on both GPUs instead of transferring the results can be an option

Frame Pipelining Workflow

Ideal

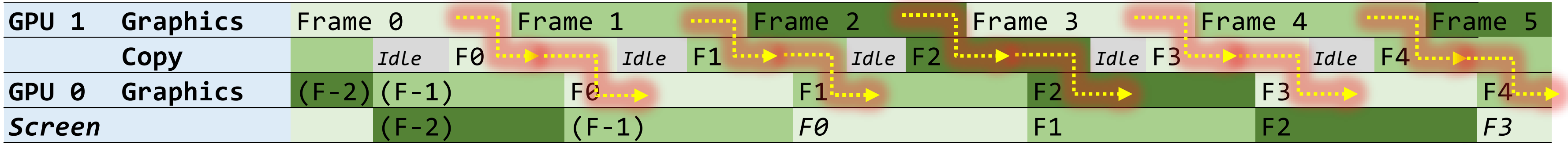


Unbalanced work

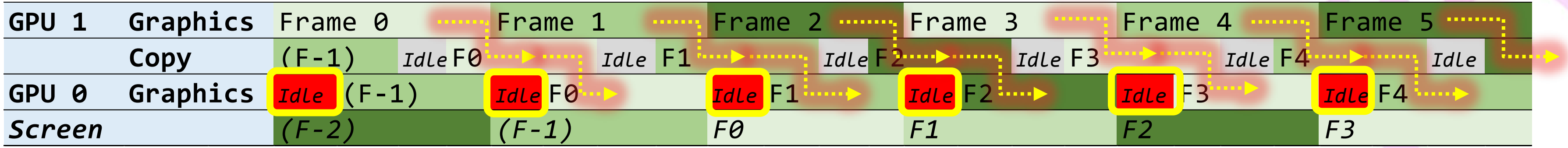


Frame Pipelining Workflow

Ideal



Unbalanced work



Pipelining – Possible Problems

- Workload balance between GPUs depends also on scene content
  - It’s never perfect, but can be reasonable
- Latency can be a problem like in AFR
- Scaling for 3 or 4 GPUs requires separate solutions

Frame Pipelining Case Study

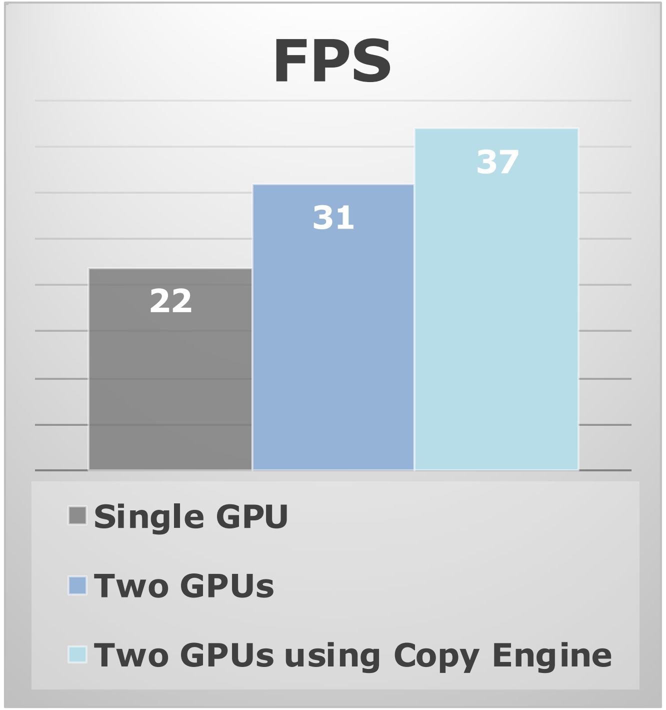
- Microsoft DX12 miniengine
  - Pre-depth
  - SSAO
  - Sun shadow map
  - Primary pass
  - Particles
  - Motion blur
  - Bloom
  - FXAA

Frame Pipelining Case Study

- As a stress test, 3840x2160 screen and 4k by 4k sun shadow map resolutions were used
- Generated on first GPU:

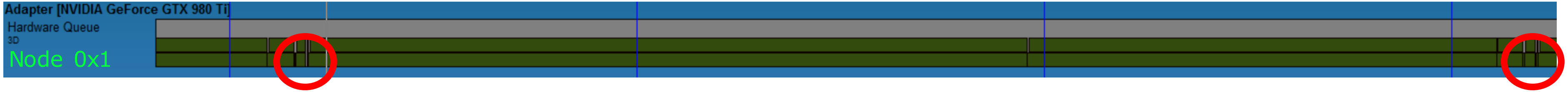
Predepth	D32_FLOAT	31.6 MB	5.3 ms
Linear Depth	R16_FLOAT	15.8 MB	2.6 ms
SSAO	R8_UNORM	7.9 MB	1.3 ms
Sun Shadow Map	D16_UNORM	32 MB	5.3 ms
Total		**87.3 MB**	**14.6 ms**

Frame Pipelining Case Study - Performance



Pipelining Case Study - GPUView

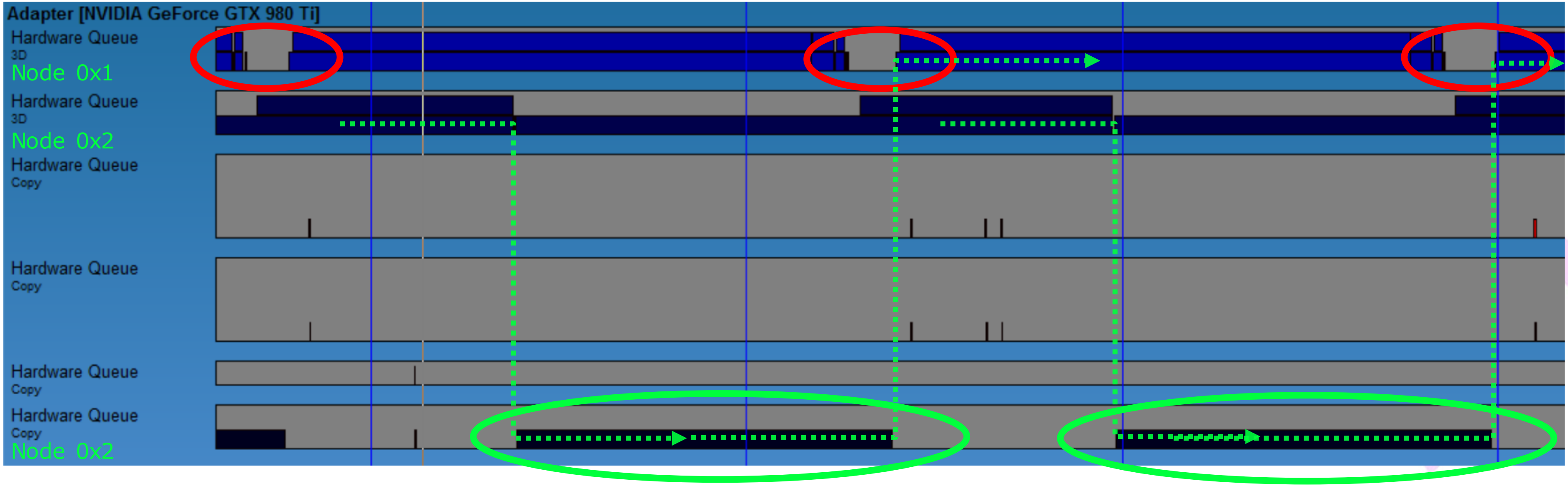
Original single GPU workflow



Two GPUs pipelined without copy engine

Pipelining Case Study - GPUView

Two GPUs pipelined with copy engine



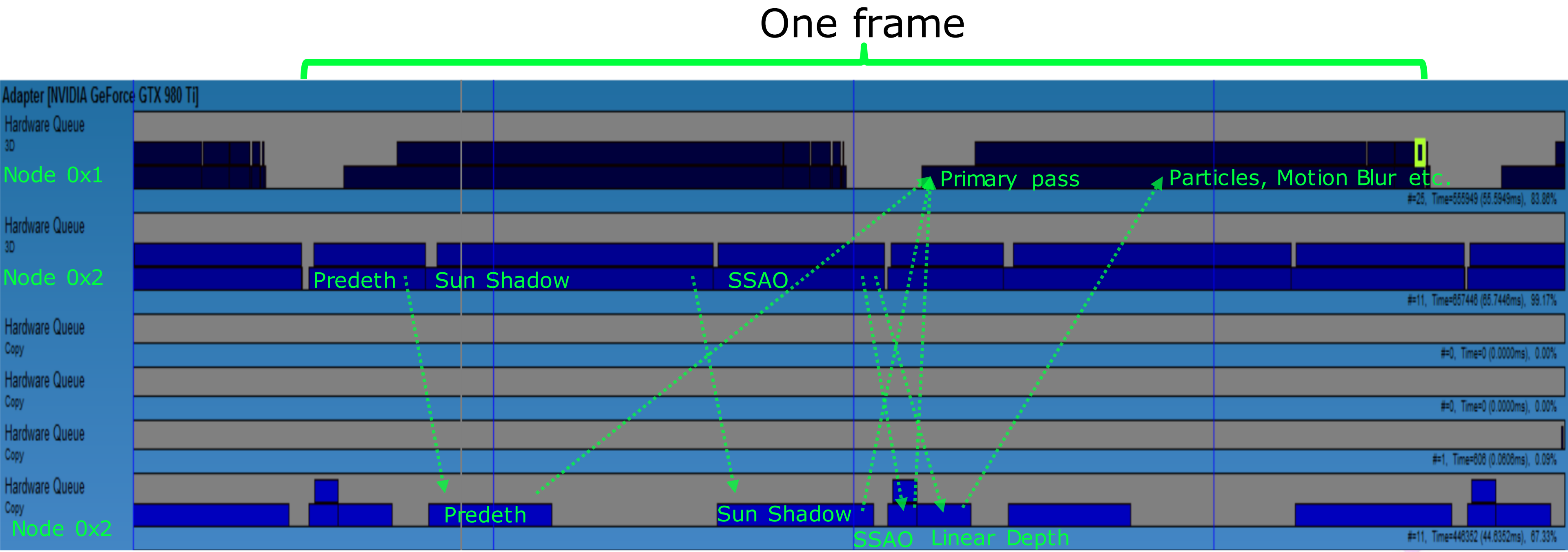
Frame Pipelining Case Study

- *1.7x framerate from single to dual GPU*
  - Pretty even workload distribution, but it’s content dependent
- Cost of copying step would limit frame rate to about 60 fps on 8xPCIe 3.0 system

Pipelining – Hiding Copy Latency

- Break up copy work into smaller chuncks
  - Overlap with other work for the *same* frame
  - More and smaller command lists
  - *Remember guidelines from the "Practical DirectX 12"*
- In the case study, the ~15 ms extra latency from copies can be almost entirely hidden

Hiding Copy Latency - GPUView



Summary

- No more driver magic
- You‛re in control of AFR
- Try pipelining with temporal techniques!
- Remember copy engines!
- You can do anything you want with that extra GPU - Surprise us!

Questions?

- jsjoholm@nvidia.com