VISIBILITY BUFFER AND DEFERRED RENDERING

in DOOM: The Dark Ages

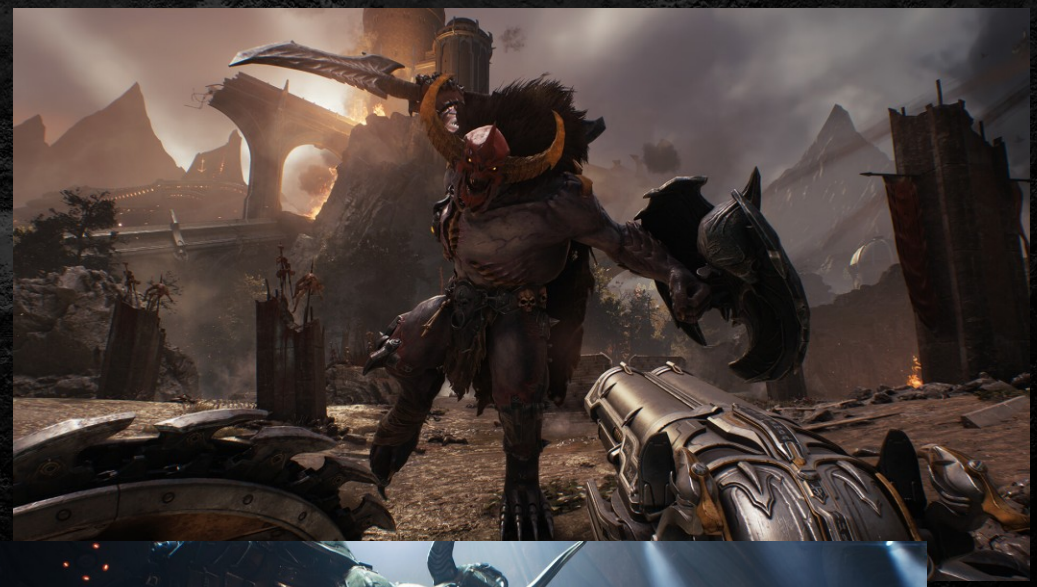
Dominik Lazarek
Principal Engine Programmer

Philip Hammer
Principal Engine Programmer

idTech: Driving the iconic FPS games at id Software

Mission-Statement for idTech 8 and DOOM: The Dark Ages

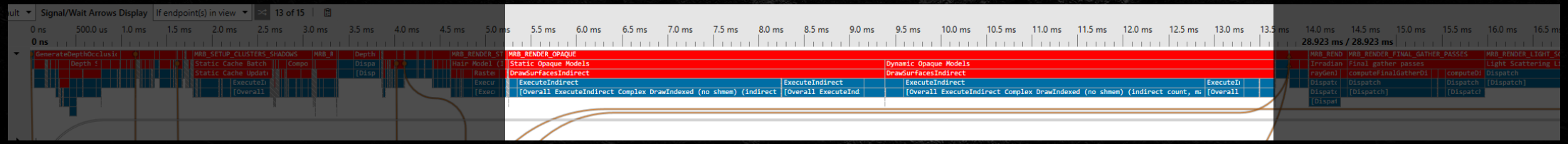
- Higher-fidelity geometry and material layering
- “More of everything”: Triangles, NPCs, Map-sizes,….
- 60 FPS + low latency across all targets



Fast-Forward: September 2024 (Release: May 2025)

- ~8 Months before ship date
- Main content and design done
- Nowhere near our performance targets
- Optimizations desperately needed
- Engine department entered optimization phase across the board

- Some of us focused on opaque geometry rendering



Dissecting Opaque Geometry Bottlenecks

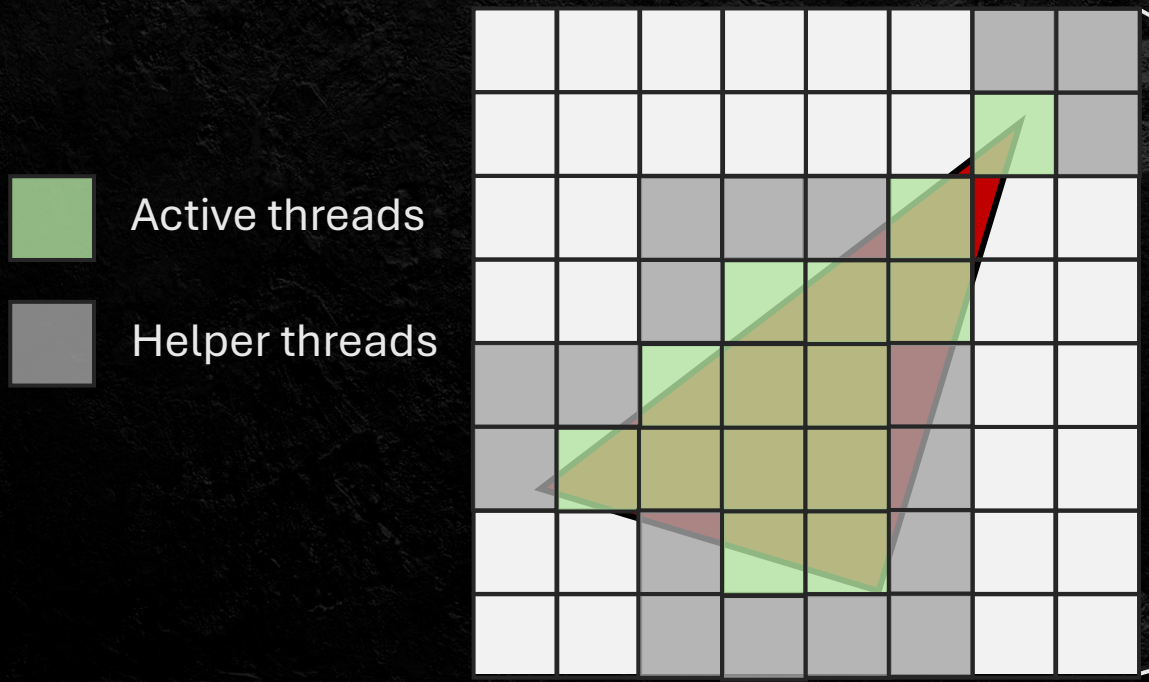
- Opaque geometry rendering contributes to ~40% of total frametime
- Where do we lose performance?

Observations:

- Opaque rendering pipeline: Clustered Forward+
- Two full geometry-passes needed
- Complex texturing & lighting shader
  - Added features (POM, Material-blending,…)
  - Large code-size, high VGPR count
- Main slowdowns on views with high triangle density

Dissecting Opaque Geometry Bottlenecks

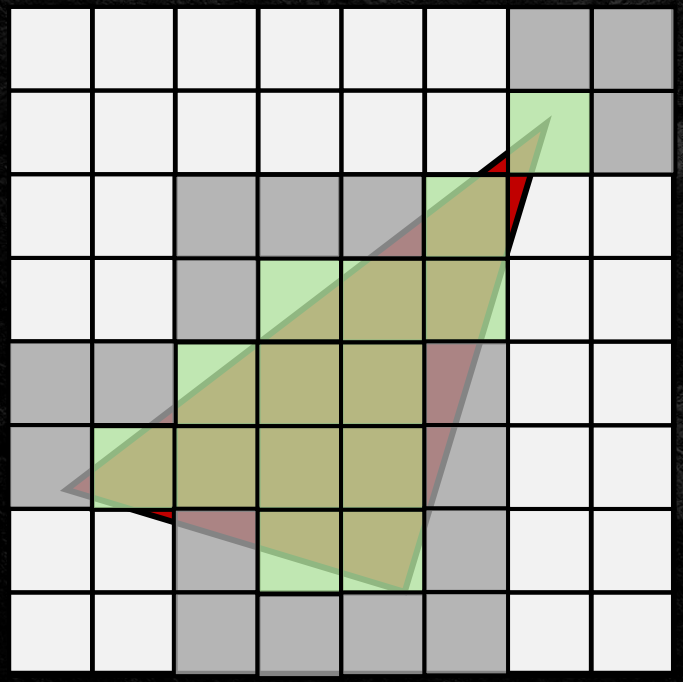
- Profiling showed: Very bad quad utilization efficiency
- What is quad utilization anyway?
- Let's look at a quick example of a rasterized triangle somewhere on screen



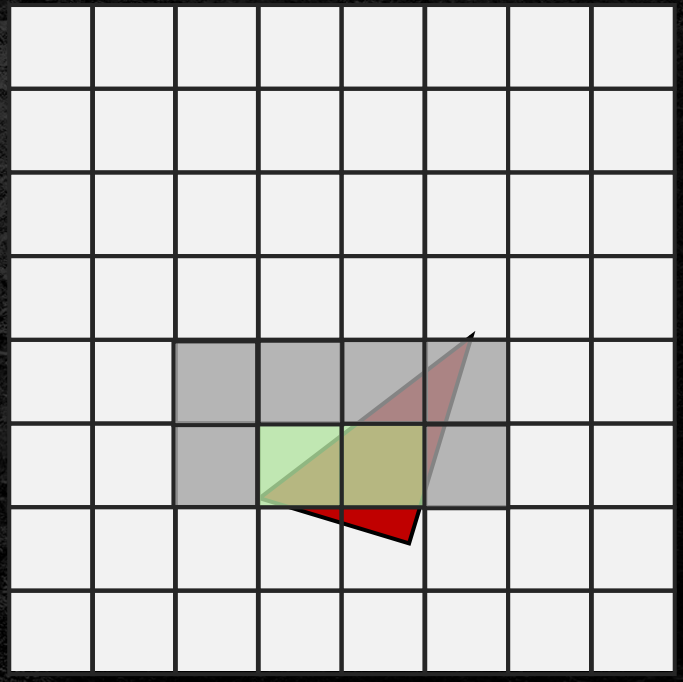
14 active threads + 18 helper threads = 32 pixel shader threads overall
Quad utilization: 14 active / 32 overall = 43% of threads „utilized“

Dissecting Opaque Geometry Bottlenecks

- Quad utilization efficiency becomes **worse** with smaller triangles (at constant resolution)



14 active / 32 overall = 43% utilization



2 active / 8 overall = 25% utilization

Dissecting Opaque Geometry Bottlenecks

How to deal with bad quad utilization efficiency?

- Can’t avoid quad-shading in HW rasterization
- Software rasterization for sufficiently small triangles? -> Out of scope
- Limit impact of helper-thread overhead
  - Make depth pass the only rasterized geometry pass
  - Will suffer from helper-threads, but cheap shader logic
  - Expensive texturing & lighting logic: Avoid expensive helper-thread overhead (e.g. offload to compute shader)

Dissecting Opaque Geometry Bottlenecks

Things to solve when Texturing & Lighting is done in compute:

- Need triangle-attributes per pixel (e.g. UVs, derivatives…)
- Need to dispatch different compute shaders for different groups of pixels (e.g. different materials,…)
- No hardware-VRS usable anymore

Triangle Visibility Buffer +Attribute Interpolation

Material Compute Dispatch

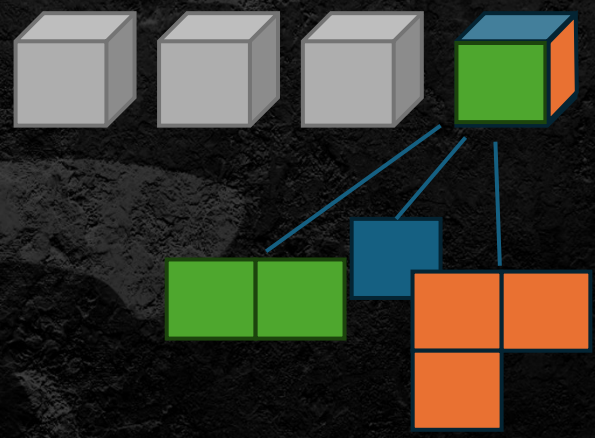
Variable-Rate Compute Shaders (VRCS)

While we're at it:

- Split texturing- & lighting into separate passes
  - Lower VGPR count for each stage
  - Easier for shader-compilers to optimize
  - We had G-Buffer anyway…

Interlude: Clarifying Terminology / Common idTech concepts

- „Surfaces“ are the smallest elements from models that we render.
- Identifies a portion of the geometry + a material
- “Material”: Textures + Parameters – not shader
- Vertex Position + attributes available as large buffer on GPU
- GPU-Driven model-gather & (triangle-)culling
- Material-Textures are „bindless“

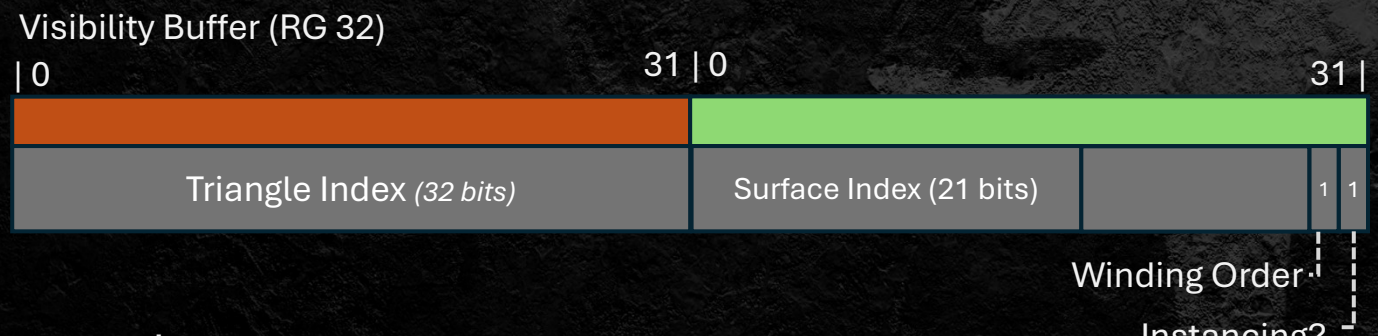


Models

Surfaces
(Mesh-part + Material)

Triangle Visibility Buffer

- Generated during Depth Prepass early in frame
- Writing `gl_PrimitiveID` in pixel shader
- „Depth Prepass“ → „Visibility Pass“



- 64 bpp texture storing:
  - Triangle index
    (resolved to pre-culled index; relative to mesh)
  - Surface/Instance index
  - Flag: Winding order
  - Flag: Uses HW-instancing?

Triangle Index

Surface/Instance Index

Triangle Visibility Buffer

- Export more data during Visibility Pass?
  - UV-Derivatives?
  - Tangent Frame?
  - Both required for texturing/lighting later on
  - Trivially available during Visibility Pass (HW rasterization with quad helper threads)

Let's profile!

- „Slim" Visibility Buffer @ 64 bpp (Triangle- & Surface Index)
  - No measurable performance impact
  - Still bottlenecked by vertex processing; Adding more pixel-exports „free".

- „Fat" Visibility Buffer @ 128 bpp („Slim" + UV-Derivatives + Tangent Frame)
  - Performance impact measurable now
  - Bottleneck becomes pixel-export bandwidth ☹

→ Stick to „Slim" variant and reconstruct from barycentrics later

Material Dispatch Generation

- Visibility buffer generated – now what?
- Need to derive draw-calls from it
- Lots of good references around
  e.g. Horizon: Forbidden West ([McLaren22]), Epic Games ([Karis21], [Wihlidal24]), Eidos Dawn Engine (Doghramachi17)

Two often-cited approaches:

- Quad-Dispatch
  - Render (Fullscreen-) Quad per material.
    „Early-out“ on pixels not matching material

- Compute Dispatch
  - Build explicit lists of pixels per material
  - Sort into GPU-Waves and dispatch over them

  - → Decided to go with Quad-Dispatch first (faster to get up and running…)

Material Quad Dispatch

- How to efficiently „early-out“? Discard() per pixel? → Too late! Wave already launched
- Idea [Doghramachi17]: Use a „fake“ depth buffer
  - Depth buffer contains material-IDs, primed in additional full-screen triangle pass
  - For each unique material on screen:
    - Render quad over screenspace bounds of that material (depth test = `GL_EQUAL`)
    - Vertex shader: Output matching „fake depth“ value as well
  - → Only pixels matching this material will get rasterized.

Material Quad Dispatch

Ok, so how does it perform?

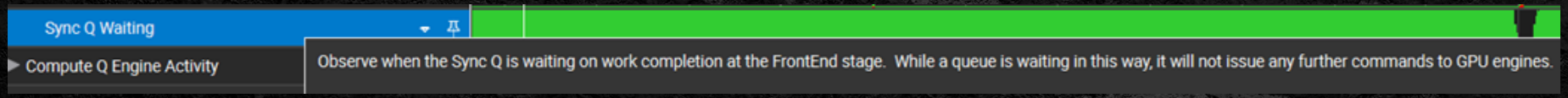
Forward+	Quad Dispatch
14.3 ms	14.26 ms

(RTX 5080 @ 4k)

☹ This isn‘t any improvement at all! Why?!

Material Quad Dispatch – Dissecting Performance Bottlenecks

Let's fire up NSIGHT and profile!



New pixel shader work is stalled on waits on the FrontEnd stage

Depth Pixel Data Flow	Pixels-per-Cycle	Throughput %	Pixels
ZCULL Input Pixels	0.4	0.2%	64,564,880.0
ZCULL Output Pixels	0.3	0.1%	39,391,968.0
PROP Input Pixels	0.3	0.4%	38,316,775.0
ZROP EarlyZ Input Pixels	0.3	0.4%	38,306,007.0
ZROP EarlyZ Output Pixels	0.0	0.0%	2,663,442.0
SM Pixels Shaded in EarlyZ Mode	0.0	0.0%	42,033.0

A LOT of input pixels!

Still A LOT after coarse rasterization (Hi-Z)

Most of the pixels only get culled during fine rasterization (Early-Z)

Interpretation:

- Hi-Z isn't very effective with the „fake" depth buffer.
  - Contains only material-Ids, not hierarchical at all – can appear random on screen
- Fine rasterization has too much work, becoming a bottleneck

- Decided to also implement the compute-based dispatch approach next!

Interlude: Opaque Shader Performance Characteristics

Let's first review some opaque shader performance properties:

- Forward+: Each drawcall comes from a single surface
  - Material- & Entity- constants are **uniform** across a whole pixel-shader wave
  - Material-Textures are sampled **uniformly** across a wave
  - **Uniformity** known at **compile-time**: Shader compiler can produce more optimal instructions.

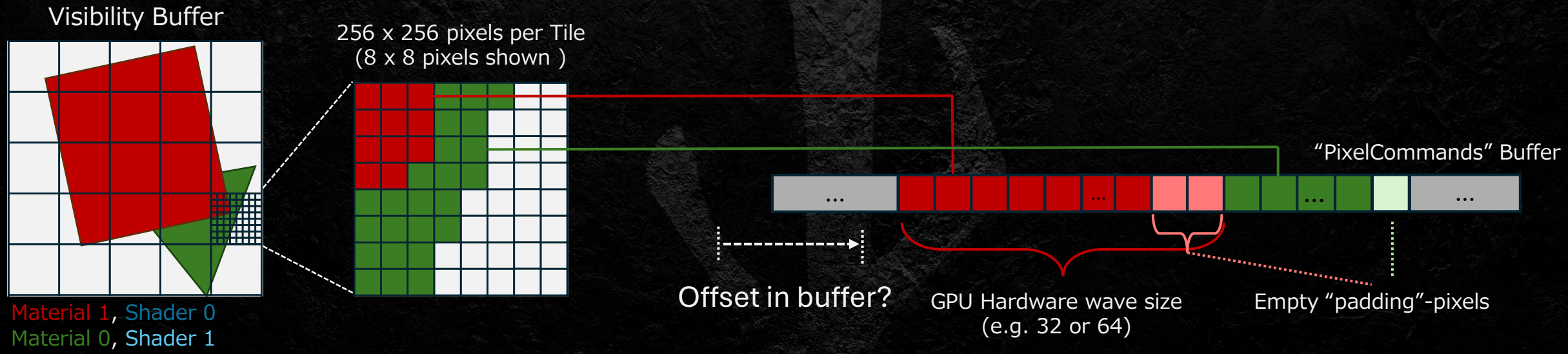
- → **Compute-Dispatch needs to maintain uniformity as best as possible**
  - Keep materials uniform per wave → Uniform texture- and buffer-reads

Material Compute Dispatch

- Our approach loosely follows the ideas presented in [McLaren22]

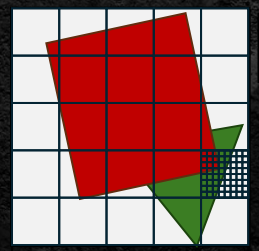
Core idea:

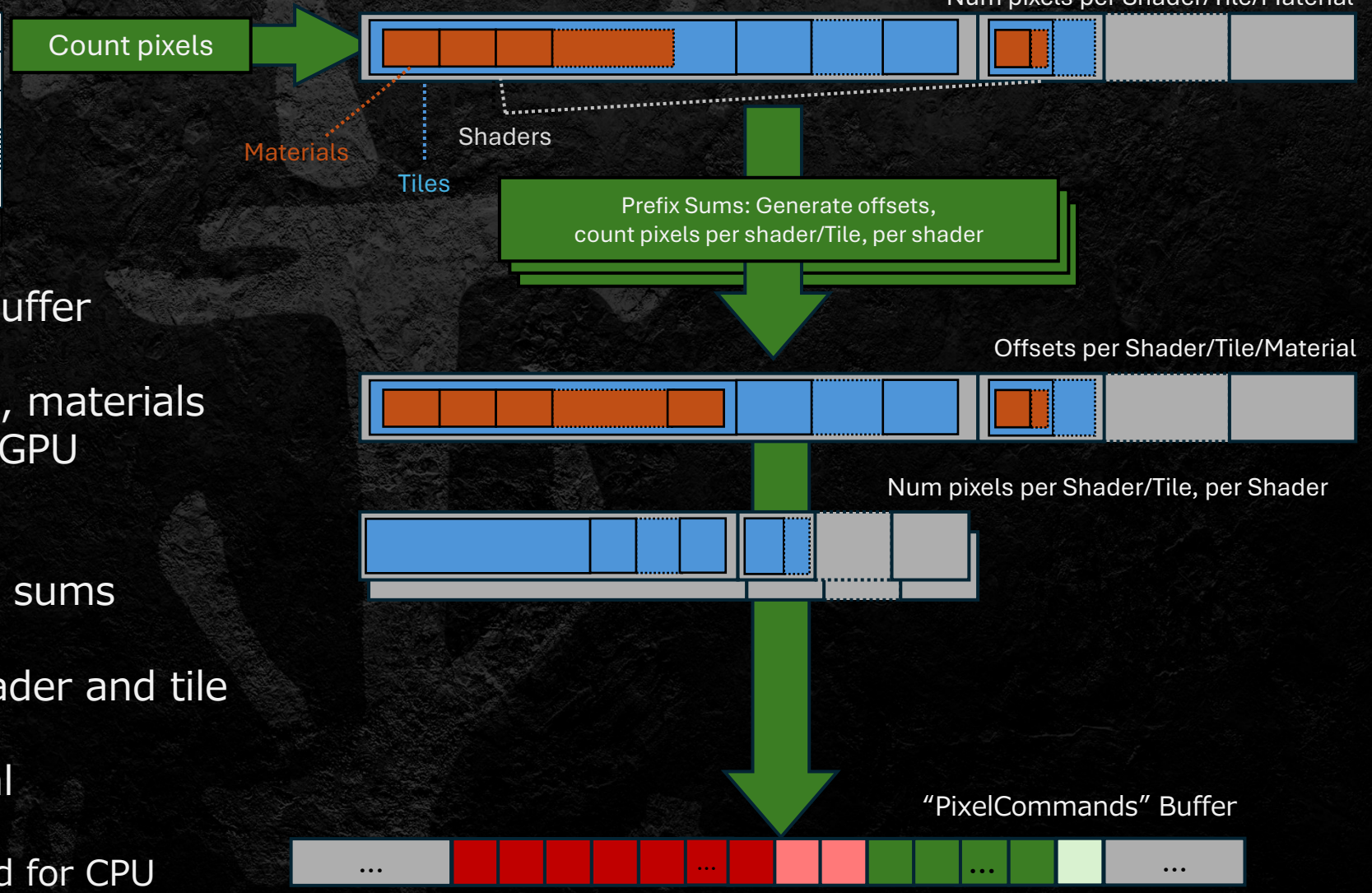
- Process Visibility Buffer in Tiles
- Write tile-local pixel-coordinates into large buffer
- Padded up to HW wave size – keeps material uniform within wave
- Sorted by shader, tile and material



- But: How do we come up with the correct offset into the buffer?

Material Compute Dispatch - Details





- Series of compute passes
- Counting pixels in Visibility Buffer per shader, tile, mat
- Restricted to **visible** shaders, materials
  Visible set registered during GPU Gather/Culling

- Generate offsets using prefix sums
- Results:
  - Number of pixels per shader and tile (all materials)
  - Global offset per material
  - Global offset per shader
    (all tiles, materials – needed for CPU dispatch)

Material Compute Dispatch – Dispatching Waves

- On CPU-Side: (Multi*-) indirect dispatch call for each Shader-pipeline
- Each dispatch launches all material-waves within a single tile
- In dispatched shader:
  - Read offset into `pixelCommandsBuffer` at `DispatchID*` (encodes PSO- & tile-index)
  - Read pixel-coordinates at `pso_and_tile_index + gl_LocalInvocationID.x`
  - (Actual material inferred from Visibility Buffer (surface Index))

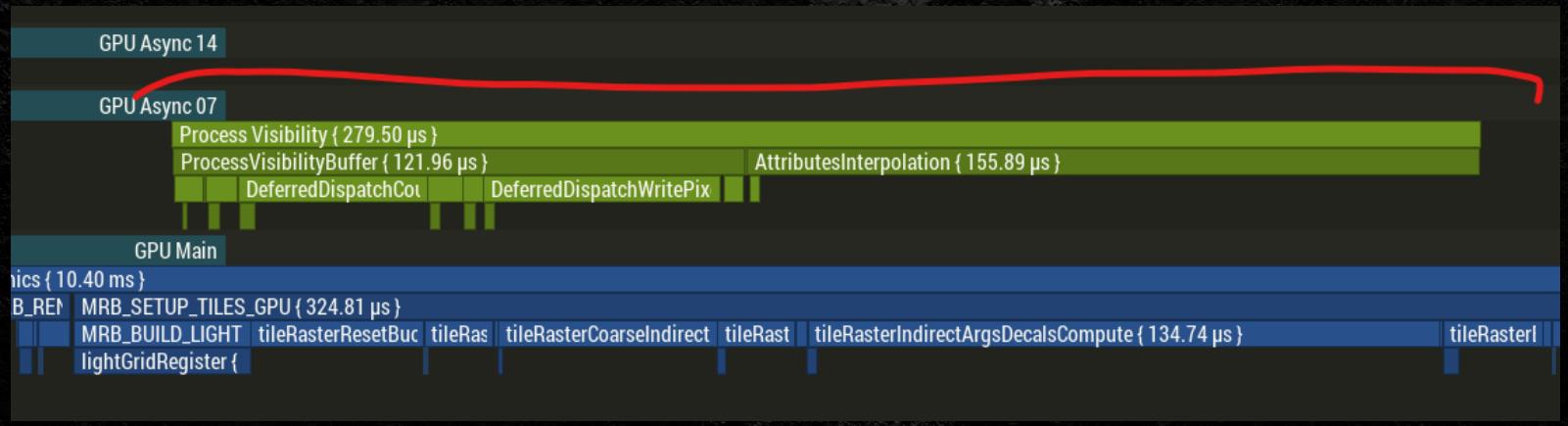
- *Multi-indirect dispatch only available on D3D12 (custom ID3D12CommandSignature)
- On other platforms: Have to emulate
  - Group multiple (logical) dispatches within each `Group in X-dimension`
  - `Group Y-dimensions` collects individual dispatches → `gl_LocalInvocationID.y == DispatchID`
  - Reduce overhead from larger Group-counts in X-dimension by splitting dispatches into a series of power-of-two sized buckets

Material Compute Dispatch – Debugging

- No actual shader-logic yet – but we can already debug!
- Investment in debugging visualizations / shader-printf / shader-asserts always a win

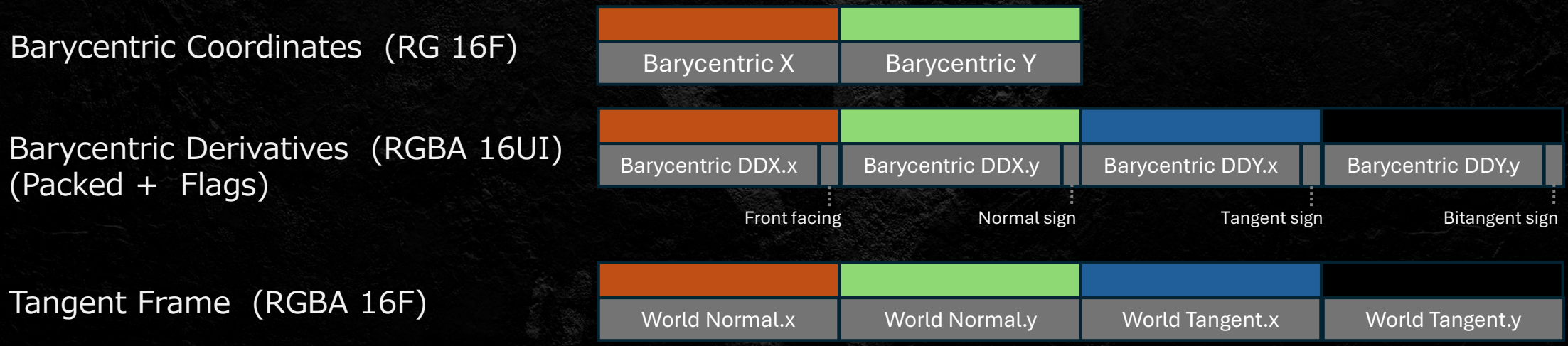
Deferred Attribute Interpolation

- We can render random colors very efficiently now!
- Next up: Textures & Lighting
- All compute based now – need to manually compute vertex-attributes, UV-Derivatives, Tangent Frame
- Performed in a dedicated compute pass running on async together with pixelCommand builds
- Could also do this inline in texturing/lighting shader, but increases VGPRs considerably



Deferred Attribute Interpolation

- Actual interpolation math nothing new (follows old software rasterizers such as [Hecker95])
- We interpolate & store:
  - Barycentric coordinates (2x Float16)
  - Packed Barycentric derivatives (DDX / DDY) + flags (winding order, tangent sign)
    Derivatives clamped to `]-2.0..2.0[` to save one bit and store flags.
  - Tangent Frame (4x Float16)



Material Compute Dispatch – Results

- Initial version: Compute dispatch w/ attribute interpolation as direct replacement for Forward+ (Texturing + Lighting combined)
- Using only the most basic features to compare performance w/ Forward+

- Early performance results look much more promising than quad dispatch
  → We decided to continue with this direction!

Forward+ (ms)	Quad Dispatch (ms)	Compute Dispatch (ms)
14.3	14.26	10.4*

* 10.4 = 0.6 dispatch generation + 9.8 render w/ attribute interpolation

- Caveat: Performance-wins vary with visible scene
  (e.g. large triangles on ground: Not much win – but also not worse!)

Forward+ (ms)	Compute Dispatch (ms)
2.75	2.7

Deferred Texturing

- Each shader-pipeline supporting deferred texturing gets a new compute-shader stage
- Same logic as Forward+ pixel shader, but only blends materials

- Dispatched using the pixelCommands buffer and inidrect dispatch args preared from Visibility Buffer earlier

- Output to GBuffer

```
shaderPipeline outside {
    hlsl_vp {
      // Forward+ Vertex Shader
    }
    hlsl_fp {
      // Forward+ Pixel Shader
    }
    hlsl_cp {
      localSize { 64, 1, 1 }
      // Deferred Texturing Compute Shader
    }
}
```

Deferred Texturing - Optimizations

- Pixel-command wave packing ensures that bindless texture indices are **uniform**
- **However**: Meshes/Entities can still be divergent:
  Less optimal Mesh/Entity-cBuffer access patterns
- Results in higher VGPR count, less efficient compiled shader code

Deferred Texturing - Optimizations

Observation:
In most cases, all wave-pixels in fact come from the same mesh & entity

- Leverage with a simple trick:
  Dispatch all waves in **two** permutations – divergent vs uniform data
- Check uniformity with subgroup ops – early out if assumption is violated
- Slightly better: Build two different dispatch commands from the start. Would avoid overhead starting waves and earlying out

```
const bool dataIsUniform = subgroupAllEqual( meshDataHandle );
#if defined( SHADER_VERSION_UNIFORM_DATA )
    if ( dataIsUniform == false ) {
        // Uniform version but data is divergent, early out
        // Pixels will be processed by the divergent version
        return;
    }
    // Data is uniform, use subgroupBroadcastFirst() to let the shader compiler know
    meshDataHandle = subgroupBroadcastFirst( meshDataHandle );
#else
    if ( shaderHasUniformDataVersion && dataIsUniform ) {
        // Divergent version but data is uniform, early out
        // Pixels will be processed by the uniform version
        return;
    }
#endif
```


Deferred Lighting
G-Buffer Update
Tile Classification
Variable-Rate Compute Shaders

Deferred G-Buffer Update & Lighting

Current state: G-Buffer from our opaque geometry – what's next?

Blend G-Buffer modifications

- Geo Decals, Projective Decals, Wetness and Rain VFX, Triplanar Blood Layer

Compute Lighting

- Final G-Buffer, only write-out 1 RT with final lighting
- Same clusters and very similar shader code as F+

Option to compute combined or split into 2 phases

Split setup: 1 additional G-Buffer read, but less complex shaders

- On most platforms a decent win
- Only write touched G-Buffer channels

Deferred G-Buffer Update & Lighting

Deferred G-Buffer Update & Lighting

Deferred G-Buffer Update & Lighting

Deferred G-Buffer Update & Lighting

Deferred Dispatches with Tile Classification

Split uber-shaders into feature set permutations with varying complexity

-> Keep VGPR usage low

Select permutation based on classification tile mask

Segment screen into 32x32 pixel tiles

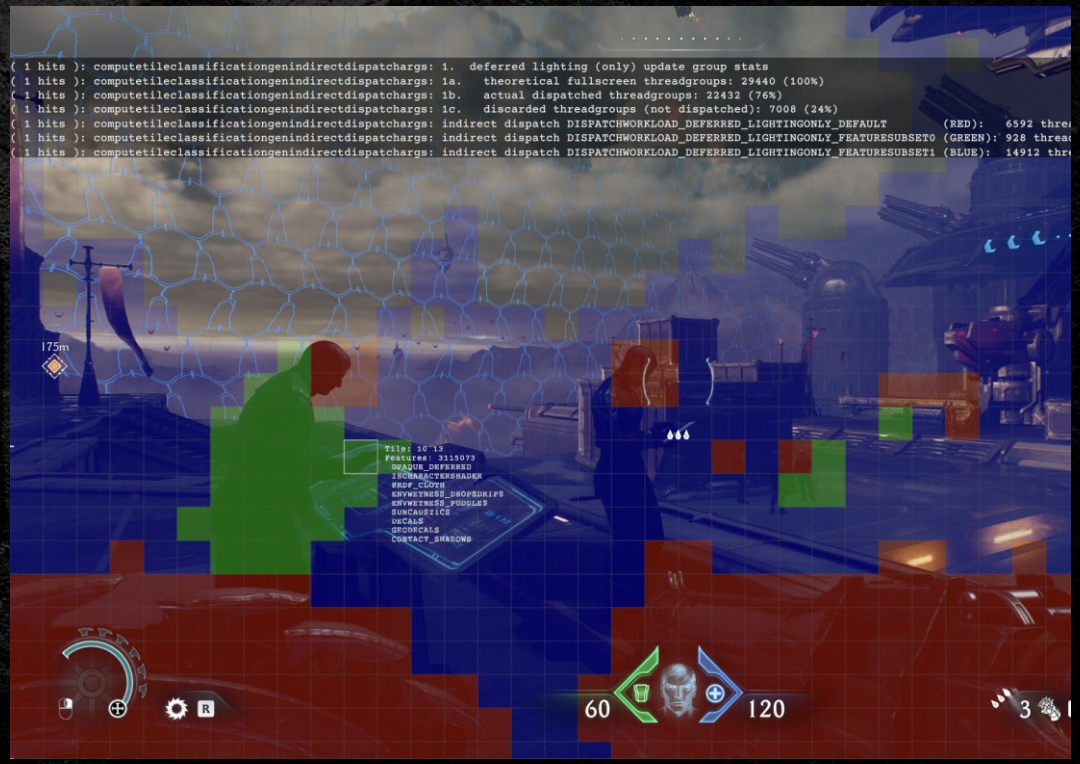
Matches clustered bin tile size for lights/decals

Goal is to indirect-dispatch tailored shaders for each tile

Split uber shaders into smaller subsets

Better wave occupancy (less VGPRs)

No dispatch for unnecessary tiles (avoids per-pixel checks)



Deferred Dispatches with Tile Classification

Each tile stores 32-bit bitmask

Add feature bit if pixel touches the tile

A feature can be BRDF type, „hasDecal“, etc.

Build feature bitmask using scalarized **atomicOr** throughout the frame

TC_FEATURE_OPAQUE	0x01
TC_FEATURE_BRDF_CLOTH	0x02
TC_FEATURE_BRDF_SKIN	0x04
TC_FEATURE_DECALS	0x08

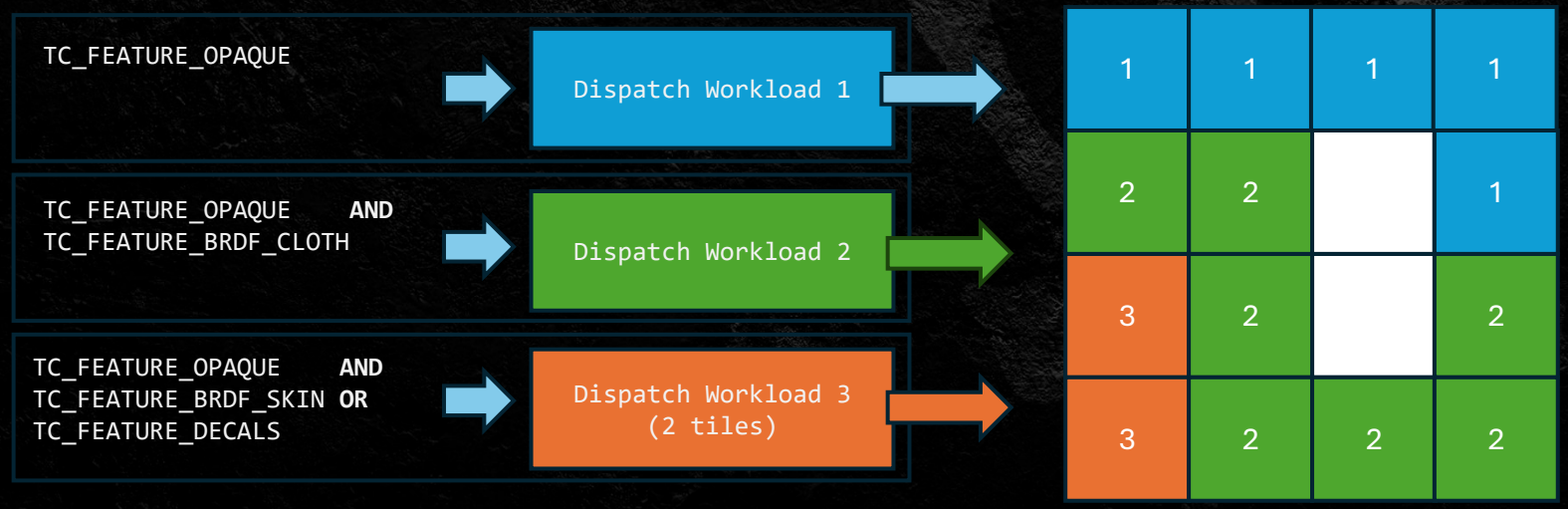
```
uint featureBitmask = 0x0;
featureBitmask |= (hasSkinBRDF) ? (1 << TC_FEATURE_BRDF_SKIN)   : 0x0;
featureBitmask |= (hasClothBDRF) ? (1 << TC_FEATURE_BRDF_CLOTH) : 0x0;

uint64 remainingLanes = subgroupBallot( true );
while ( anyNotEqual( remainingLanes, 0x0 ) ) {
  uint laneIndex = subgroupBallotFindLSB(remainingLanes);
  uint uniformIndex = subgroupBroadcast(bufferBaseIdx, laneIndex);
  uint64 equalMask = subgroupBallot(bufferBaseIdx == uniformIndex);
  remainingLanes &= ~equalMask;

   if (gl_subgroupInvocationID == laneIndex) {
       atomicOr($tileClassificationPayloadBufferRW[index], featureBitmask);
   }
}
```

Deferred Dispatches with Tile Classification

Tile features translate to specific compute workloads
- Each workload is an indirect-dispatch call
- Each tile is an indirect argument as part of the workload

Workload generation
- Store tile count per workload
- Store tile position per tile as 1D index

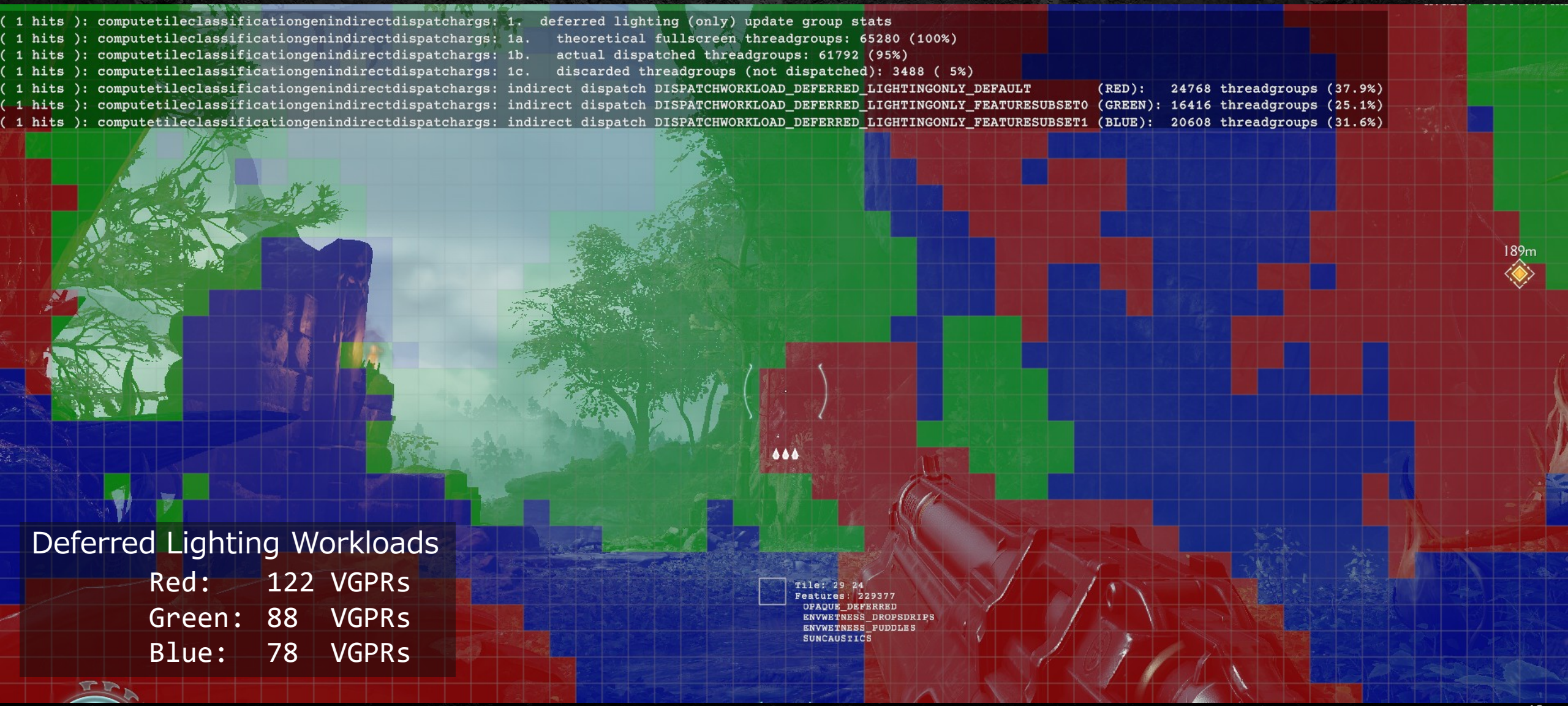
Workload dispatch
- Reconstructed global 2d screen position in shader



Example Tile Classification for different Passes

Example Tile Classification for different Passes

Example Tile Classification for different Passes



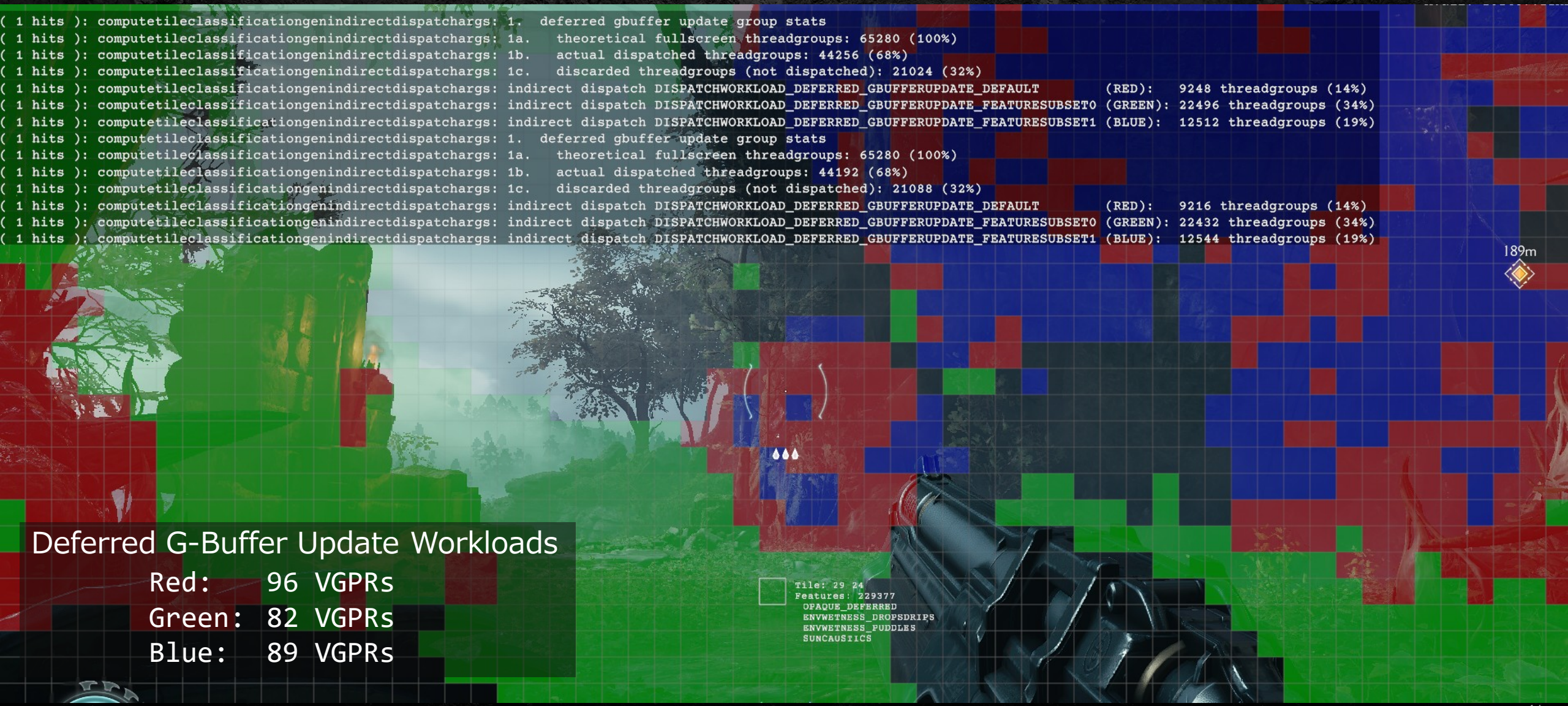
Deferred Lighting Workloads

Red: 122 VGPRs

Green: 88 VGPRs

Blue: 78 VGPRs

Example Tile Classification for different Passes



Deferred G-Buffer Update Workloads

Red: 96 VGPRs

Green: 82 VGPRs

Blue: 89 VGPRs

Example Tile Classification for different Passes

Deferred SSS Update Workloads

Red: Run Filter Chain

Maintaining Visual Parity

Make sure to not introduce subtle differences

Developed debug tools on the fly when necessary

Shader printfs, debug color, debug geometry (lines, spheres, boxes, etc.)

Variable-Rate Shading

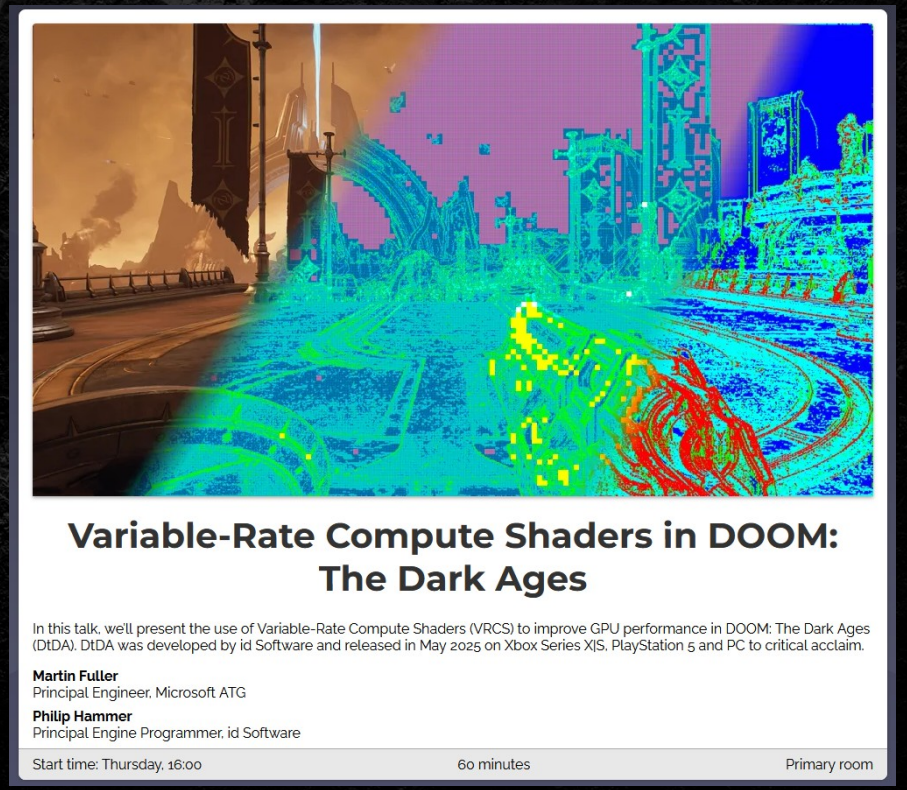
Hardware VRS worked great in DOOM Eternal / Forward+

Doesn’t work with compute-based Deferred Texturing/Lighting

Software-based VRS alternative for compute shaders

Variable-Rate Compute Shaders (VRCS)

VRCS Deep Dive later today at 4pm!



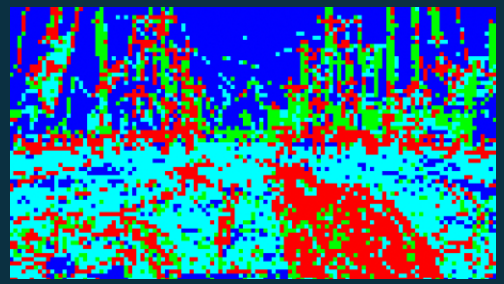
Variable-Rate Compute Shaders

Compute Shading Rate

Reduce amount of uniquely calculated pixels

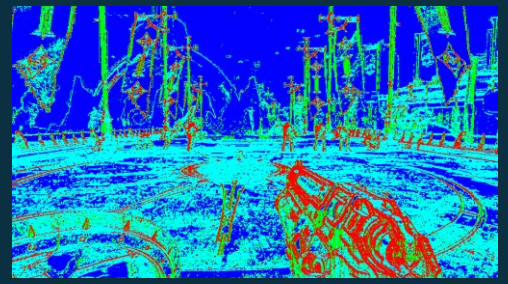
Re-schedule compute threads to retire compute waves early

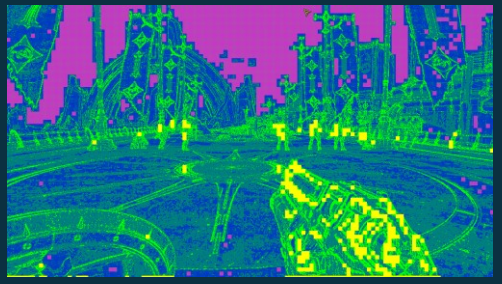
Hardware VRS



```
VkRenderingFragmentShadingRate
AttachmentInfoKHR

RSSetShadingRateImage()
```

Variable-Rate Computer Shaders





Variable-Rate Compute Shaders

VRCS for all deferred opaque geometry
    (Deferred Texturing + Lighting + G-Buffer Update)

Hardware-VRS still supported for Transparents / F+ opaque

Solid win in performance at comparable image quality
    ~10% in in GPU time in average (1-2 ms)
    Depending on platform and resolution

Great success, but also not a silver bullet
    non-trivial to catch all corner cases

scarlett m2a rs 85%	FRAME	D-TEX	D-GBU	D-LGHT
VRCS on (ms)	15.9	1.473	0.897	1.352
VRCS off (ms)	17.7	2.287	1.118	2.053
delta (ms)	1.8	0.814	0.221	0.701
perf gain (%)	10.17%	35.59%	19.77%	34.15%

Summary & Results

Shipped a hybrid rendering approach in DOOM: The Dark Ages

Visibility Buffer / Deferred Rendering / Forward+

Saved several milliseconds of GPU performance
Not all cases a substantial win, but never a loss!
Diminishing returns with low triangle density

High memory requirements
Quite an effort 8 months before ship.
Underestimated time to reach shading-parity

More complex rendering
Non-trivial to pinpoint issues

Summary & Results

- Saved up to ~25% on our targets in average
- Bonus: Resolution scaling is more effective in Deferred
  - Performance scales better (almost linearly) with number of pixels.
- Luckily, we could afford the additional memory
  - Memory scales well with resolution
  - More positive than negative effect: Weaker platforms (consoles) usually run at lower resolutions, reducing memory overhead

GPU TIME (Xbox Series X)	FWD+ 2560x1440 100%	FWD+ 1811x1019 50%	Deferred 2560x1440 100%	Deferred 1811x1019 50%
Opaque pass (ms)	9.43 (100%)	6.4 **(-33%)**	7.5 (100%)	3.9 **(-48%)**

MEMORY	3840x2160	2560x1440
Overall	**330.48 MB**	**143.39 MB**

Future Developments

Run deferred texturing on async compute
- Didn’t find a good place so far, not enough low-ALU graphics work anymore

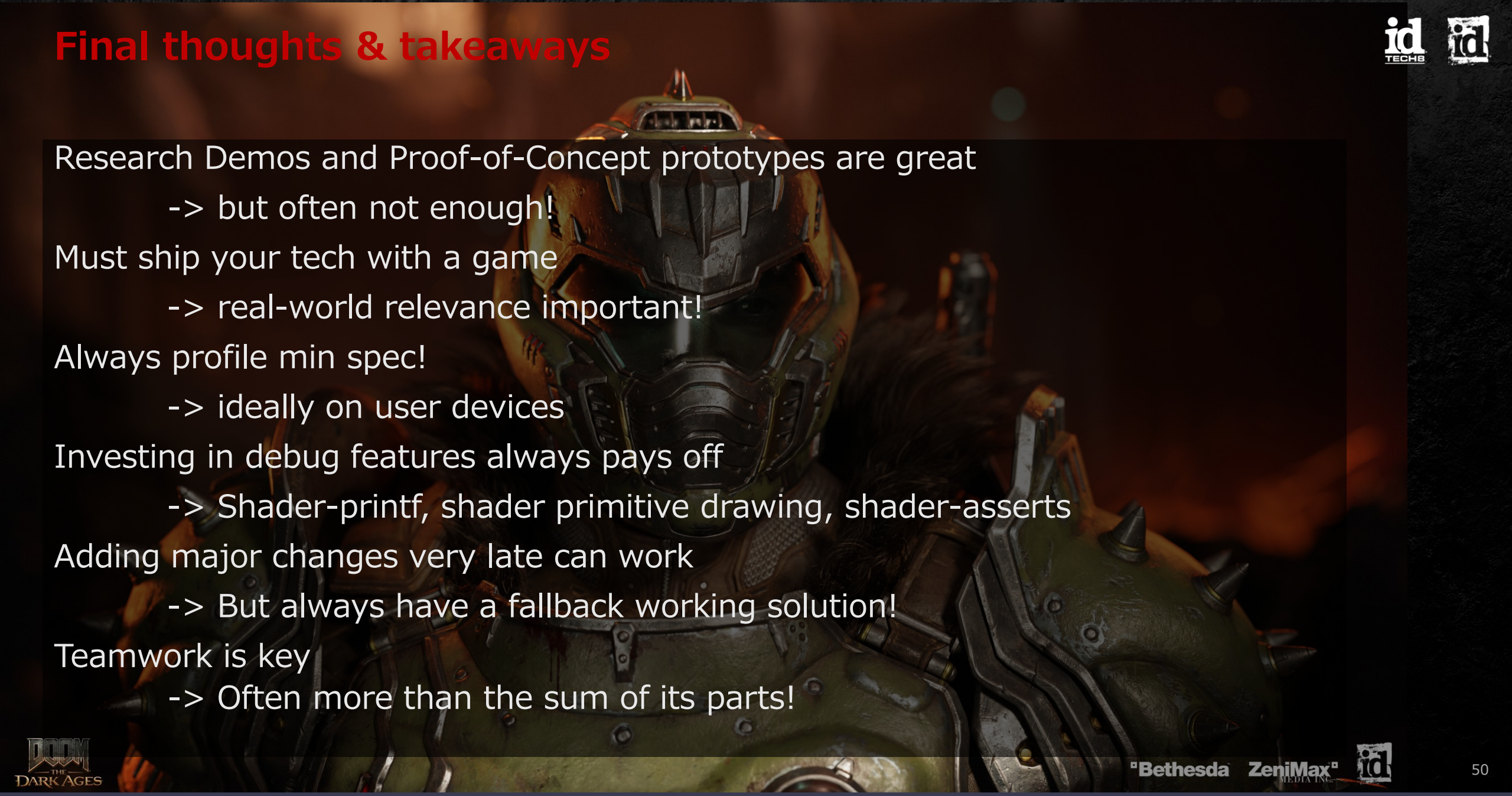
Use pixel dispatch commands also for deferred lighting stage
- Would allow more targeted shaders for lighting, less uber-shaders
- No need for (much coarser) tile classification anymore

Run Texturing, Decals, Lighting all in one shader again
- VGPR and shader complexity issues

Reduce memory footprint of all techniques
(e.g. by leaning into memory-aliasing more)

Use software-rasterization for visibility pass

Final thoughts & takeaways



Research Demos and Proof-of-Concept prototypes are great

-> but often not enough!

Must ship your tech with a game

-> real-world relevance important!

Always profile min spec!

-> ideally on user devices

Investing in debug features always pays off

-> Shader-printf, shader primitive drawing, shader-asserts

Adding major changes very late can work

-> But always have a fallback working solution!

Teamwork is key

-> Often more than the sum of its parts!

Thanks!

Please ask some questions!

Acknowledgements

The entire idTech team (alphabetical):

Allen Bogue, Billy Khan, Bogdan Coroi, Carson Fee, Dominik Lazarek, Ian Malerich, John Roberts, Jean Geffroy, Johan Donderwinkel, Mel-Frederic Fidorra, Oliver Fallows, Dr. Peeter Parna, Philip Hammer, Regan Carver, Seth Hawkins, Stefan Pientka, Thorsten Lange, Tiago Sousa and Yixin Wang

id Software leadership (Marty Stratton, Hugo Martin)

Martin Fuller & the Microsoft ATG team

Our amazing art teams

Everyone else at id Software, ZeniMax and Xbox

References

[McLaren22] “Adventures with Deferred Texturing in Horizon Forbidden West.” James McLaren GDC 2022

[Fuller22a] “Variable Rate Compute Shaders - Halving Deferred Lighting Time”, Martin Fuller, Microsoft Game Dev YouTube Channel, 2022 https://www.youtube.com/watch?v=Sswuj7BFjGo

[Fuller22b] “Variable Rate Shading Update Xbox Series X|S”, Martin Fuller, Philip Hammer, Christopher Wallis Microsoft Game Dev YouTube Channel, 2022, https://www.youtube.com/watch?v=pPyN9r5QNbs

[Hammer25] "Variable-Rate Compute Shaders in DOOM: The Dark Ages", Graphics Programming Conference 2025

[Wihlidal24] “Nanite GPU-Driven Materials.”, Graham Wihlidal, GDC 2024, 2024. https://gdcvault.com/play/1034407/Nanite-GPU-Driven

[Karis21] “Nanite: A Deep Dive” Brian Karis, Rune Stubbe, Graham Wihlidal, Siggraph 2021 https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf

[Doghramachi17] “Deferred+: Next-Gen Culling and Rendering for Dawn Engine” Hawar Doghramachi, Jean-Normand Bucci. GPU Zen: Advanced Rendering Techniques, edited by Wolfgang Engel, 2017.

[Geffroy20] “Rendering the Hellscape of DOOM Eternal”, Jean Geffroy, Axel Gneiting, Yixin Wang, Siggraph 2020 https://advances.realtimerendering.com/s2020/RenderingDoomEternal.pdf

[Hecker95] Chris Hecker. “Perspective Texture Mapping.” Game Developer Magazine April/May 1995, 1995. https://www.chrishecker.com/images/4/41/Gdmtex1.pdf


Bonus Slides
id
id TECH8
DOOM THE DARK AGES

Detailed Performance Results

	FWD+ (w/ HW VRS)	Deferred	Deferred /w VRCS
Pixel-commands & dispatch args (async)		0.35	0.35
Attribute Interpolation (async)		0.43	0.43
VRCS Shading Rate (async)			0.41
Deferred Texturing		4.20	3.79
G-Buffer Update		1.10	1.00
Deferred Lighting		1.70	1.32
Forward+ Opaque Pass	9.43		
Overall (no async)	9.43	7,78	7.30
Overall (w/ async*)	9.43	7.5 **(-20%)**	7.0 **(-25%)**

*Overall times w/ async derived from overall GPU frame time diffs

„Siege“ Scene
Xbox Series X - 2560x1440 – no resolution scaling

Memory Results

- Memory cost relatively high atm
- Some memory aliased with other techniques
  (e.g. strands hair simulation)
- Aliasing still a bit under-used in idTech8
- Memory scales a lot with resolution
  - More positive than negative effect: Weaker platforms (consoles) usually run at lower resolutions, reducing memory overhead

		3840x2160	2560x1440
Deferred	Visibility Buffer	63.75	30
	Dispatch Buffers	20.11	10.3
	Pixel Commands + Counter Buffers (aliased w/ hair)	57 (-32 strands hair)	26 (-26 strands hair)
	Barycentric Derivatives	63.75	30
	Barycentrics	31.88	15
	Tangent Frame	63.75	30
	Overall Deferred	268.24	115.3
VRCS	Shading rate Images VRS + VRCS	2.3	1.04
	Luma Images (VRS + VRCS)	16	7.5
	VRCS Copy Bits	7.97	3.75
	VRCS Coordinates	29.03	12.3
	Overall VRCS	55.24	24.59
Tile Classification	TC Buffers + Indirect args	7	3.5
	Overall	**330.48**	**143.39**

All results in MiB; includes alignment / padding