RENDERING THE HELLSCAPE OF

Jean Geffroy
Principal Engine Programmer

Axel Gneiting
Principal Engine Programmer

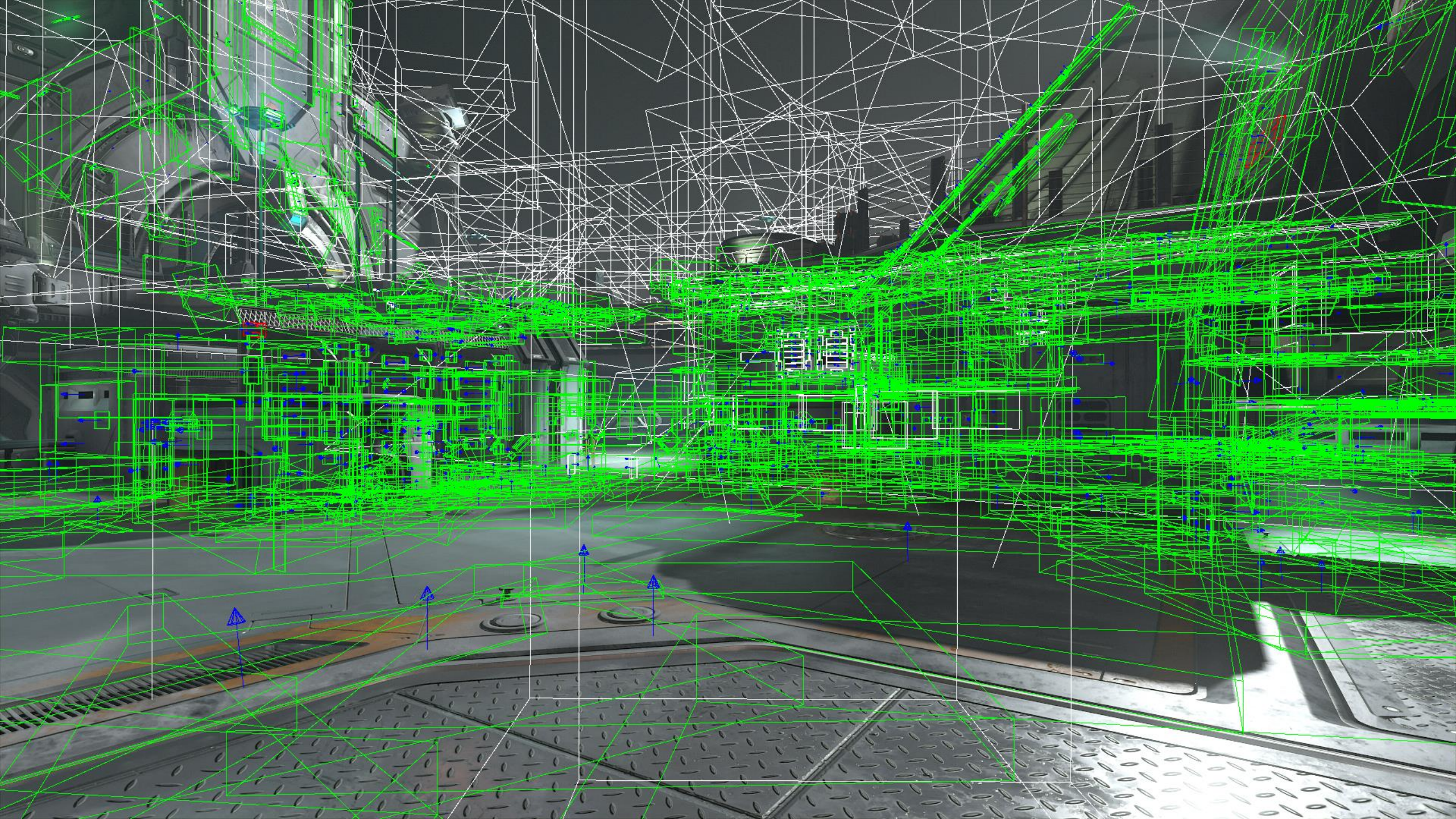
Yixin Wang
Engine Programmer

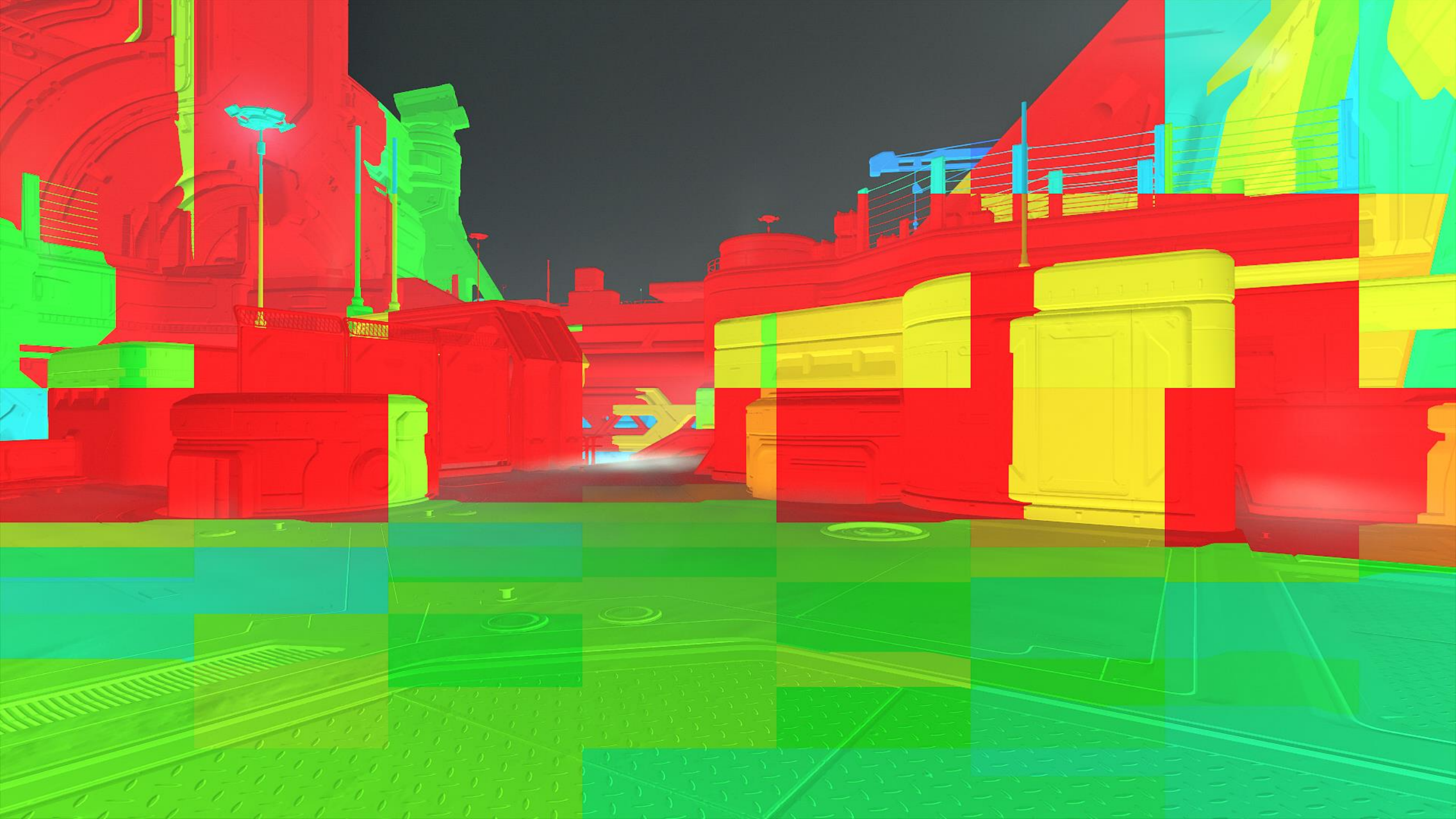
IDTECH 7

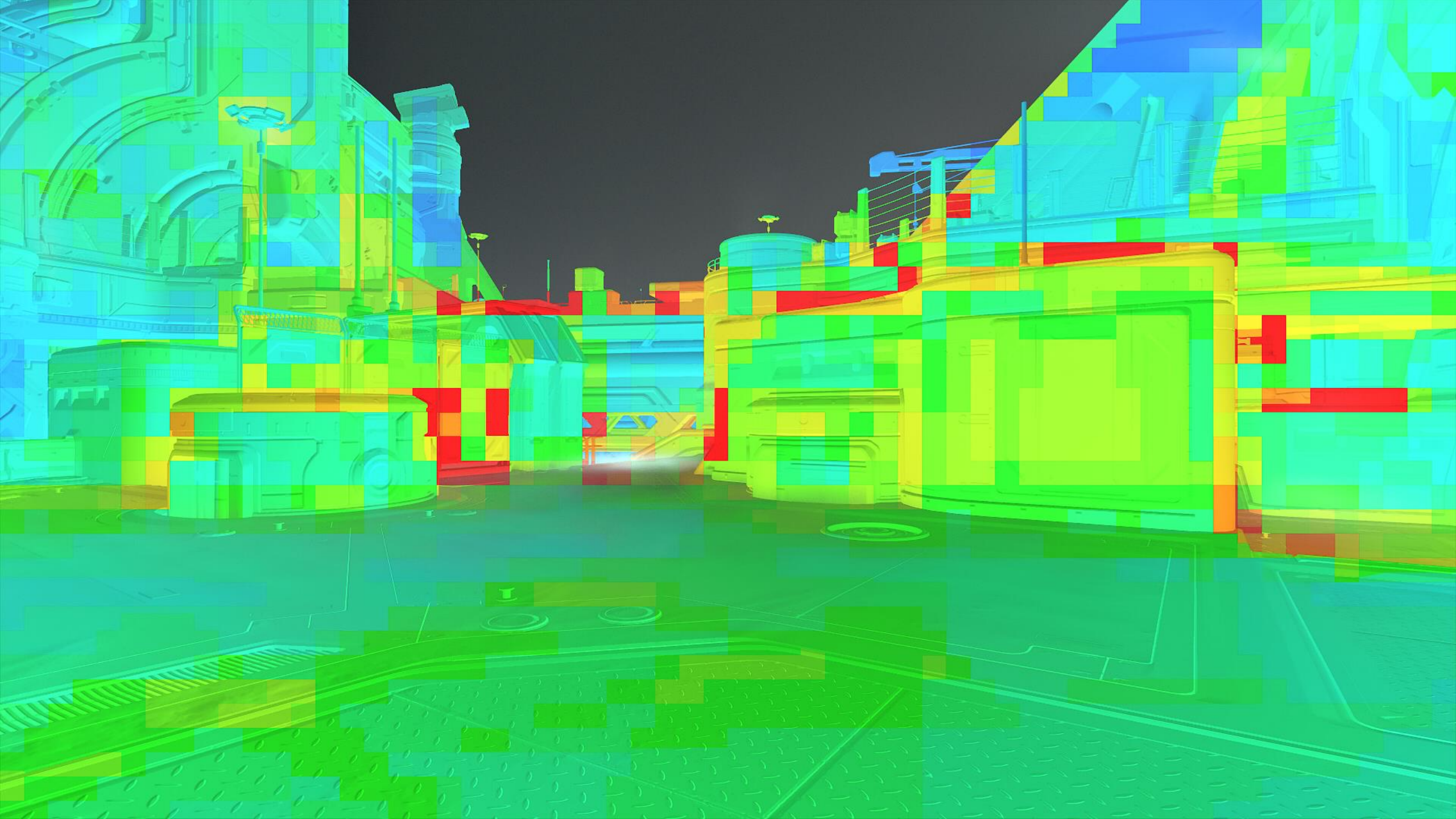
- Fully forward rendered
- Still few uber shaders variations everything
- Larger levels and more complex environments
- More streaming
- Low level API on all platforms
- Still 60FPS on consoles at same resolutions

HYBRID BINNING

- Used Clustered Binning in Doom [Olson12]
- Challenges
  - Larger scenes in Doom Eternal
  - Distant small lights and decals end up in single big cluster
  - Enemies have light rigs
  - In fights more dynamic volumes for projectiles and impact decals
  - Artists wanted to place more lights and decals
  - Wanted to move to GPU culling to reduce CPU load
- New hybrid of cluster and tile binning





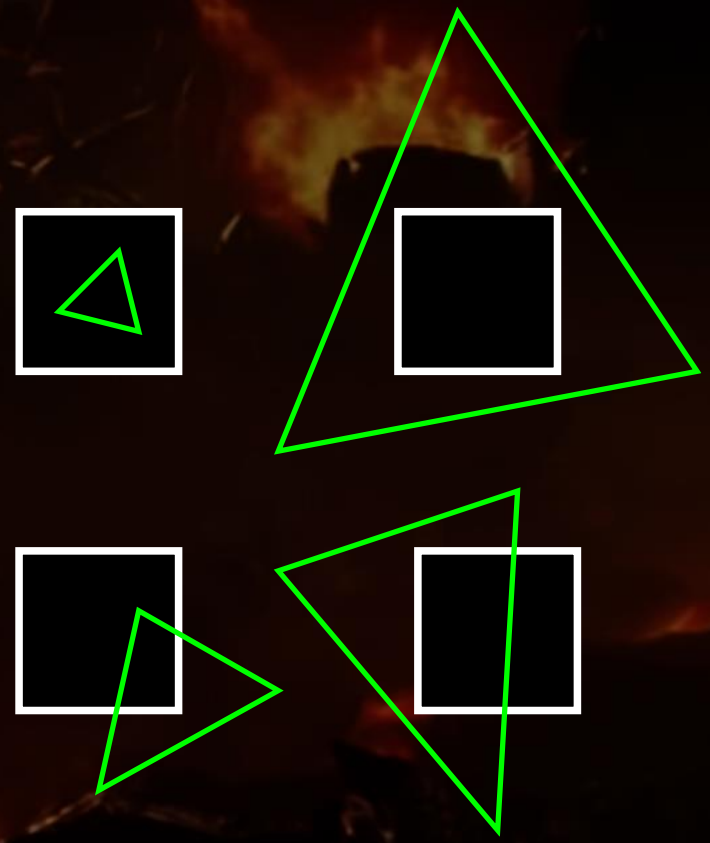


HYBRID BINNING

- Inspired by “Improved Culling for Tiled and Clustered Rendering” [Drobot2017]
- Problem: Thousands of lights or decals in some scenes
- Hardware raster tile binning inefficient due to state changes
- Very tight budget (<500us)

COMPUTE SHADER RASTERIZATION

- For simplicity we only bin hexahedra
- Low resolution
- Needs to be conservative
  - Many edge cases
  - Artists will find them all
- Reverse float depth for precision
- Binned rasterization [Abrash2009]
  1. Setup & Cull
  2. Coarse raster
  3. Fine raster
  4. Resolve bitfields to lists

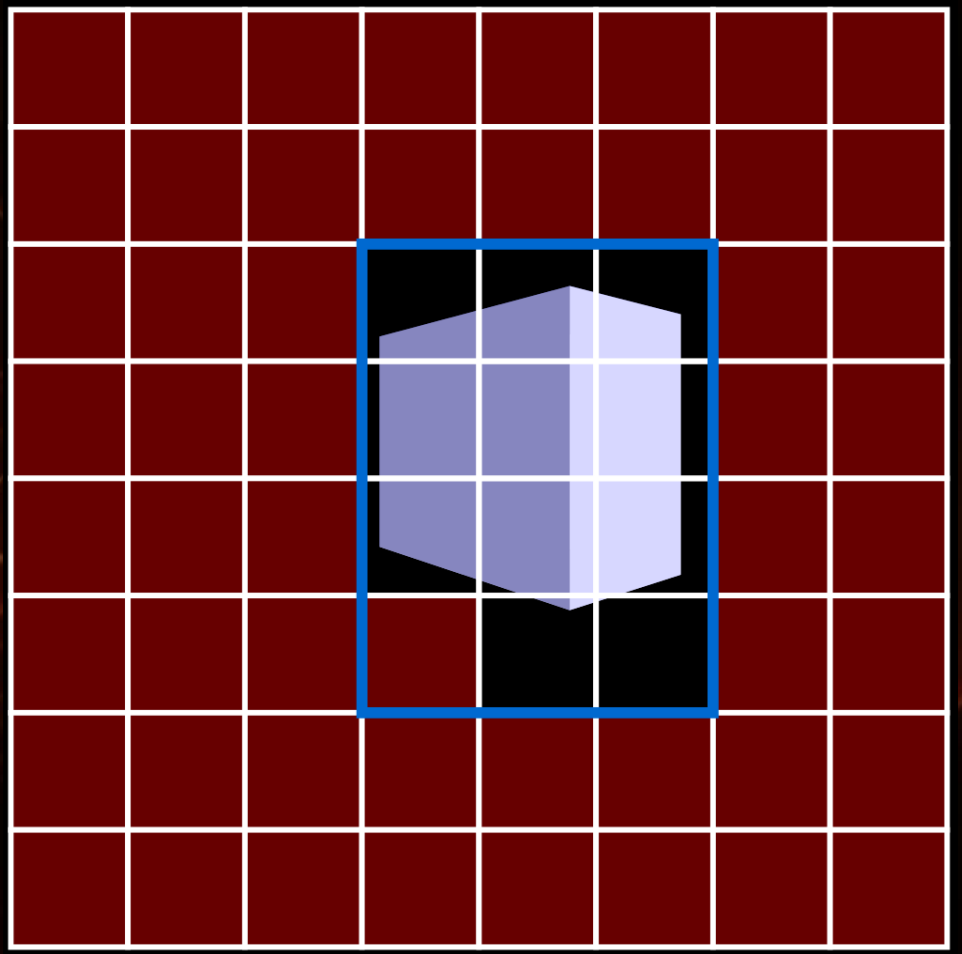


SETUP AND CULL

- Culls hexahedra against near plane
- Emits packed vertices together with side and edge indices
- Classifies front/back facing for edges and vertices
- New side at near plane removed
- Computes screen space tile bounds
- Produces work for coarse raster

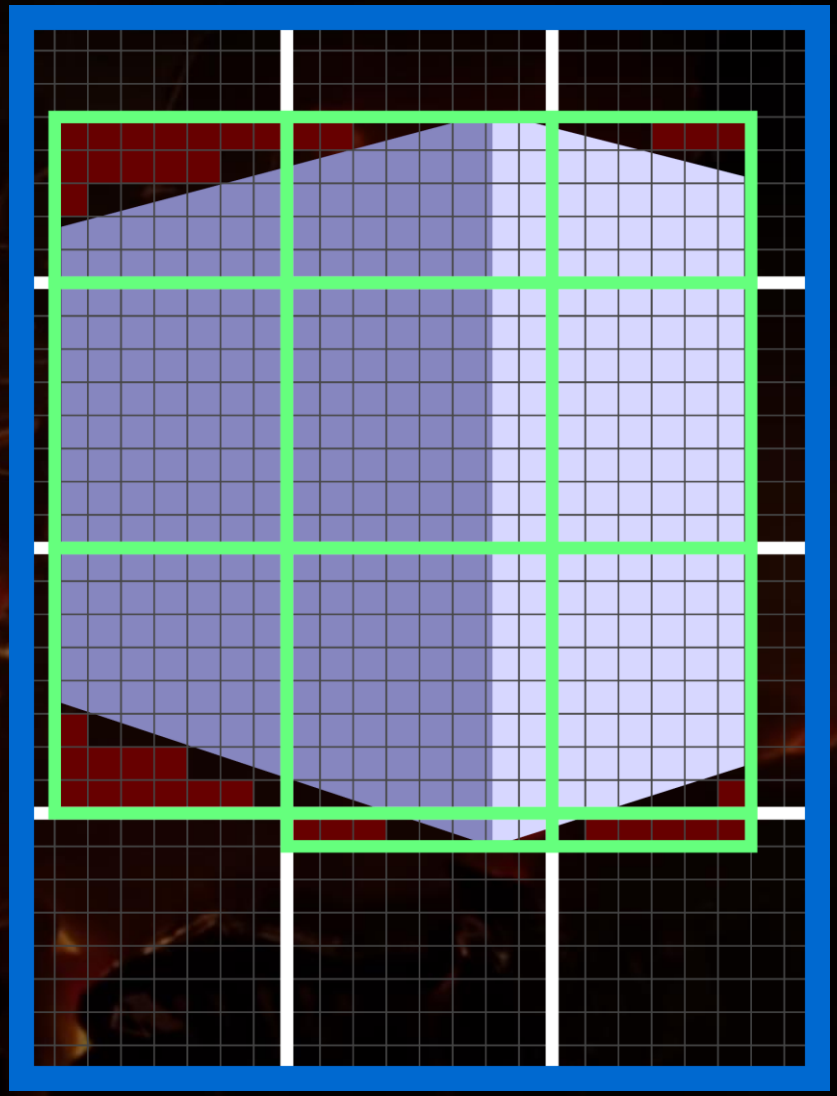
WORK DISTRIBUTION

- Want to run rasterization thread for each potentially covered tile based on rectangular screen space coverage
- Running `[8,8,1]` thread groups for 64 tiles with each thread looping over the binned hexahedra for the tile is too slow
- Results in very bad utilization
  - Up to 63 idle threads
- Instead emit one thread for each potentially covered tile



COARSE RASTER

- 256x256 pixels per coarse tile
- Also bins into clusters
  - Same depth distribution as in [Sousa2016]
- Already tighter fit than CPU cluster setup
- Produces work for fine raster
  - Only emits work for coarse tile that intersect hexahedra
  - One thread for each potentially covered fine tile
  - Each coarse tile has its own work list and screen space bounds are determined per tile



FINE RASTER

- 32x32 pixels per fine tile
- Rejection based on conservative min/max depth buffer downsample
- CS computes min/max depth value for hexahedra per tile
- Refinement step for point lights culling tile frustum against sphere

RESOLVE BITFIELDS TO LISTS

- Resolve CS with one thread group per tile/cluster
- 64 bits checked in one thread group, one bit per thread
- `(maxLights+63)/64` iterations per tile/cluster
- Uses subgroup ops for compaction

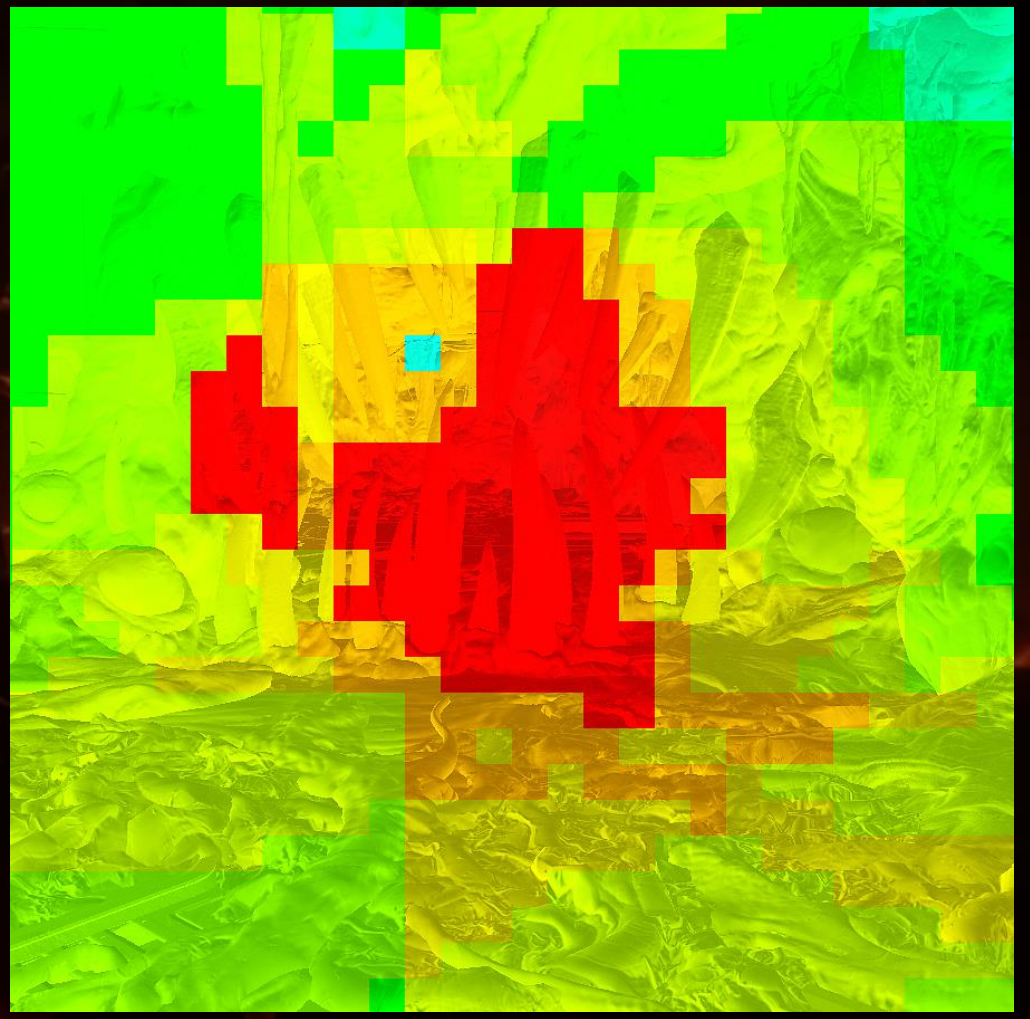
```
uint totalLightCount = 0;
for (int i = 0; i < numIterations; ++i) {
  const uint lightIndex = (64 * i) + gl_localInvocationID.x;
  const uint tileIndex = gl_localInvocationID.y;
  bool lightBitSet = fetchBitFromBitfield(tileIndex, lightIndex);
  uvec4 subgroupMask = subgroupBallot(lightBitSet);
  uint lightCount = subgroupBallotBitCount(subgroupMask);
  if (lightBitSet) {
    // Computes bitfield prefix sum for all threads < this thread idx
    uint listOffset = subgroupBallotExclusiveBitCount(subgroupMask);
    listOffset += tileIndex * LIGHTS_PER_TILE;
    listBuffer[listOffset + totalLightCount] = lightIndex;
  }
  totalLightCount += lightCount;
}
if (subgroupElect())
  countBuffer[tileIndex] = totalLightCount;
}
```

LIGHT LIST SELECTION

- Fine tiles are bad if there are large depth discontinuities
- Each fragment invocation checks *both* its cluster and tile and selects list with fewer entries
- Never worse than clustered

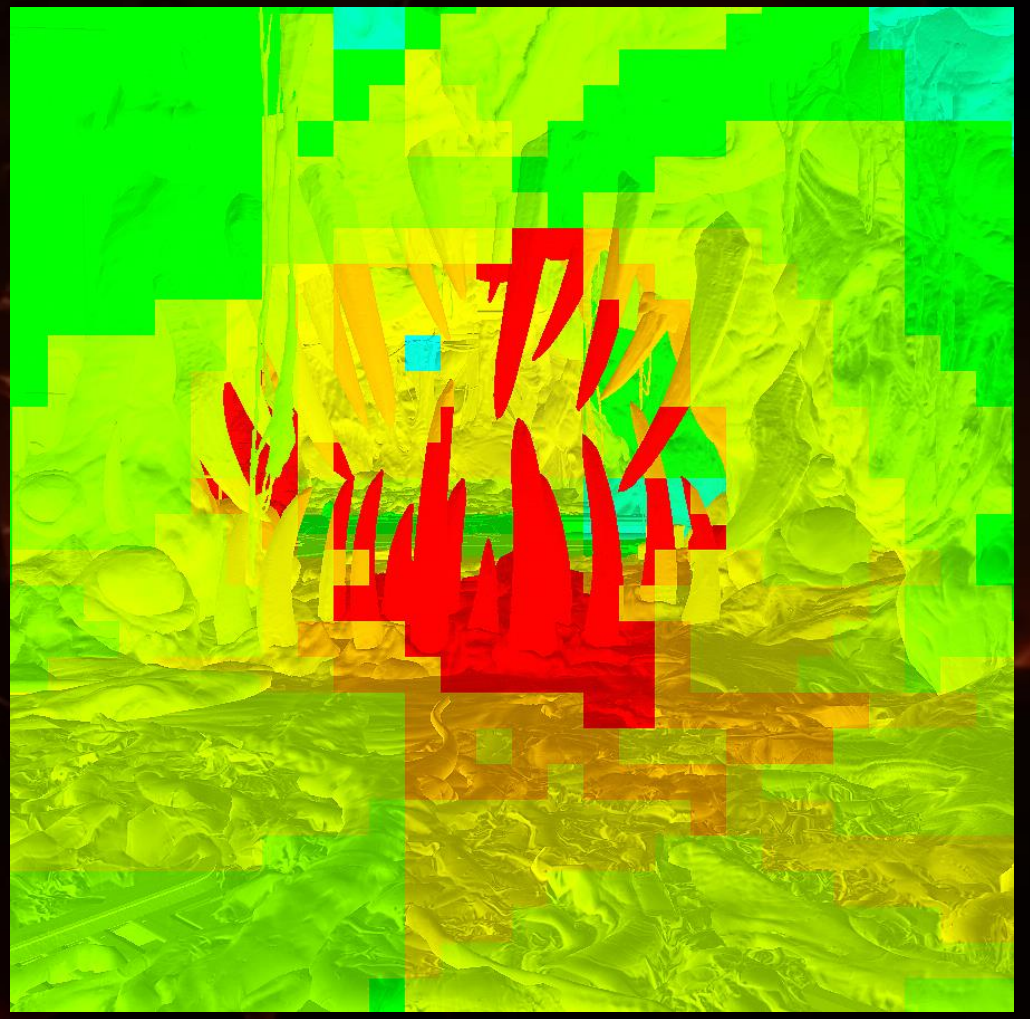
LIGHT LIST SELECTION

- Fine tiles are bad if there are large depth discontinuities
- Each fragment invocation checks *both* its cluster and tile and selects list with fewer entries
- Never worse than clustered



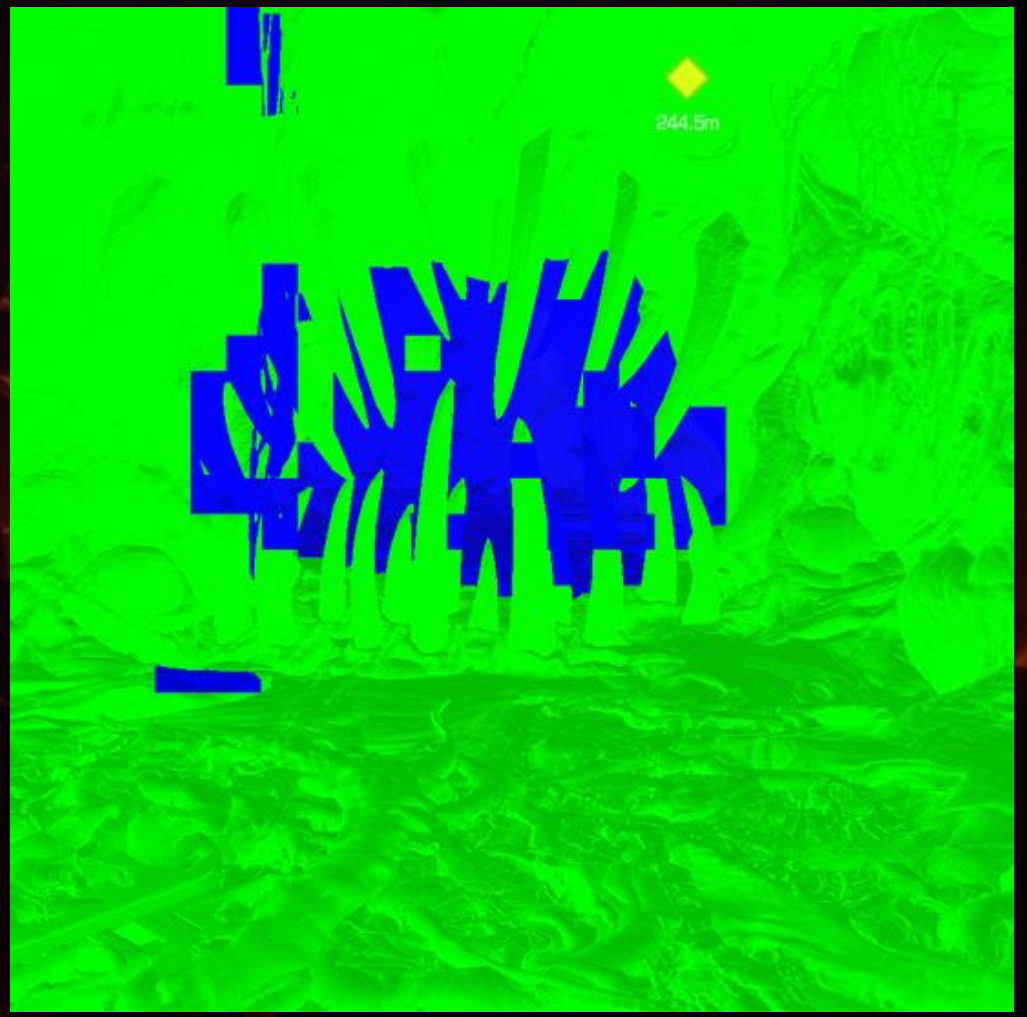
LIGHT LIST SELECTION

- Fine tiles are bad if there are large depth discontinuities
- Each fragment invocation checks *both* its cluster and tile and selects list with fewer entries
- Never worse than clustered



LIGHT LIST SELECTION

- Fine tiles are bad if there are large depth discontinuities
- Each fragment invocation checks *both* its cluster and tile and selects list with fewer entries
- Never worse than clustered

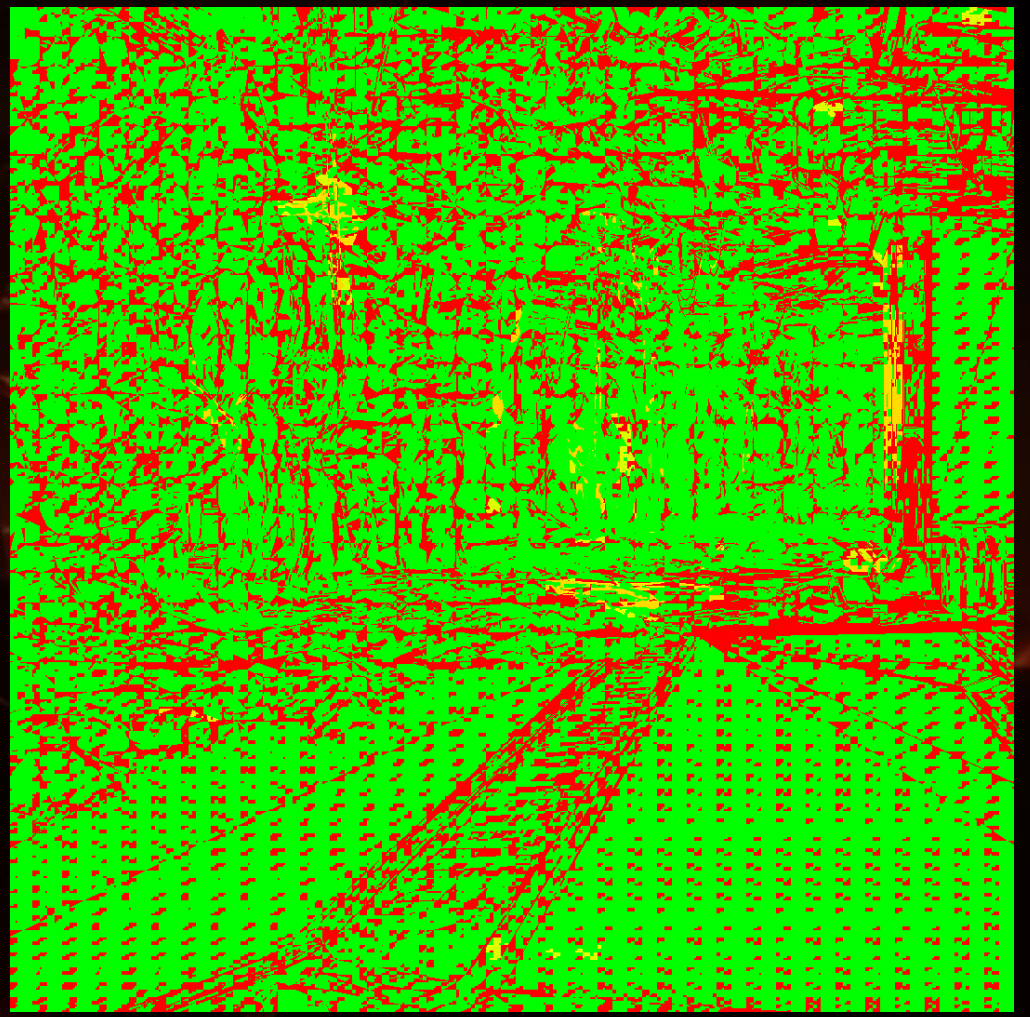


HASH SCALARIZATION

- Want to use scalar reads for light data [Sousa2016]
- Problem: Binning now has much finer granularity in screen space
- Scalarization based on cluster/tile index insufficient

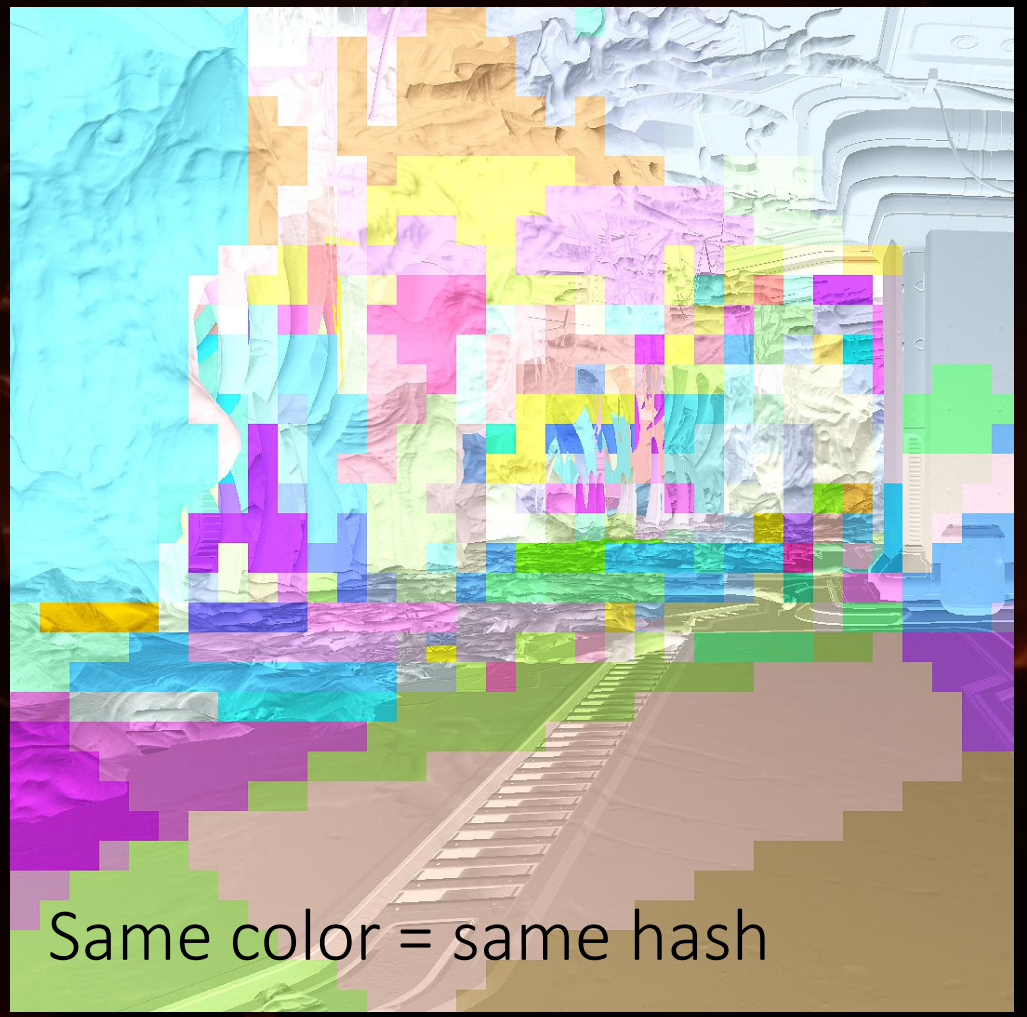
HASH SCALARIZATION

- Want to use scalar reads for light data [Sousa2016]
- Problem: Binning now has much finer granularity in screen space
- Scalarization based on cluster/tile index insufficient



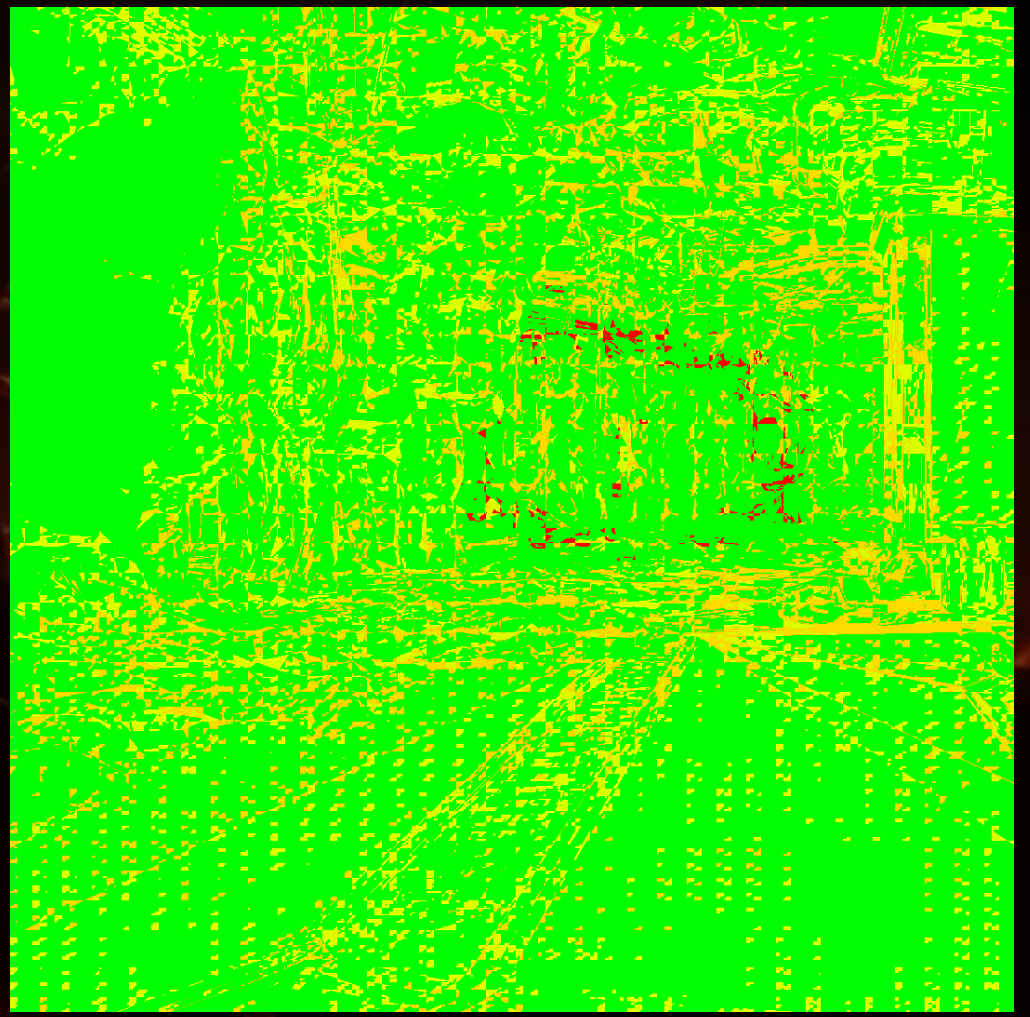
HASH SCALARIZATION

- Resolve CS computes hash of light indices for each list
- Store 24 bits of the hash + 8 bits num lights per tile/cluster
- Check for uniformity based on hash value
- Even works if fragment thread group uses cluster and tile lists



HASH SCALARIZATION

- Resolve CS computes hash of light indices for each list
- Store 24 bits of the hash + 8 bits num lights per tile/cluster
- Check for uniformity based on hash value
- Even works if fragment thread group uses cluster and tile lists



USE FOR GAMEPLAY

- Long but few threads cause GPU usage to be quite low
- Idle time can be filled with async overlap
- Adding more hexahedra to process almost free
- Idea: Run thousands of gameplay visibility queries through same code
- Changes depth test from “intersects opaque” to “in between camera and opaque”
- Fine raster tags bit fields read by CPU next frame

GEOMETRY DECALS

- Artists wanted small decals defined by geometry
- Part of authored mesh
- Challenges
  - Almost no additional frame budget
  - Forward renderer: Can’t blend on top of G-Buffer
  - G-Buffer blending too slow even if we switched to deferred

GEOMETRY DECALS

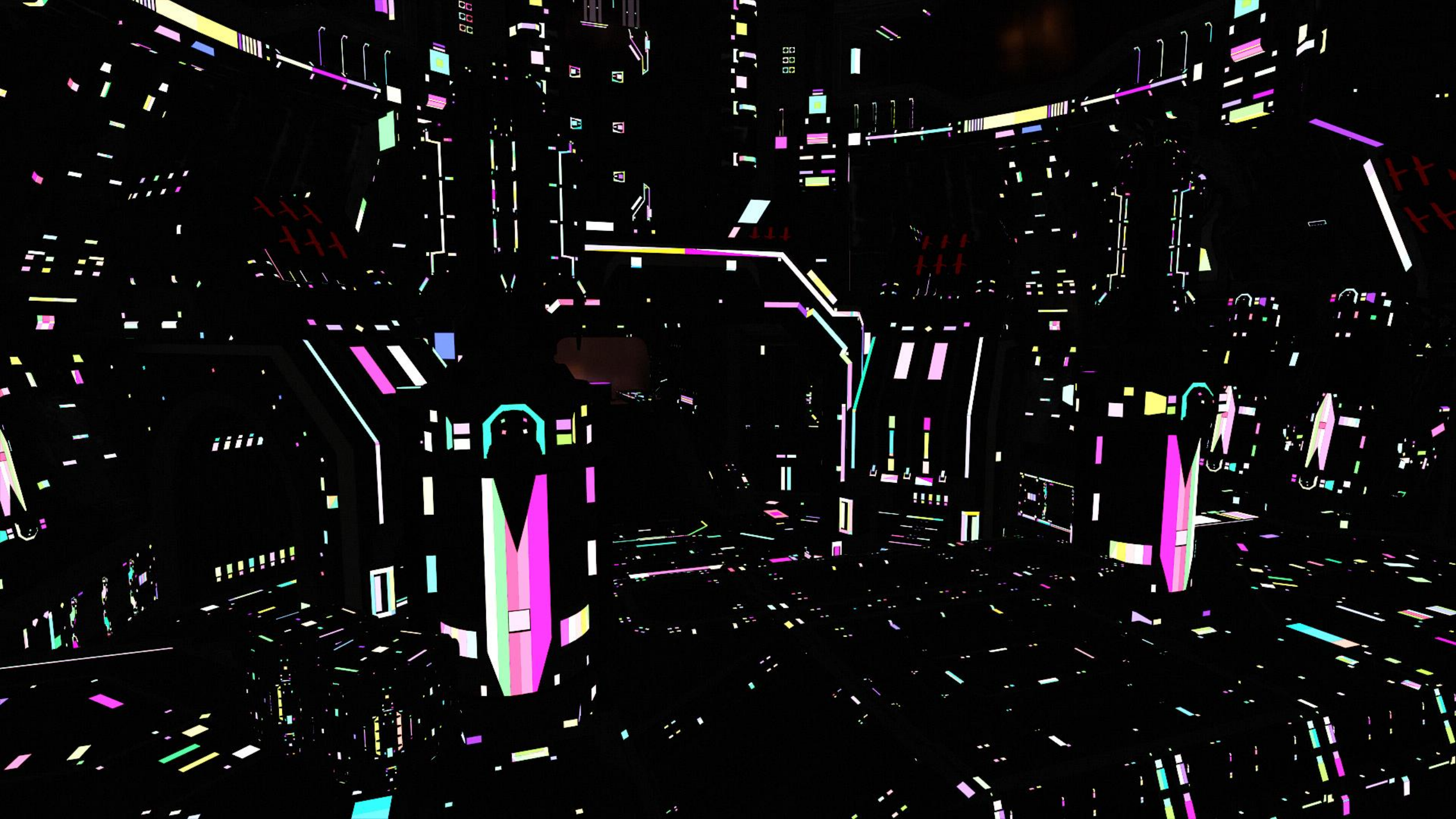
- Artists wanted small decals defined by geometry
- Part of authored mesh
- Challenges
  - Almost no additional frame budget
  - Forward renderer: Can’t blend on top of G-Buffer
  - G-Buffer blending too slow even if we switched to deferred

GEOMETRY DECALS

- Projection matrix from mesh UVs
- World space -> texture space
- $\mathbb{R}^{2x4}$ matrix per projection
  - 8 floats = 32 bytes
- On disk we store object -> texture space
  - Multiplied with model matrix gives world -> texture space matrix
- Each instance needs memory for its geo decals

GEOMETRY DECALS

- Render indices into R8 buffer
  - Post depth pass
  - Greater/equal depth compare
  - Depth bias to avoid z fighting
  - Decals need to be coplanar
  - ~50us for this pass
- Index per sub mesh projection list in fragment shader
  - List bound to instance descriptor set
  - Arbitrary blending because no G-Buffer passthrough



LIMITATIONS

- Max 254 projections per sub mesh
- One decal might need multiple projections
- Can’t do curved geometry very well
- Need projection storage per instance
  - Currently 1M decals max per level
- Binned decals always on top
- Animated geometry projections calculated every frame

GEOMETRY CACHES

- Based on [Gneiting2014]
- Compiled from Alembic caches
- Improved compression
  - Predictive Frames
  - Hierarchical B-Frames
  - Forward and backward motion prediction
  - Oodle Kraken for bitstream compression
- Instances can share streamed data blocks if same cache
- Animated colors and UVs

AUTOMATIC SKINNING

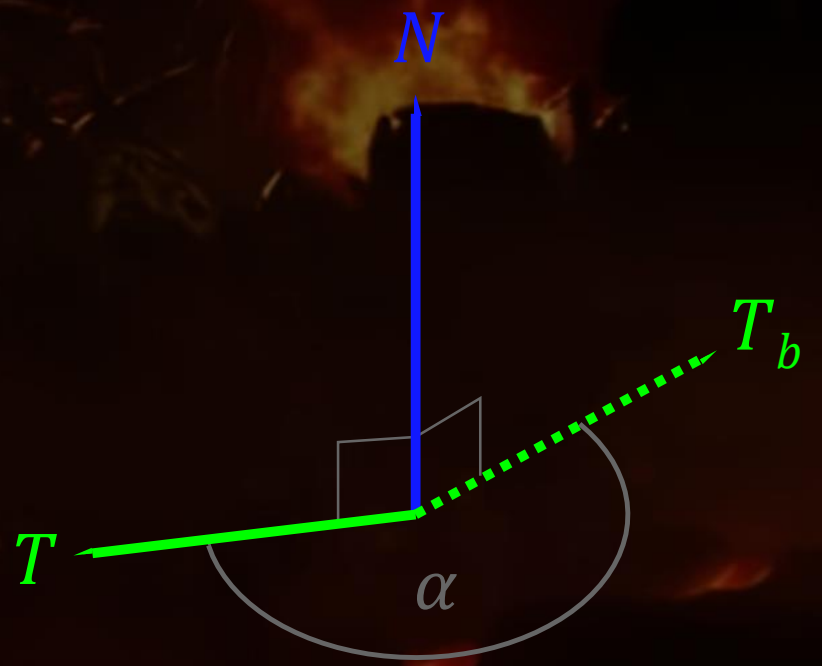
- Large data rate reduction for position stream [Kavan2010]
- Bone matrices quantized and compressed like other streams
- Only works well with some meshes
  - We still support vertex animation

3 BYTE TANGENT FRAMES

- Tangent frames are computed for each frame and streamed
- Storing two $\mathbb{R}^3$ vectors for normal and tangent is expensive
- Idea: Store only normal + rotation for tangent
  - Bitangent reconstructed with cross product
- 2 bytes for octahedron normal encoding [Meyer2010]
  - Good enough for vertex normals
- 1 byte for rotation

DECODE

- Need deterministic orthonormal vector to rotate for tangent
  - Naïve cross product of normal with e.g. $(1, 0, 0)^T$ causes singularities
  - $T_b = \begin{cases} |(-N_y, Nx, 0)|^T & if\ |N_x| > |N_z| \\ |(0, -N_z, N_y)|^T & else \end{cases}$
- Rodrigues' Rotation Formula [Euler1770] gives Tangent
  - $T = Tb \cos\alpha + (N \times T_b) \sin\alpha$
  - Last term cancels out because vectors are orthonormal

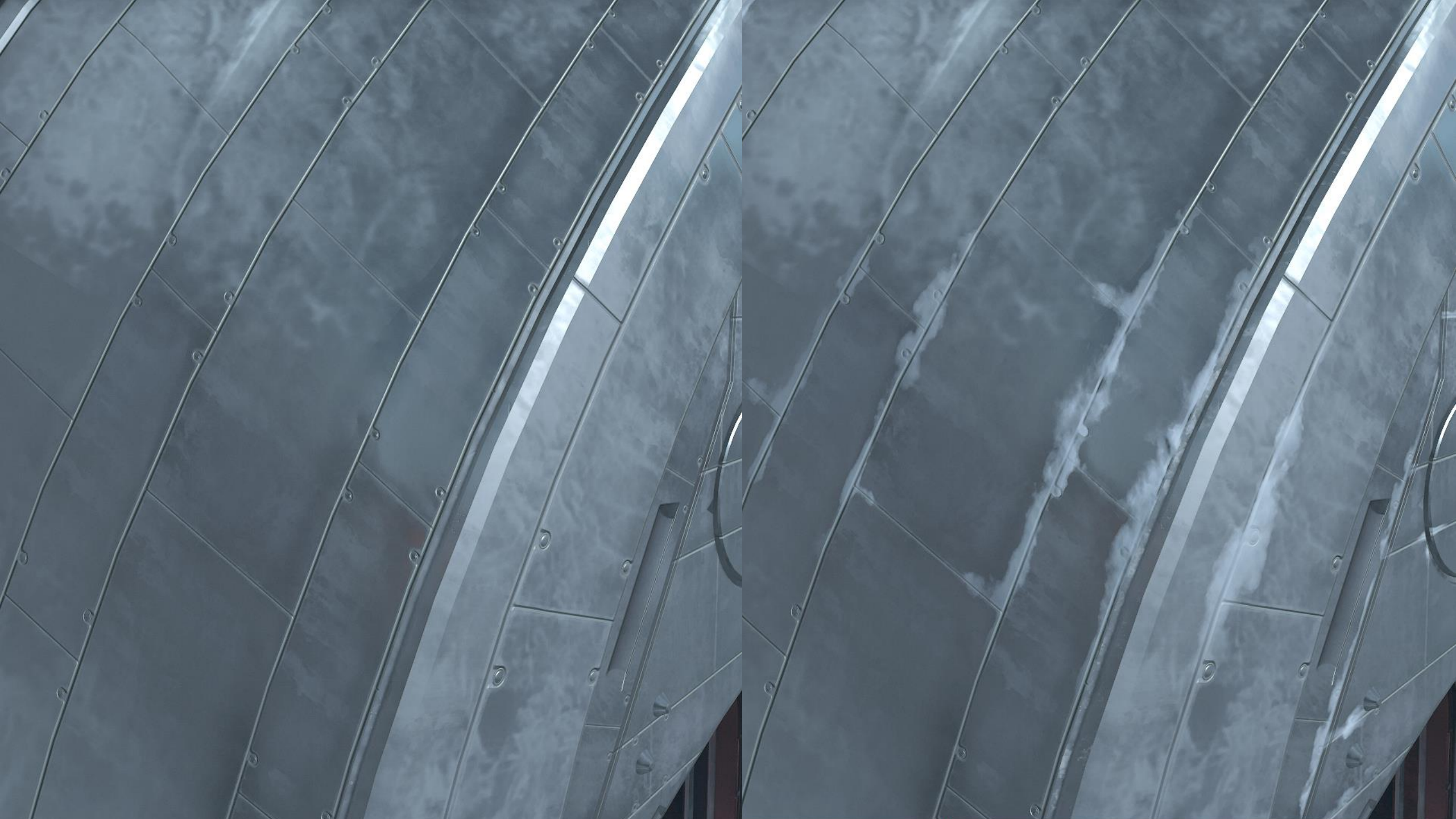


MATERIAL BLENDING

- First step to add detail to the world
  - Precedes world decaling pass
  - Part of the prop-building process (e.g. grime)
  - Can also be environment-specific (e.g. snow)
- Blending of multiple full material stacks
- Mostly vertex painted
- Texture blend weight maps for organic assets and characters
  - Required granularity may not match tessellation level
- Blend weights modulated by material heightmaps
  - High frequency detail
  - Embrace physical features of underlying materials

VERTEX PAINTING

- Built-in painting tool
- 4 materials per asset, 2 materials per triangle
- LOD0 geometry shared by all painted instances
- Lower LODs uniquely generated based on paint job
  - Required to mitigate overly noticeable LOD transitions
  - Memory overhead manageable thanks to limited paint permutations



GPU TRIANGLE CULLING

- Motivation
  - 3 times the on-screen triangle count compared to Doom 2016
  - Early culling of triangles on the GPU in a compute prepass
  - Remove pressure of the HW graphics pipeline
- Backface, frustum & micro-triangle culling
  - Similar to "Optimizing the Graphics Pipeline with Compute" [Wihlidal2016]
- Occlusion culling with depth mip chain generated from Umbra's SW buffer
- Runs in addition to all standard CPU culling
- Removes about 70% of submitted triangles in a typical scene
  - Budget for 3M visible triangles after CPU culling and LOD selection
  - Only about 1M makes it all the way to the rasterizer

GPU GEOMETRY MERGING

- Motivation
  - Levels built with large amount of instantiated models, up to 15k in view
    - Individual meshes can be low-poly, especially at lower LODs
  - Lots of unused VS lanes and Command Processor overhead
    - Cascades into poor PS occupancy
  - Significant CPU cost to issue draw calls
    - Even more considering the use of a depth prepass
- Concept
  - Merge surviving triangles of multiple models in a single indirect index buffer
  - Combined with GPU triangle culling, guarantees close to 100% useful VS lanes
  - Single draw call for many models lowers CPU setup cost

GPU GEOMETRY MERGING - REQUIREMENTS

- Merged geometry has to use the same pipeline state
  - Works great with ubershaders!
  - Different materials are fine, only requirement is same PSO
- Relies on completely bindless GPU pipeline
- Globally indexable shader resources
  - Vertex data: Allocated from single pool, consistent with geometry streaming
  - Textures: Maintain global texture descriptor list as we stream them in/out
  - Uniform/Constant buffers: One pool per layout
    - Only 3 different indexable layouts in the game: world, characters and geom caches

IMPLEMENTATION - CULLING

- CPU scene traversal generates Geometry Sets
  - Up to 256 meshes sharing the same PSO per Geometry Set
  - Reserve indirect index buffer space for Geometry Set
- One culling dispatch per Geometry Set
- Culling shader outputs merged indirect index buffer for surviving tris
  - Combine VertexID and InstanceID into each 32-bit output index
  - Makes it possible to fetch instance data at render time
  - Correct behavior when it comes to HW vertex reuse

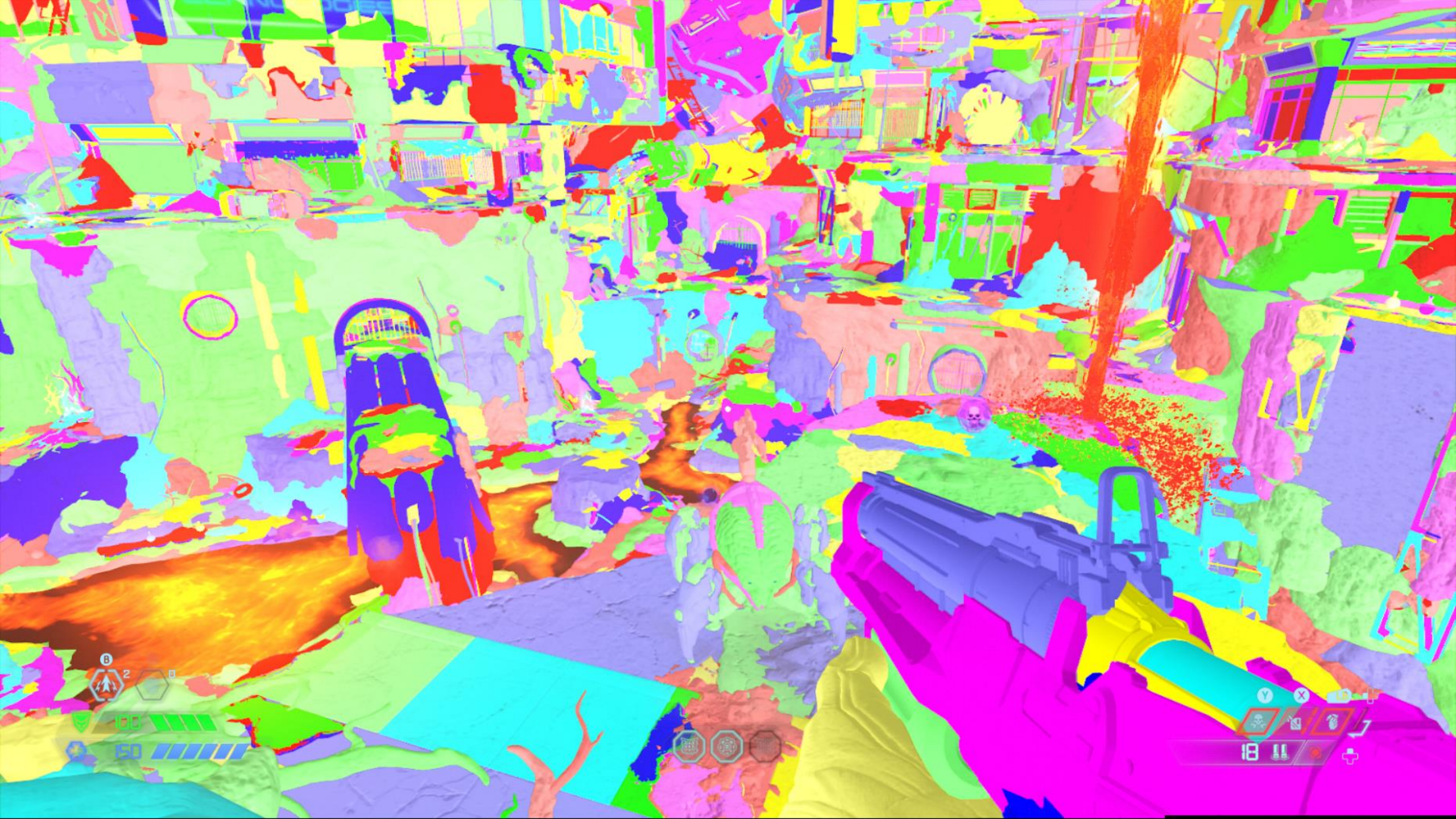
IMPLEMENTATION - RENDERING

- One indexed indirect draw call per Geometry Set
  - Effectively draws 256 meshes at once
- Extract VertexID and InstanceID from packed index value
- InstanceID used to resolve all instance properties
  - Base offset within vertex pool
    - Buffer fetches instead of vertex attributes
  - Instance uniform buffer offset
    - Includes material data and texture indices
- Scalarized iteration for divergent texture/buffer fetches
  - NonUniformResourceIndex
- “Mergeable” shader variation automatically generated by compiler

IMPLEMENTATION - DETAILS

- CPU code embarrassingly parallel
  - Prepare buffers for GPU culling/merging as we're traversing the scene
- Culling/Merging shader runs async, started before shadows
  - Synchronization done per geometry set, so possible culling/draw overlap
- Extra divergent indirection to fetch instance data at render time
  - Scary sounding
  - Added divergence mostly happens inside the VS or in "cold" code paths.
  - Practically imperceptible, largely compensated by other savings


B
2
100
150
Y
X
LB
18



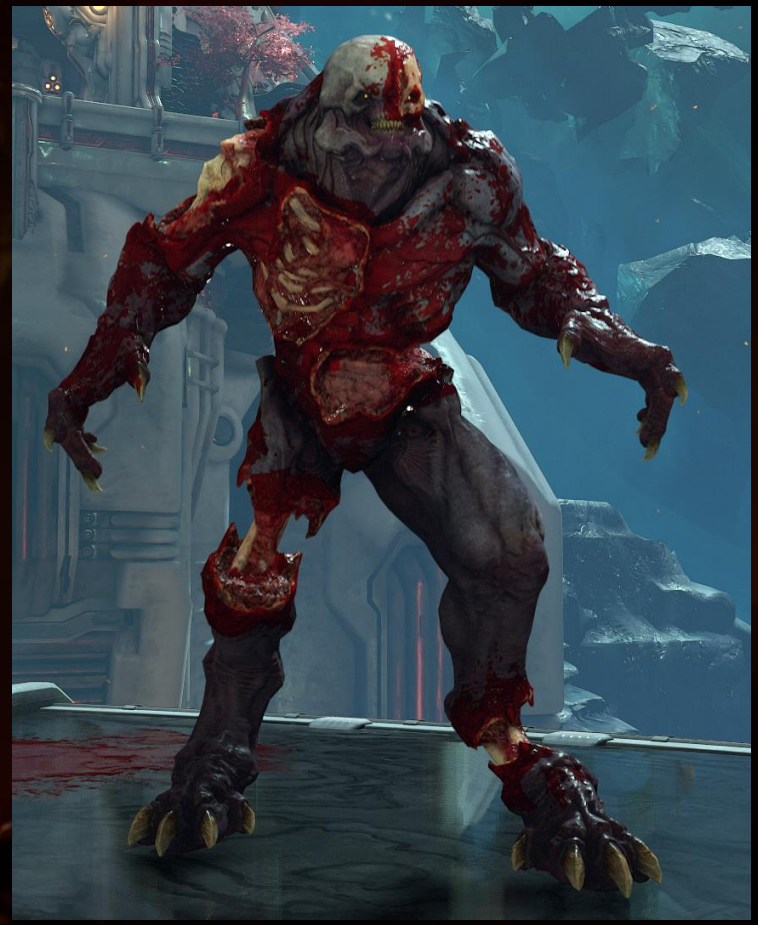

B
2
B
100
150
Y
X
LB
18

RESULTS

- Up to 5ms GPU savings in dense scenes with triangle culling+merging
  - Close to no waste in the VS
  - Mostly gets rid of the fixed-function bottlenecks
- Similar CPU savings
  - Reuse the same indirect index buffer for depth and opaque passes
  - Setup done during CPU visibility pass
  - Opaque/PreZ draw calls almost completely disappear

GORE SYSTEM

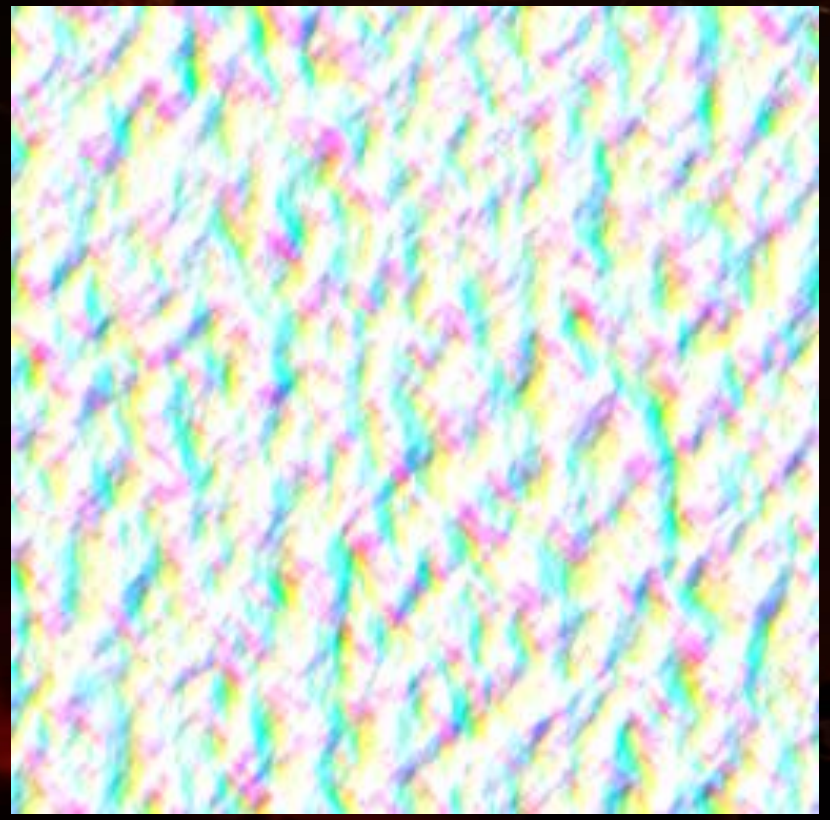
- Separate mesh for each wound
  - Authored alongside a clip and blood mask
  - Up to 12+ active wounds on 16 active enemies with multiple base materials
    - GPU Merging to the rescue once again!
- Instanced 1 byte/vertex wound buffer
  - Bit pack clip and blood weights
  - Update when a wound is applied in a CS
  - Procedural blood spawning on bullet impact
  - Much faster than testing all masks in the VS/PS



WATER RENDERING

DISPLACEMENT MAP

Generate displacement map and normal map for water surface according to *Simulating Ocean Water* [Tessendorf2001]

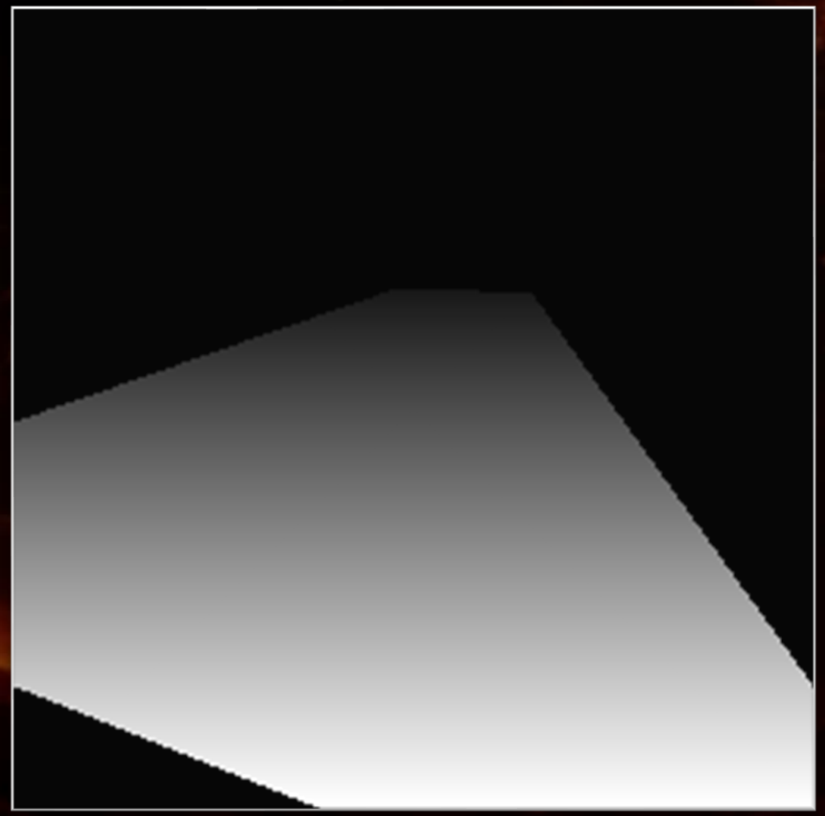


GENERATE WATER SURFACE MESH

- *Real-Time Water Rendering, Introducing the Projected Grid Concept* [Johanson2004]
- Project a screen-space grid onto the water planes
- Displace grid vertices in world-space
- Render water surfaces by rasterizing grid

GENERATE WATER SURFACE MESH

Render water planes depth image with same resolution as the screenspace grid (256x256)



Depth image (256x256)

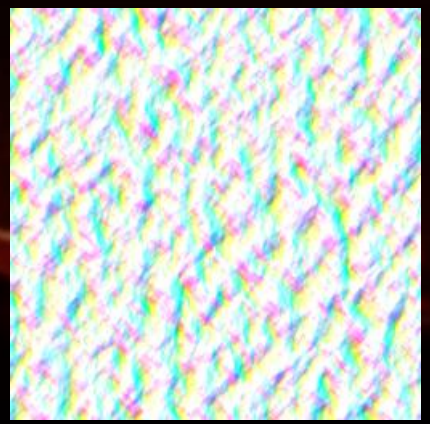
Scene for reference

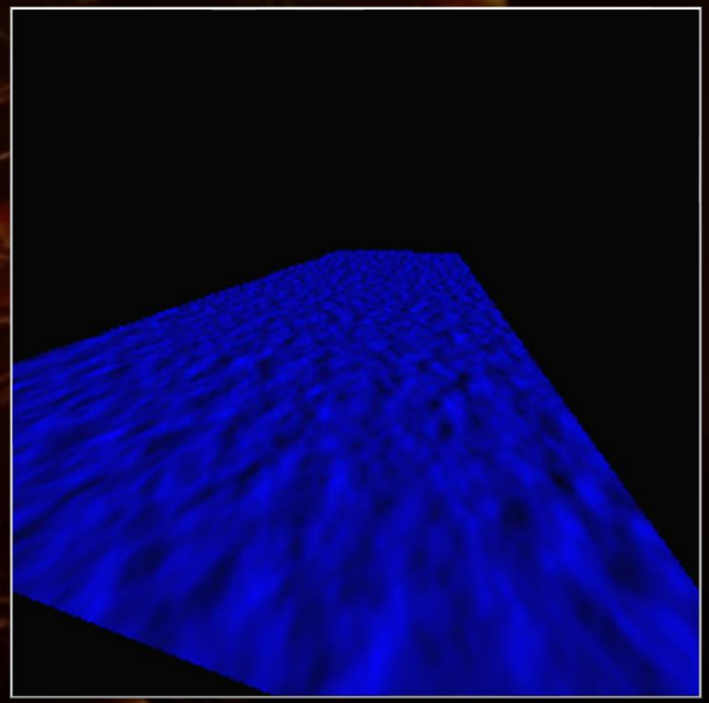
GENERATE WATER SURFACE MESH

Apply displacement map to depth image, output image of grid vertex world positions (referred to as “vertices image”)

Depth image (256x256)

Displacement map (256x256)

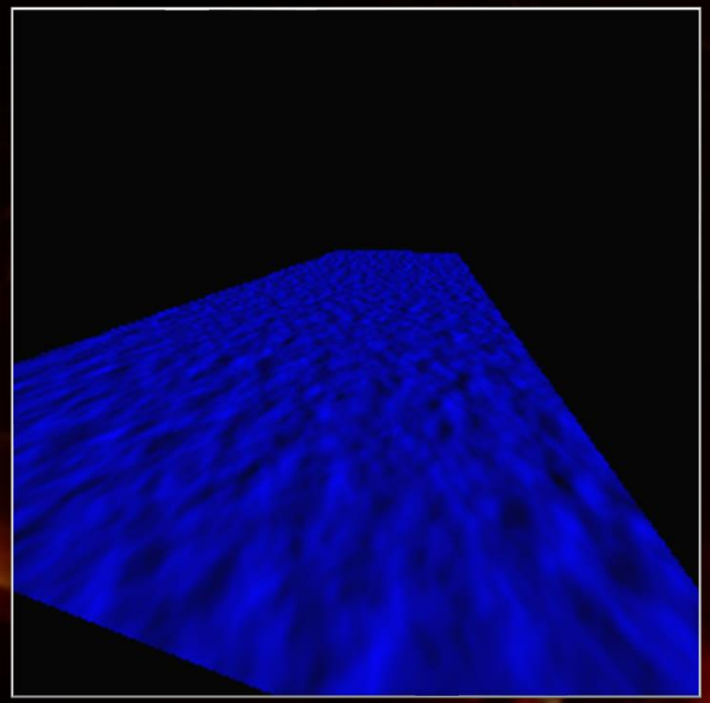




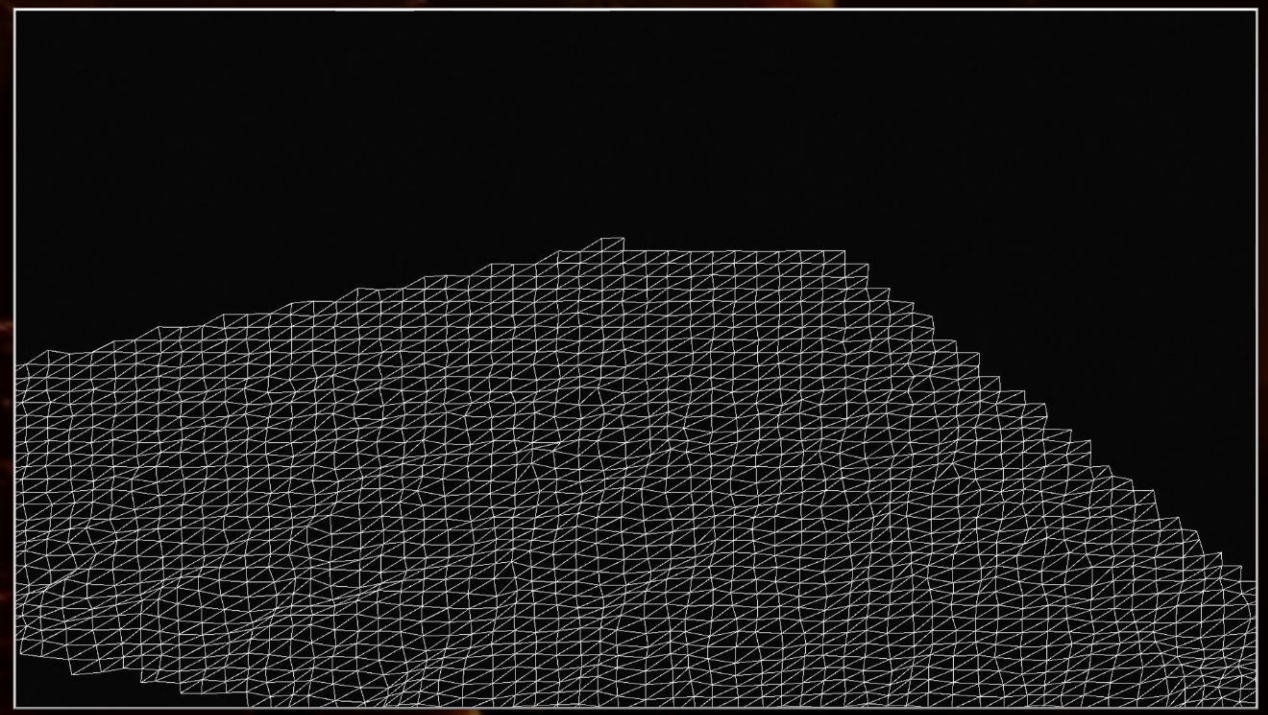
Vertices image (256x256)

GENERATE WATER SURFACE MESH

Rasterize screen-space grid mesh using vertices image, output G-buffer containing depth, normals



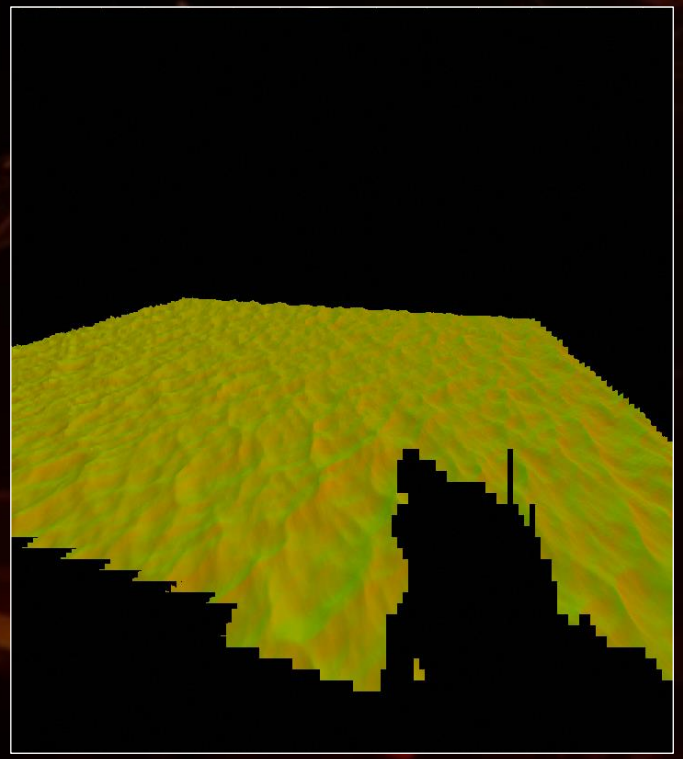
Vertices image (256x256)



Rasterized screenspace grid (grid is 256x256)

WATER G-BUFFER

Rasterize screen-space grid mesh using vertices image, output G-buffer containing depth, normals



G-Buffer (1/2 horizontal resolution)

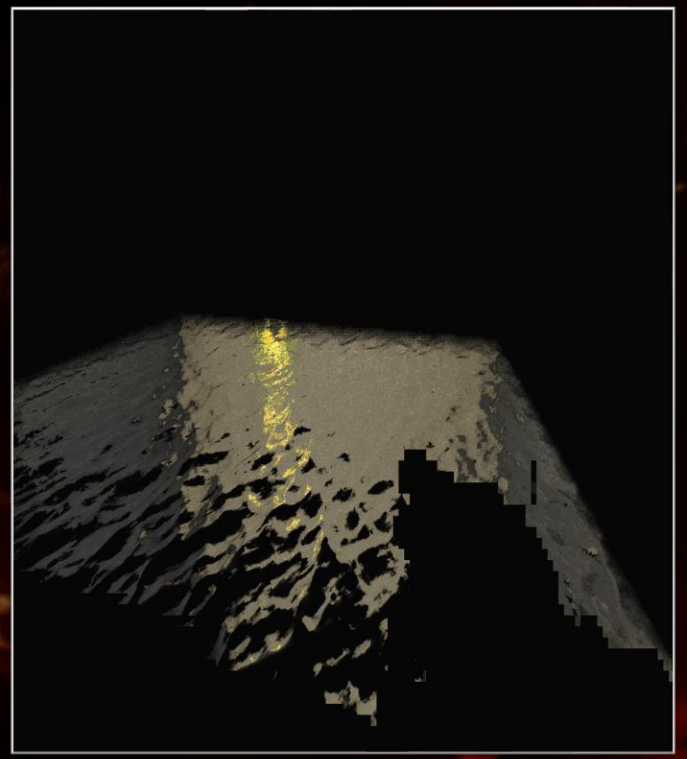
Scene for reference

VERTICES IMAGE TAA DISABLED

VERTICES IMAGE TAA ENABLED

WATER SURFACE SSR

Compute water surface SSR to SSR image using depth and normals from G-buffer

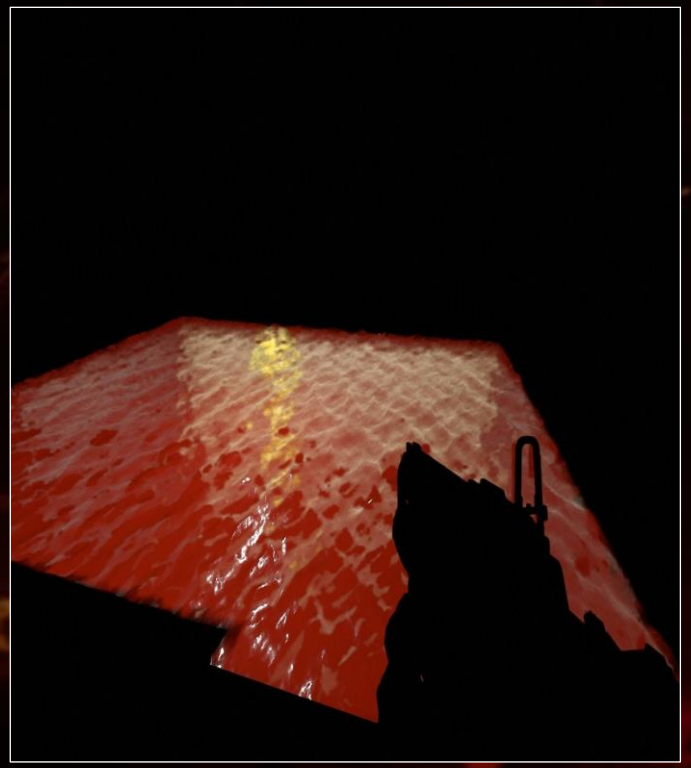


Water SSR (1/2 horizontal resolution)

Scene for reference

WATER SURFACE LIGHTING

Compute final water surface lighting



Water final (1/2 horizontal resolution)

Scene for reference

PREPARATION BEFORE RENDERING TRANSPARENT SURFACES

Split main color buffer into before- and after-water pixels

Main color buffer after rendering opaque surfaces

beforeWater

afterWater

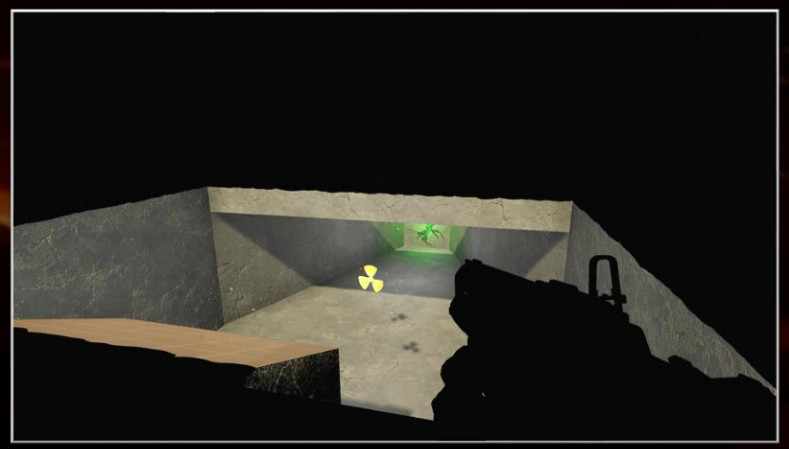
RENDER TRANSPARENT SURFACES

Render transparent surfaces to before- and after-water buffers instead of main color buffer. Compare fragment depth against water G-buffer depth

beforeWater

Render transparent surfaces

afterWater



RENDER TRANSPARENT SURFACES

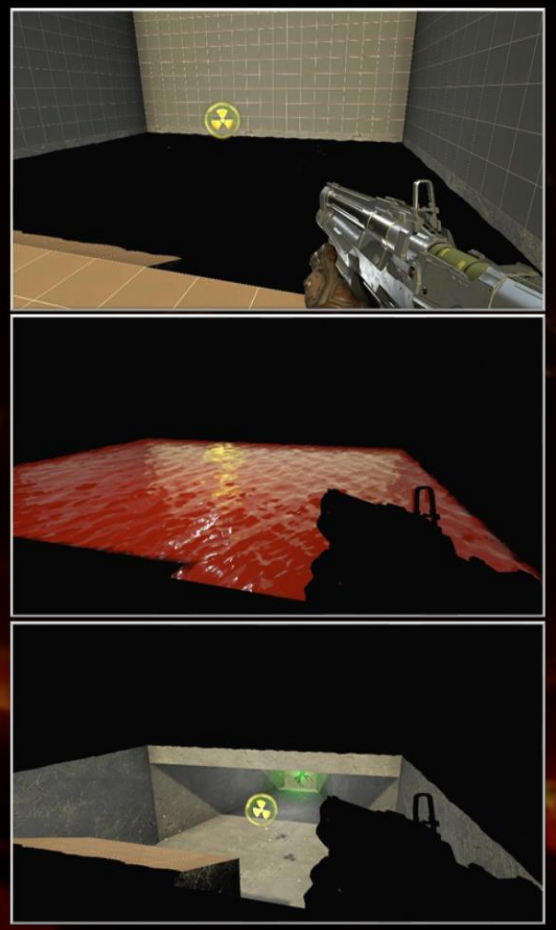
Need additional target to track before-water alpha

beforeWater

beforeWaterAlpha

FINAL COMPOSITE

Do final composite, with distorted texcoord lookup [Sousa2008] for after-water image



distorted texcoords

Final composite

WATER HITS SIMULATION

- *Fast Water Simulation for Games Using Height Fields* [Müller-Fischer2008]
- Simulated on grid centered around camera, top-down projection [Sousa2011]
  - Reproject texcoords when fetching previous-frame height field when camera moves
  - Snap grid position to multiple of grid cell size to avoid reprojection aliasing
  - One 512x512 simulation grid for all water planes
- Resulting heightfield is added as additional displacement to water vertices

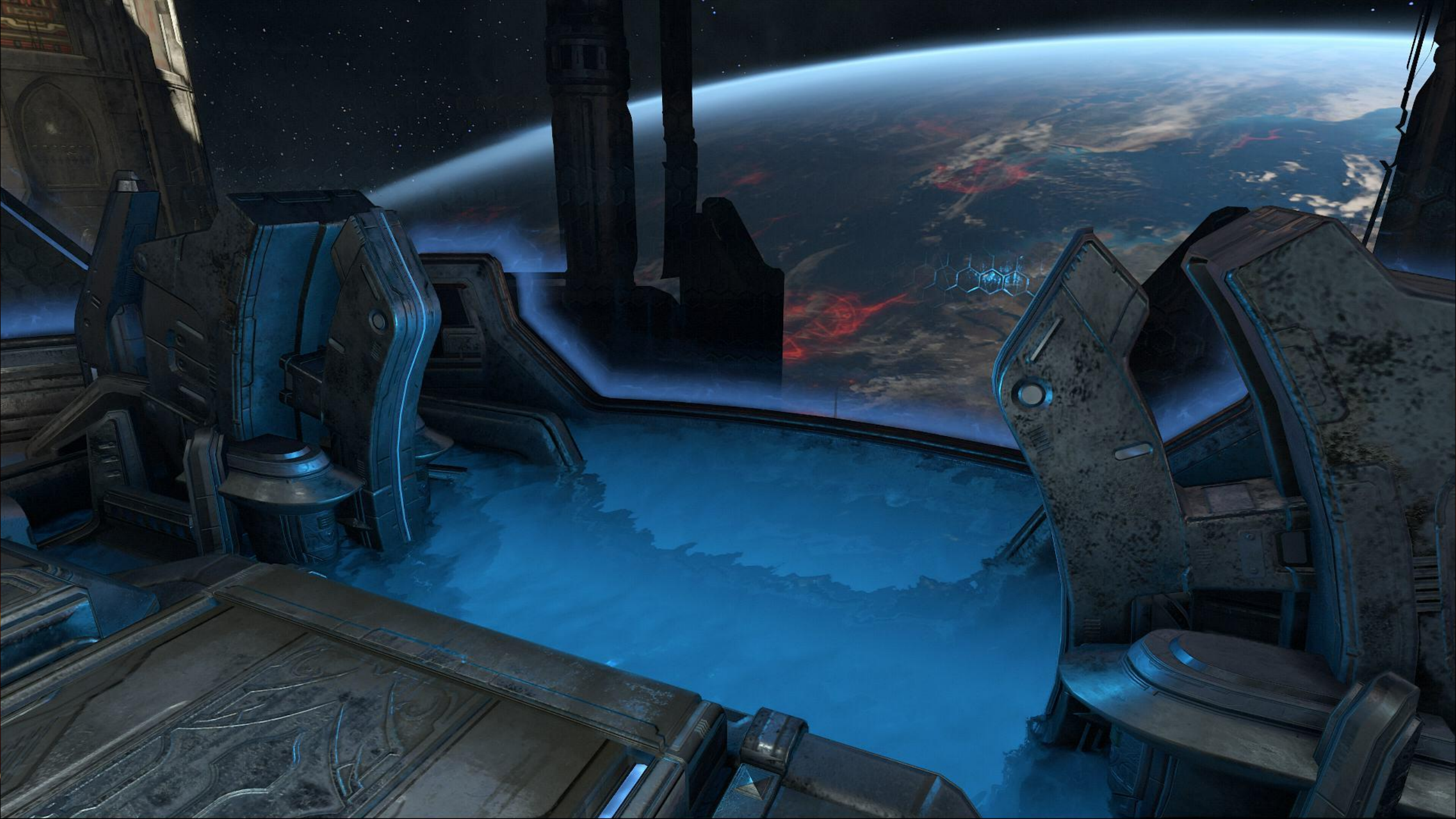
WATER CAUSTICS

- Water caustics image is generated from the water displacement map
- Rasterize grid mesh aligned to the displacement map, and perturb each vertex by simulating refraction to some constant distance
- Resulting caustics image is used next frame: it's projected onto scene geo that's underwater or right above water

```
float3 refractVertex( pos, normal, lightDir ) {
    float3 refractDir = refract( lightDir, normal );
    return pos + REFRACT_DEPTH * refractDir;
}

out float triangleArea;
out float refractedTriangleArea;
vertex_shader {
    // Positions and normals of this grid vertex and its neighbors to the right and above
    // Assume these have already been computed from the water displacement map
    float3 pos, posRight, posUp;
    float3 normal, normalRight, normalUp;

    float3 posRefracted = refractVertex( pos, normal, lightDir );
    float3 posRightRefracted = refractVertex( posRight, normal, lightDir );
    float3 posUpRefracted = refractVertex( posUp, normal, lightDir );

    gl_Position = transformToClipSpace( posRefracted );
    triangleArea = area( pos, posRight, posUp );
    refractedTriangleArea = area( posRefracted, posRightRefracted, posUpRefracted );
}

out float4 fragColor;
fragment_shader {
    // Value describes how focused or diffused the lighting is
    fragColor = triangleArea / refractedTriangleArea;
}
```

ADDING SUPPORT FOR FLOWING WATER

- Rasterize flowing water surfaces directly to water G-buffer
- Flowing water surface normals are perturbed with normal map and flow map authored by artists
- No displacement applied to flowing surfaces (this is room for improvement)



SPECIAL THANKS

id Software	Evgeny Andreeshchev, Derek Best, Bogdan Coroi, Oliver Fallows, Billy Khan, Sascha Herfort, Tiago Sousa
AMD	Timothy Lottes, David Ziman
NVIDIA	Nuno Subtil, Eric Werness

REFERENCES

[Olson2012] “Clustered Deferred and Forward Shading”, Ola Olson et al., HPG 2012

[Sousa2016] “idTech 666 - The Devil is in the Details”, Tiago Sousa et al., Siggraph 2016

[Drobot2017] “Improved Culling for Tiled and Clustered Rendering”, Michal Drobot, Siggraph 2017

[Meyer2010] “On floating-point normal vectors”, Quirin Meyer et al. EGSR’10

[Gneiting2014] “Realtime Geometry Caches”, Axel Gneiting, Siggraph 2014

[Kavan2010] “Fast and Efficient Skinning of Animated Meshes”, Ladislav Kavan et al. CGF 2010

[Euler1770] “Problema algebraicum ob affectiones prorsus singulares memorabile”, Leondar Euler, Opera omnia 1st series 6 (1770): 287-315.

[Wihlidal2016] "Optimizing the Graphics Pipeline with Compute." Graham Wihlidal, GDC 2016

[Tessendorf2001] “Simulating Ocean Water”. Jerry Tessendorf, 2001

[Johanson2004] “Real-Time Water Rendering – Introducing the Projected Grid Concept”. Claes Johanson, Masters Thesis Lund University 2004

[Sousa2008] “Crysis Next Gen Effects”. Tiago Sousa, GDC 2008

[Sousa2011] “CryENGINE 3 Rendering Techniques”. Tiago Sousa, Gamefest 2011

[Müller-Fischer2008] “Fast Water Simulation for Games Using Height Fields”. Matthias Müller-Fischer, GDC 2008