Rendering Technology in 'Agents of Mayhem'

Scott Kircher
Principal Programmer
Core Technology Group
Deep Silver Volition

Rendering Technology in Agents of Mayhem

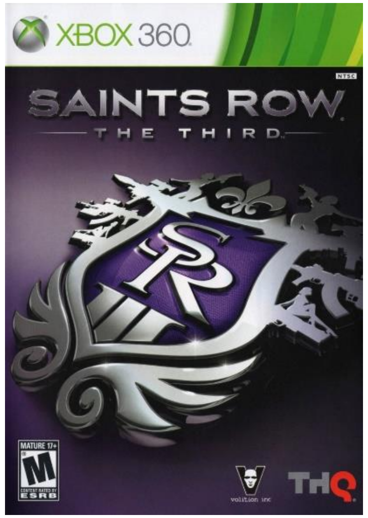
Scott Kircher

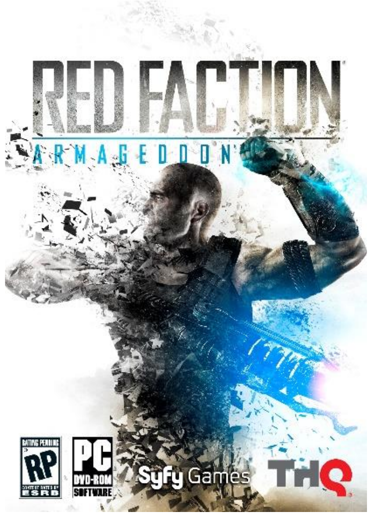
AGENTS OF MAYHEM

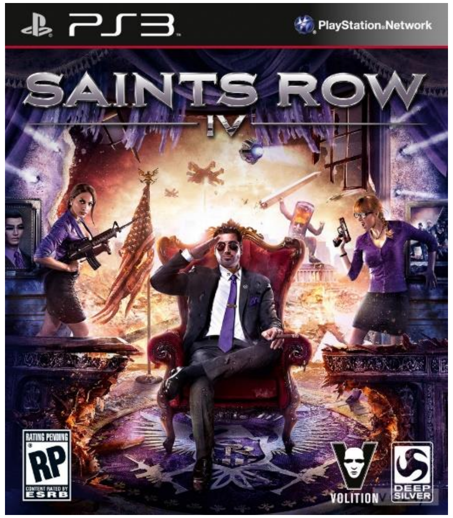
volition

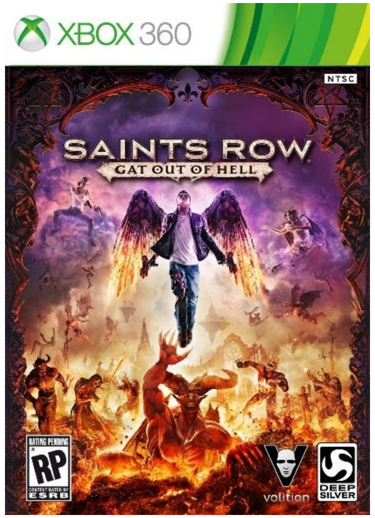
Who am I?

- Principal Programmer
  - Rendering Team in Volition's Core Technology Group
- Ph.D. in Computer Science from UIUC
- Nearly Eleven years of experience at Volition









Agents of Mayhem

- Open World City
- Third-person Action
- Stylized Art with Physically Based Rendering
- Tons of Particles & Alpha Meshes

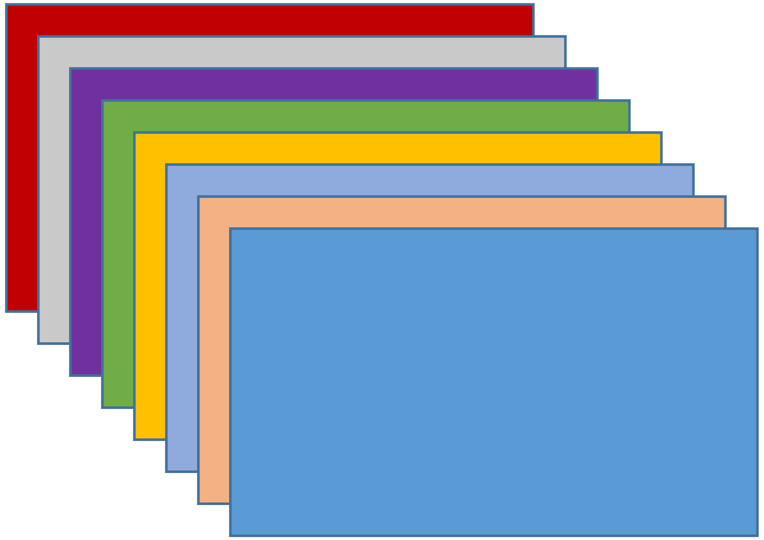
Topics

- Order Independent Transparency
  - Modifications to Weighted Blended OIT [McGuire2013]
- Lighting Compute
  - Features and Optimization
- Global Illumination
  - Better Occlusion for Light Propagation Volumes [Kaplanyan2010]

Order Independent Transparency

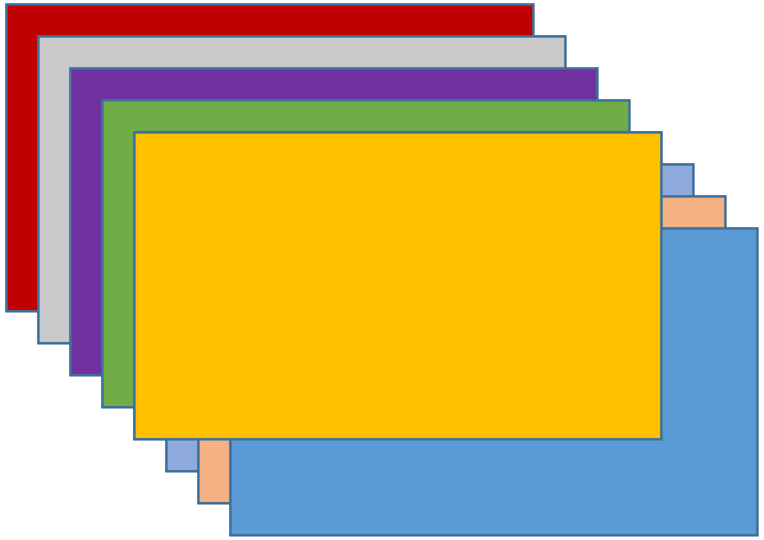
Rationale

- All previous Volition games:
  - Traditional back-to-front CPU sorted alpha



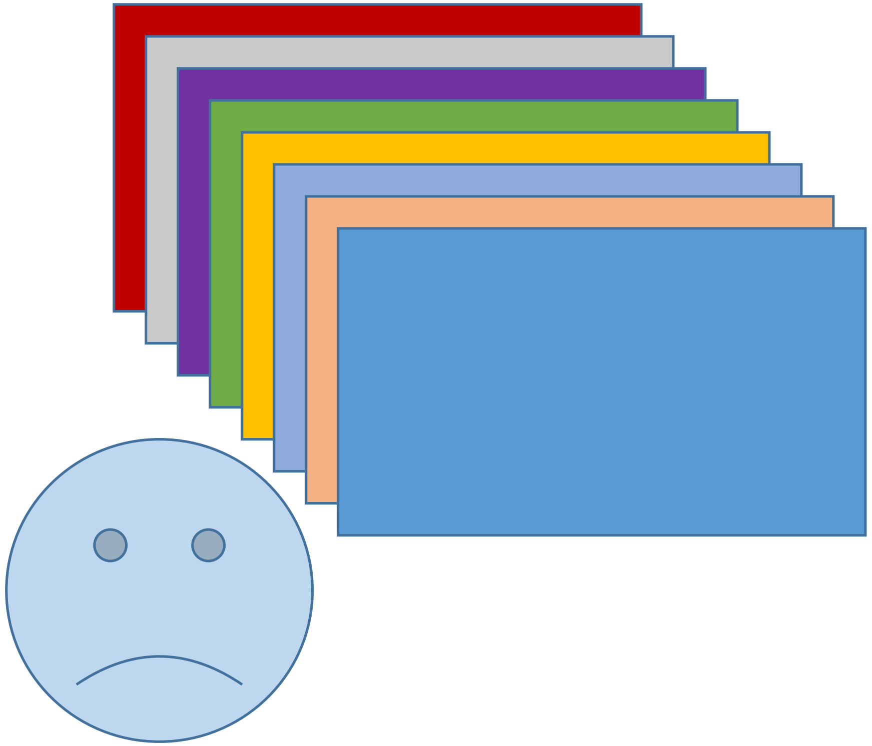
Rationale

- All previous Volition games:
  - Traditional back-to-front CPU sorted alpha
- Lots of sorted alpha means:
  - Inefficient CPU rendering
  - Per “object” sorting, not per-pixel
  - Sort “popping”
  - Low-res alpha doesn’t sort with high-res



Rationale

- All previous Volition games:
  - Traditional back-to-front CPU sorted alpha
- Lots of sorted alpha means:
  - Inefficient CPU rendering
  - Per “object” sorting, not per-pixel
  - Sort “popping”
  - Low-res alpha doesn’t sort with high-res
- Solution: OIT?
  - Many OIT techniques inefficient on GPU



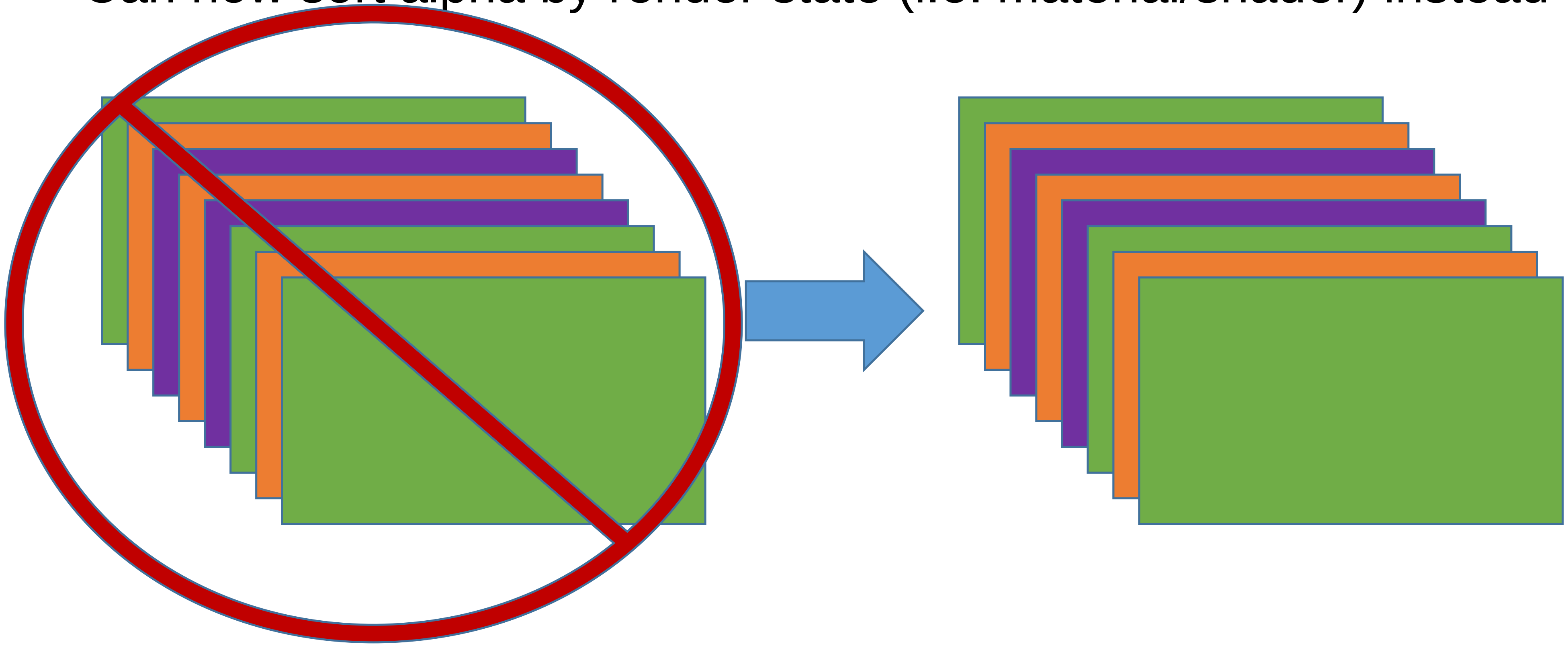
Weighted-Blended OIT

- Enter McGuire & Bavoil [McGuire2013, McGuire2015]

Image from [McGuire2013]

Weighted-Blended OIT Pros

- “Negative” CPU cost
  - Can now sort alpha by render state (i.e. material/shader) instead of depth



Weighted-Blended OIT Pros

- "Negative" CPU cost
  - Can now sort alpha by state instead of depth
- Efficient on GPU
  - Some math added to alpha shaders
  - Simple full-screen composite step
- Low-res and high-res alpha "sort" seamlessly
- No popping, ever.
  - "Sort" issues transition smoothly
- Simple?
  - No. But close enough.

Weighted-Blended OIT Cons

- MAGIC NUMBERS EVERYWHERE
- Very opaque alpha behaves badly
- Always “wrong”
  - (But not wrong enough!)

How WBOIT Works (McGuire)

- Replace ordered blending with weighted average

$$C = \frac{\sum S_n \alpha_n w_n}{\sum w_n}$$

$$R = \prod (1 - \alpha_n)$$

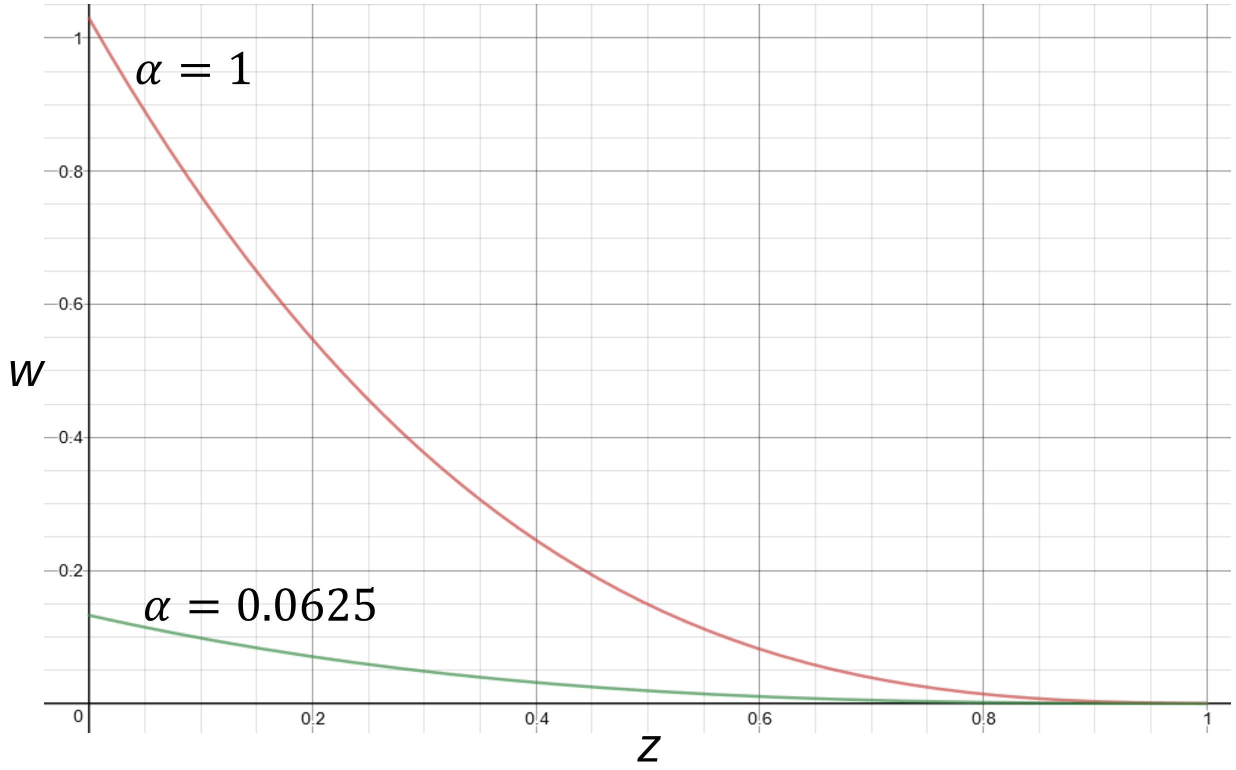
$$final\ color = C + D(1 - R)$$

$S_n = Output\ color\ of\ fragment$
$\alpha_n = Output\ opacity\ of\ fragment$
$w_n = WBOIT\ weight\ of\ fragment$
$D = Existing\ color\ in\ destination\ buffer$

Weighting Function (McGuire)

- Weights are the "magic"
- Weight high-coverage things more
- Weight near things more

$$a = \min(8\alpha, 1) + 0.01$$
$$b = 1 - 0.95z$$



$$w = f(a^3, b^3)$$ [McGuire2015]

Where *f* rescales/clamps *w* for precision

Emissive Alpha – Major Problem

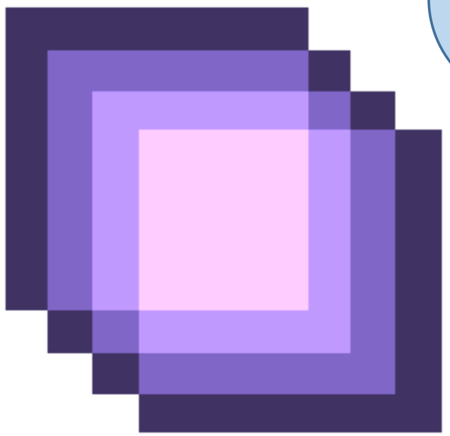
Intuition

- Consider n layers of the same emissive alpha value E

$$C = \frac{\sum_{i=1}^{n} E w_i}{\sum_{i=1}^{n} w_i} = E$$

$$C' = \sum_{i=1}^{n} E = nE$$

$$\frac{C'}{C} = n$$

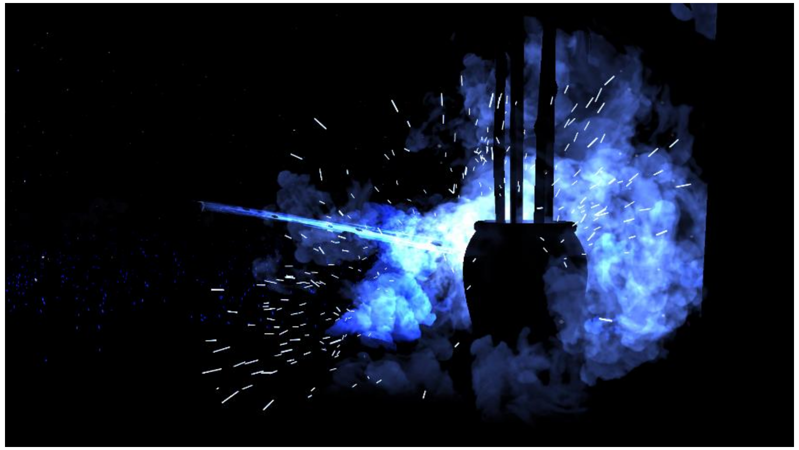


Main Idea

- Accumulate additional information
  - “Additiveness” ≈ Number of additive layers
- Amplify weighted average by additiveness

Visual Summary of New WBOIT

Accumulated weighted colors



÷

Accumulated weights

×

Additiveness

Blended via

Revealage

Visual Summary of New WBOIT

WBOIT Formulas + Additiveness

- Revealage remains the same

$$R = \prod (1 - \alpha_n)$$

- Color is the same
  - But with emissive explicitly identified

$$C = \frac{\sum (S_n \alpha_n + E_n) w_n}{\sum w_n}$$

- Additiveness is new

$$A = \sum \min(10 \cdot \mathrm{lum}(E_n), 1)$$

Arbitrary sensitivity constant

$S_n = Output\ non-emissive\ color\ of\ fragment$
$\alpha_n = Output\ opacity\ of\ fragment$
$E_n = Output\ emissive\ color\ of\ fragment$
$w_n = WBOIT\ weight\ of\ fragment$

New WBOIT Composite

- Additiveness amplifies weighted average color
  - But needs to be mitigated for mixed emissive/non-emissive

Reduces additiveness in areas of high opacity (low revealage)

Prevents darkening in absence of emissive

$$A' = \left(\frac{A}{4}(1 - R) + A \cdot R\right) + \min(2(1 - R), 1)$$

$$final\ color = (A')C + D(1 - R)$$

$A = Accumulated\ additiveness$
$R = Accumulated\ revealage$
$C = Weighted\ average\ of\ colors$
$D = Color\ in\ destination\ buffer$

Weighting Function and Emissive

- Purely emissive alpha has zero opacity
- Must include emissive in computation of weight
- Must allow weight to go to zero

$$a = \min(8\alpha, 1) + 0.01$$

$$a = \min(3\alpha + k \cdot \mathrm{lum}(E), 1)$$

$k = 2$	Particles
$k = ?$	
$k = 20$	Alpha Meshes

$$w = f(a^3, b^3)$$

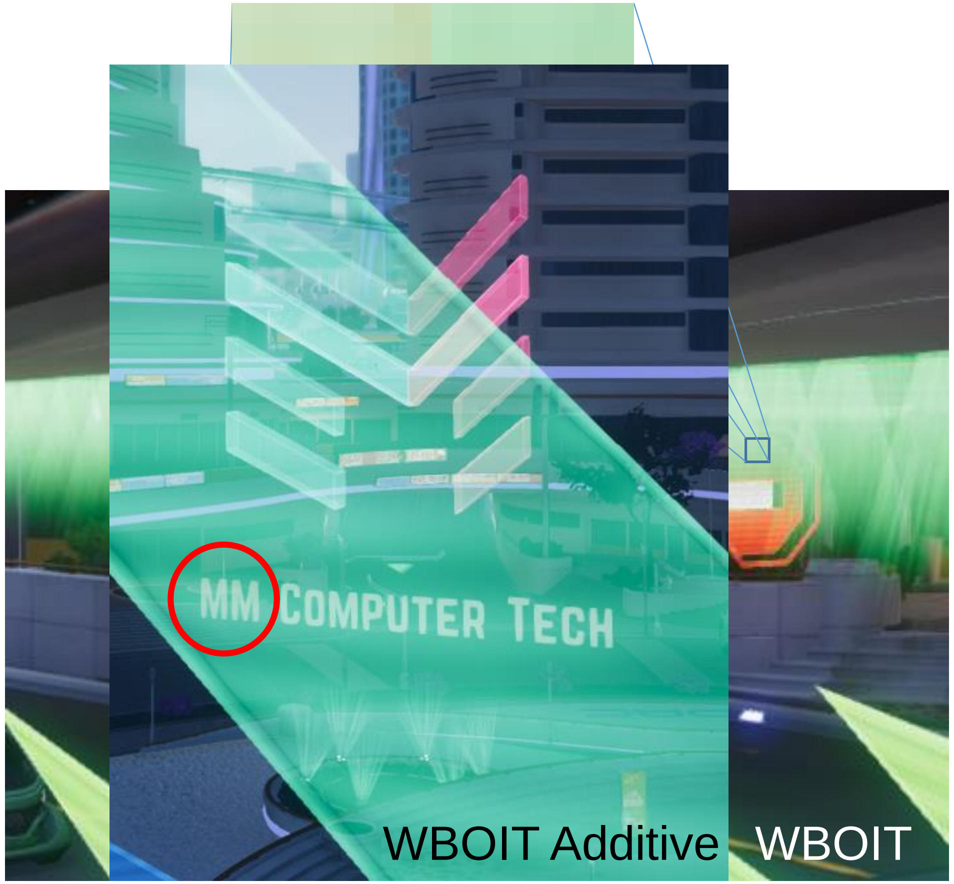
$\alpha = Opacity\ of\ fragment$
$E = Emissive\ color\ of\ fragment$
$k = New\ emissiveness\ weighting\ sensitivity$

Other WBOIT Issues

And how we dealt with them

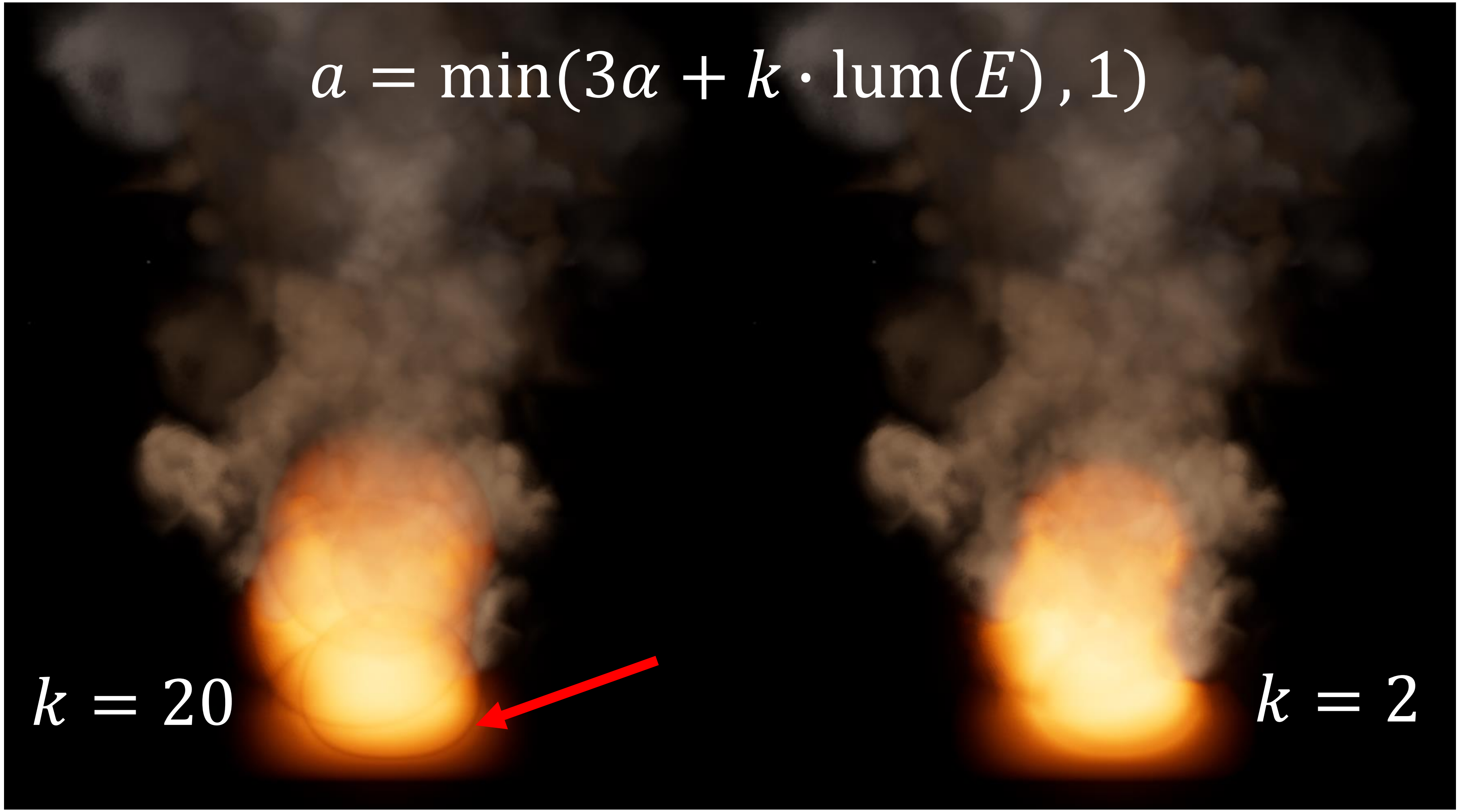
Color Dominance

- Side-effect of WBOIT with Additiveness
  - Luminance is adjusted, but hue dominated by foreground layers
  - Our artists actually *liked* this



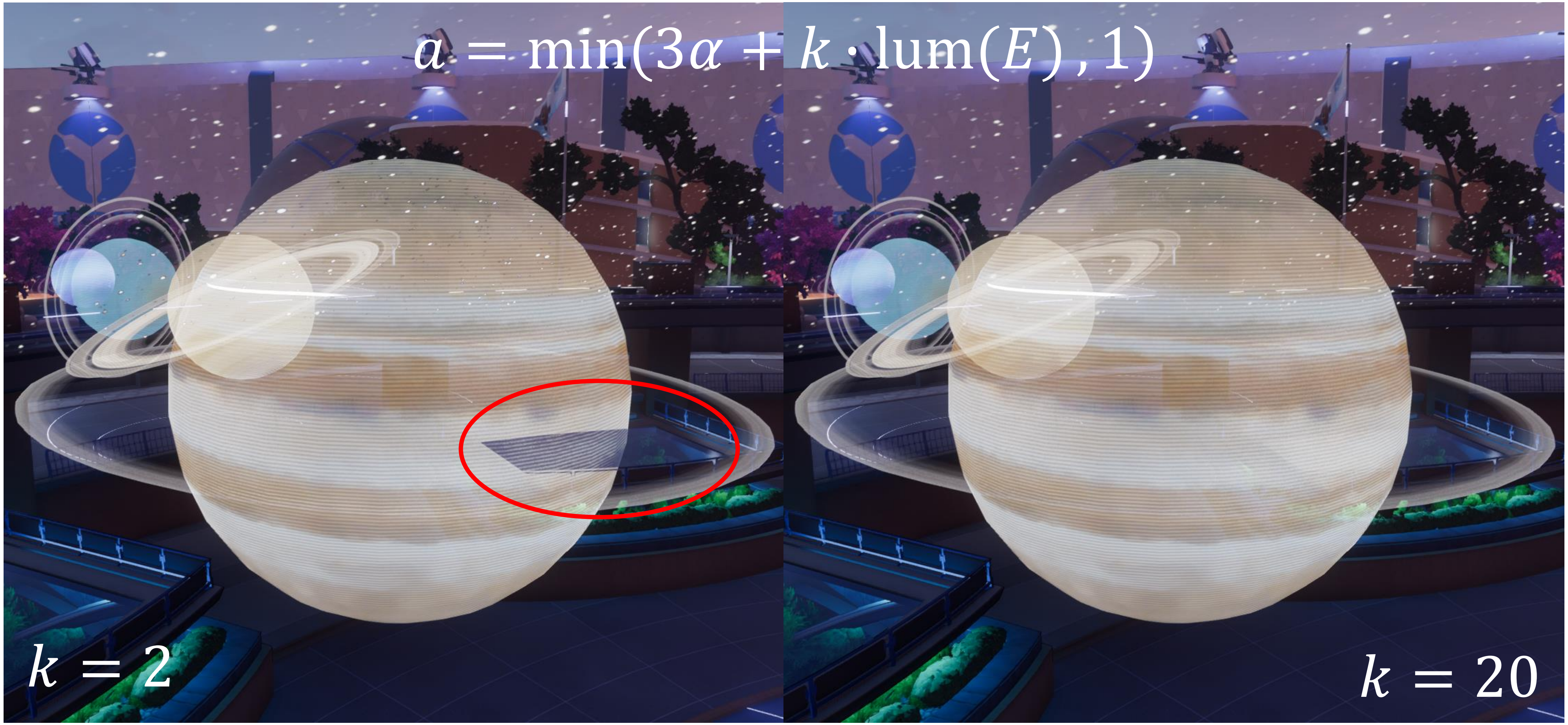
Dark Halos

- High “sensitivity” to opacity or emissivity produces these



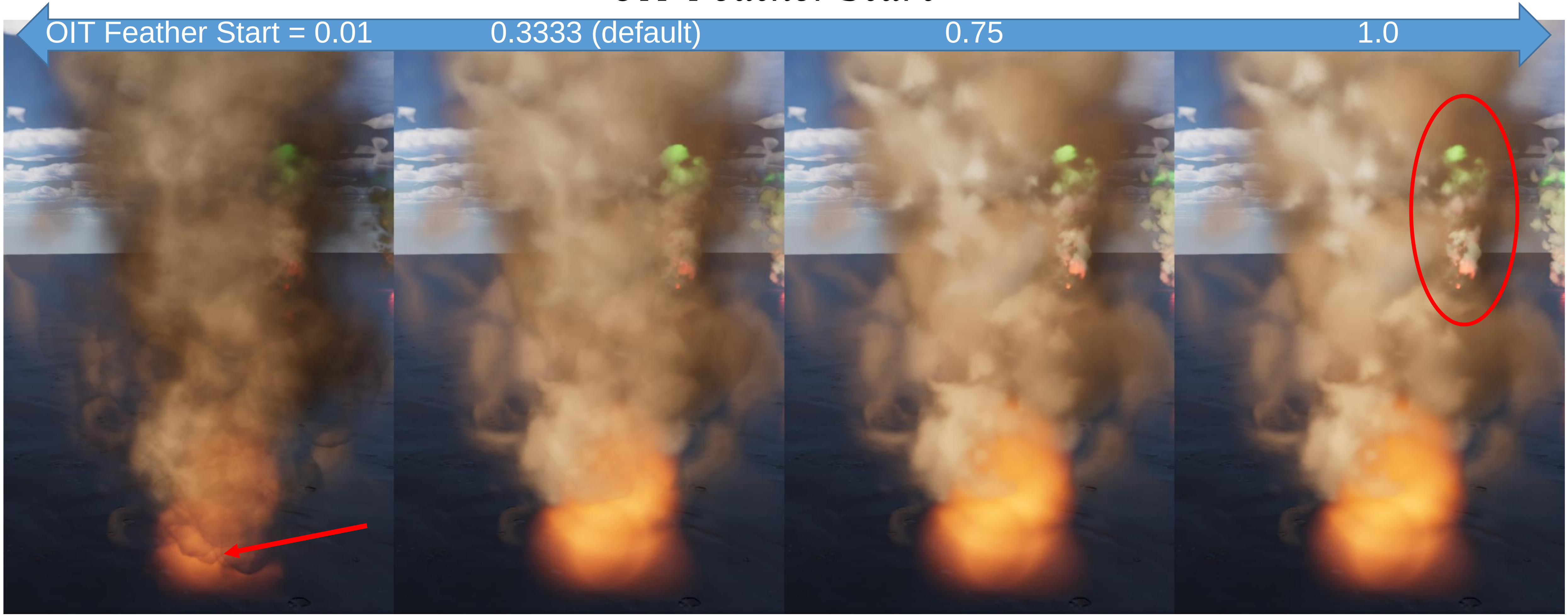
Punch Through

- Low “sensitivity” with dim emissive can produce punch-through



Halo vs. Punch Through Control

$$k = \frac{Camera\ Exposure}{OIT\ Feather\ Start}$$



Depth Range

- To avoid retuning, convert depth to a canonical range
- We chose near = 0.5m, far = 300m
- Also, we allow *b* to go to zero
  - We have an alternate method of dealing with very low weights

$$z' = \text{saturate}\left(\frac{F}{F-N} - \frac{F \cdot N}{d(F-N)}\right)$$

$$b = 1 - z'$$

$$w = f(a^3, b^3)$$

$N = 0.5$

$F = 300$

$d =$ linear view depth

Weight Biasing/Clamping

- FP16 Precision is an issue. Solved already [McGuire2013,2015]
- Large variance in weights between near and far alpha is bad.

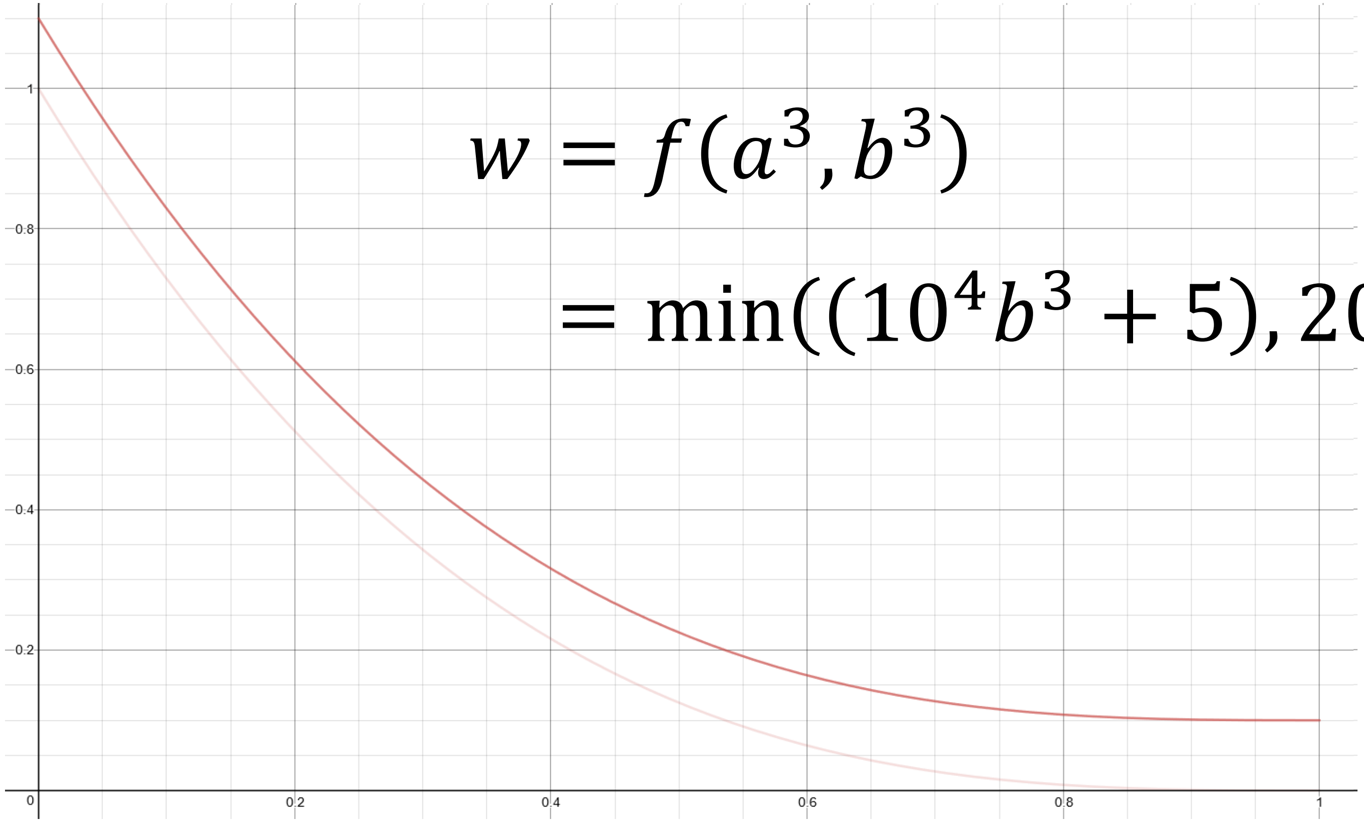
$$w = f(a^3, b^3)$$

$$= \min(10^4 a^3 b^3, 300)$$

$a = Opacity\ weighting\ factor$
$b = Depth\ weighting\ factor$

Better Weight Biasing/Clamping

- Can’t just introduce big clamp at low end
  - Lose depth sorting when weights are clamped
  - Instead, shift weights up (only depth-related portion)



$$w = f(a^3, b^3)$$

$$= \min((10^4 b^3 + 5), 20) a^3$$

Opacity weight multiplied in *after* biasing and clamping!

$a = Opacity\ weighting$
$b = Depth\ weighting$

Implementation

- Simple 2-target MRT setup.
  - Second MRT stores Revealage in Red and Additiveness in Alpha
  - Use separate blending control for alpha channel
- See Appendix for more details
  - Shader Source Code
  - CMASK Optimization

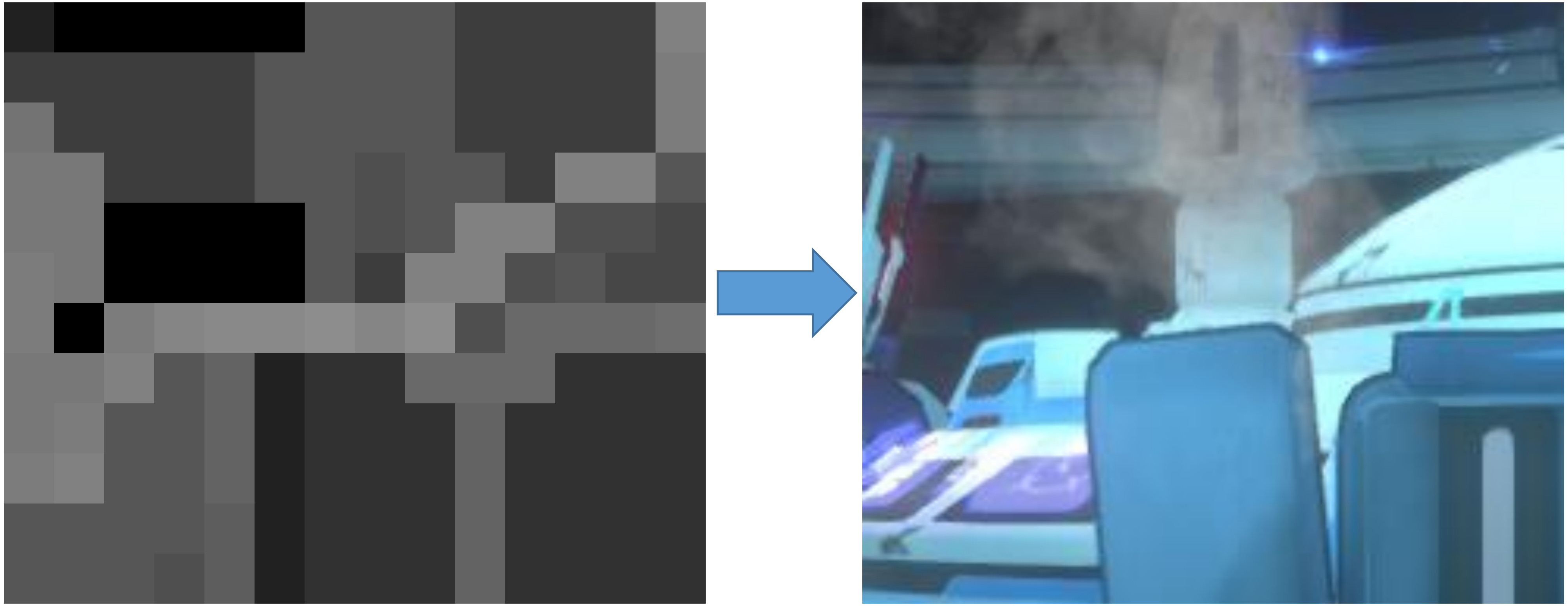
Lighting Compute

Tile-Based Lighting Compute

Tile-Based Lighting Compute

Tile-Based Shading Review

- Compute shader culls lights to tiles (groupshared list per tile)
- Then shades pixels in tile per those light lists

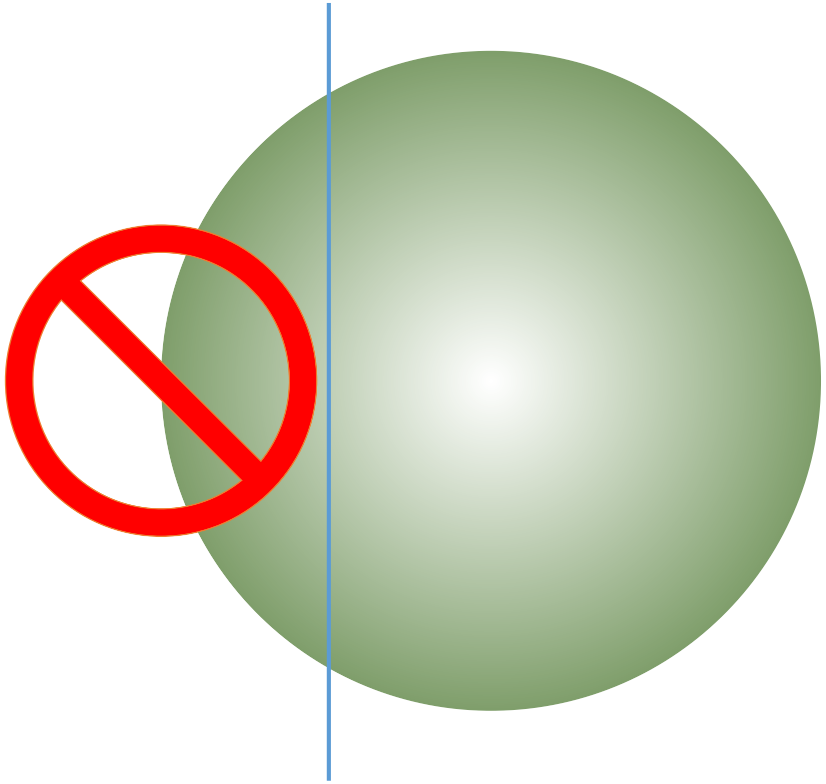


Features

- Lots of (expensive) lighting features implemented
  - Multiple lighting models (all PBR)
  - PCF shadows
  - Variable penumbra shadows (PCSS)
  - Projected textures
  - Textured-emitter area lights
  - Omni lights
  - “Realistic” tube lights
  - Square or round spot lights
  - Darks (negative lights)
  - Light clip planes
  - Light blockers & portals

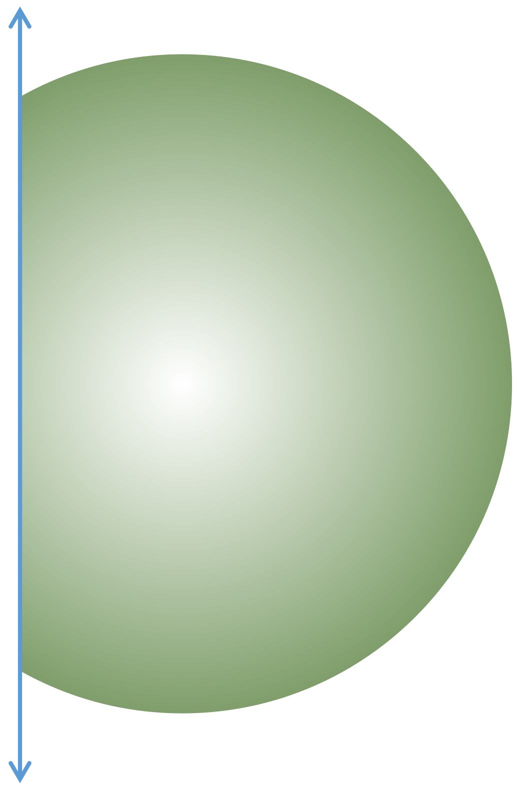
Light Leaking

- Familiar problem

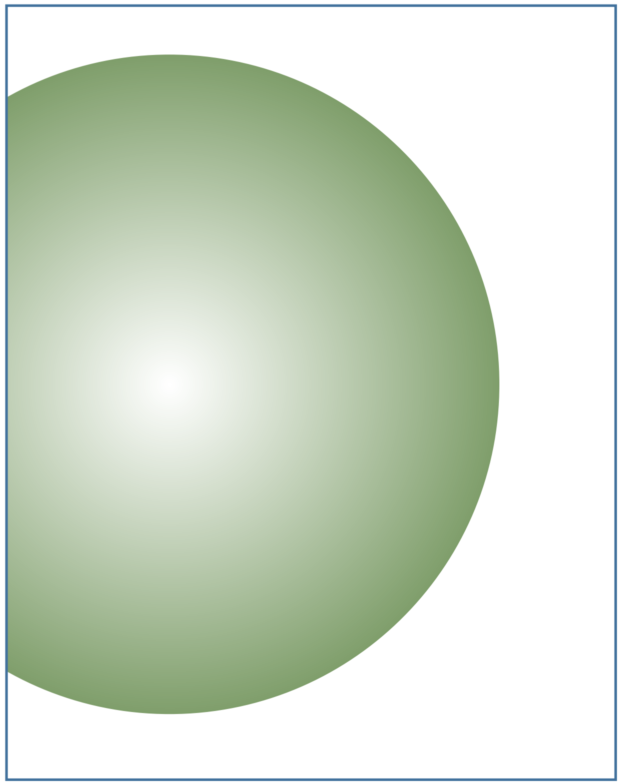


Light Leaking

- Familiar problem, standard solutions



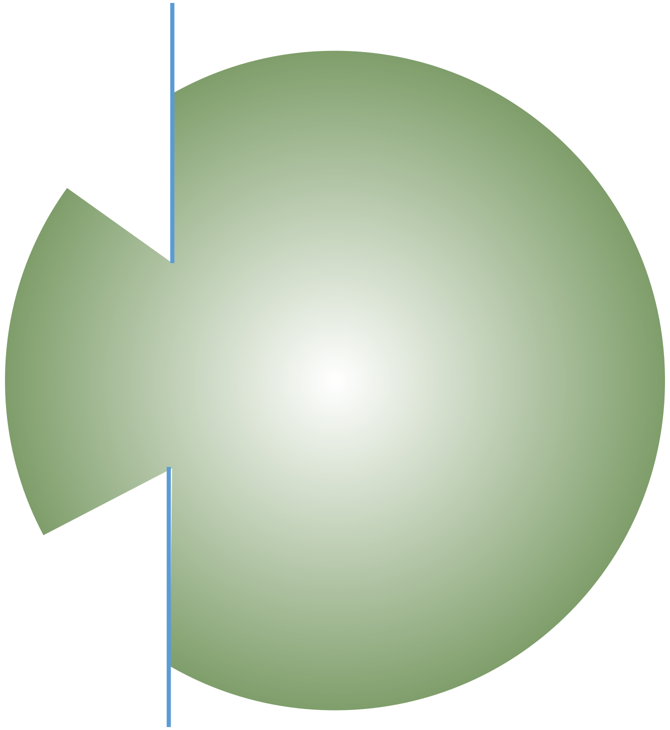
Infinite clip planes



Stencil clip meshes

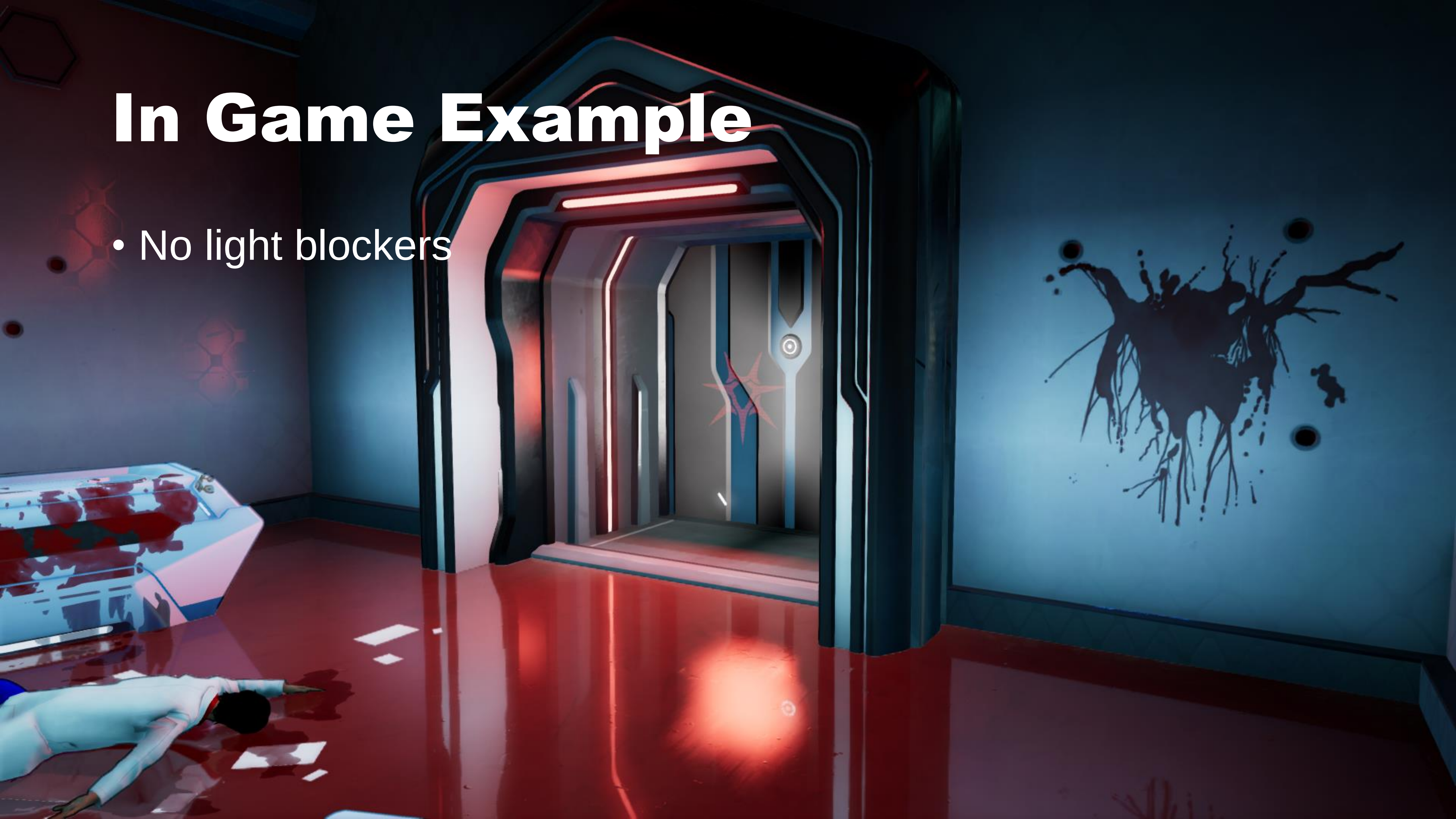
Light Blockers

- *Finite* light clip planes



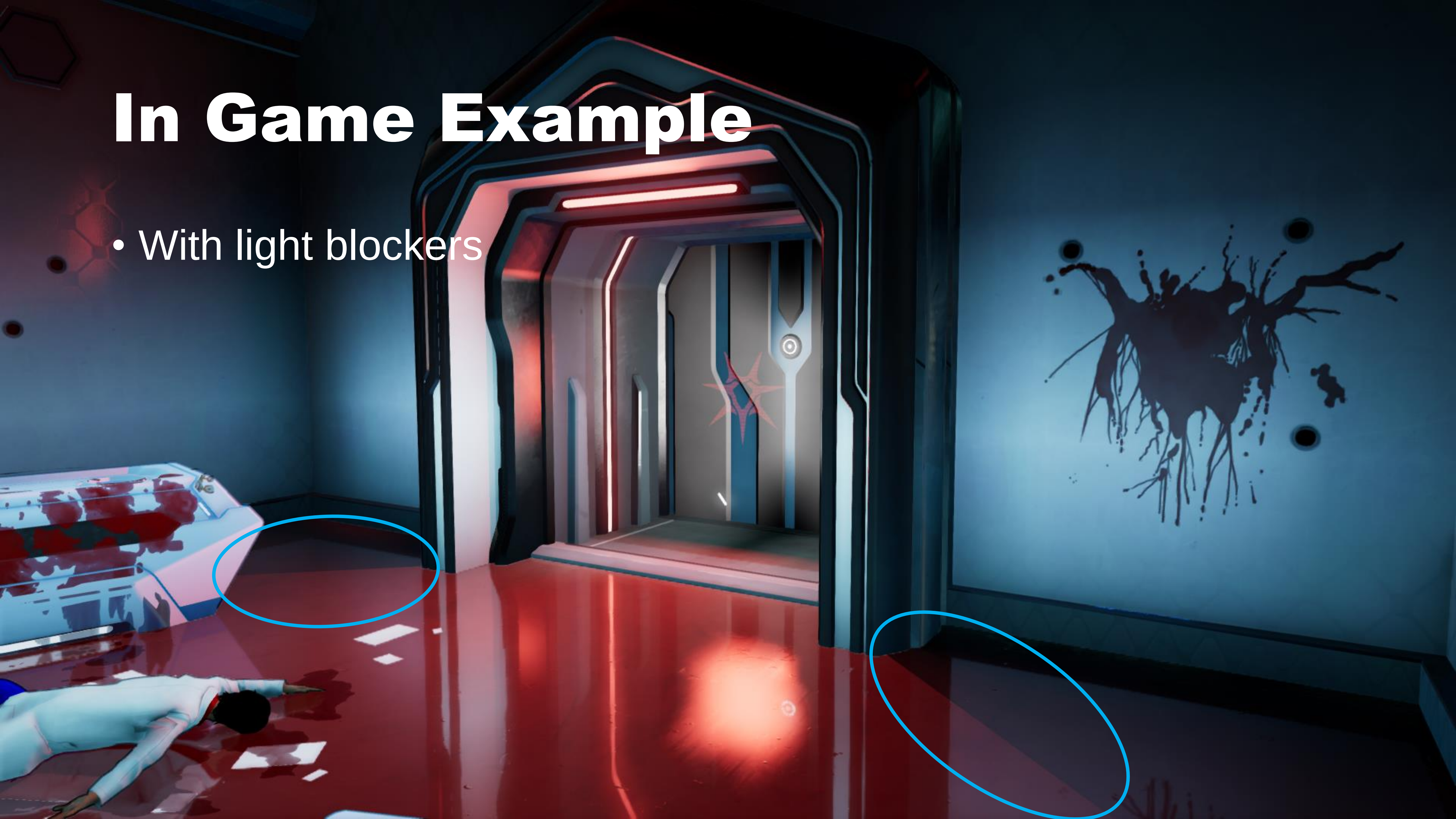
In Game Example

- No light blockers



In Game Example

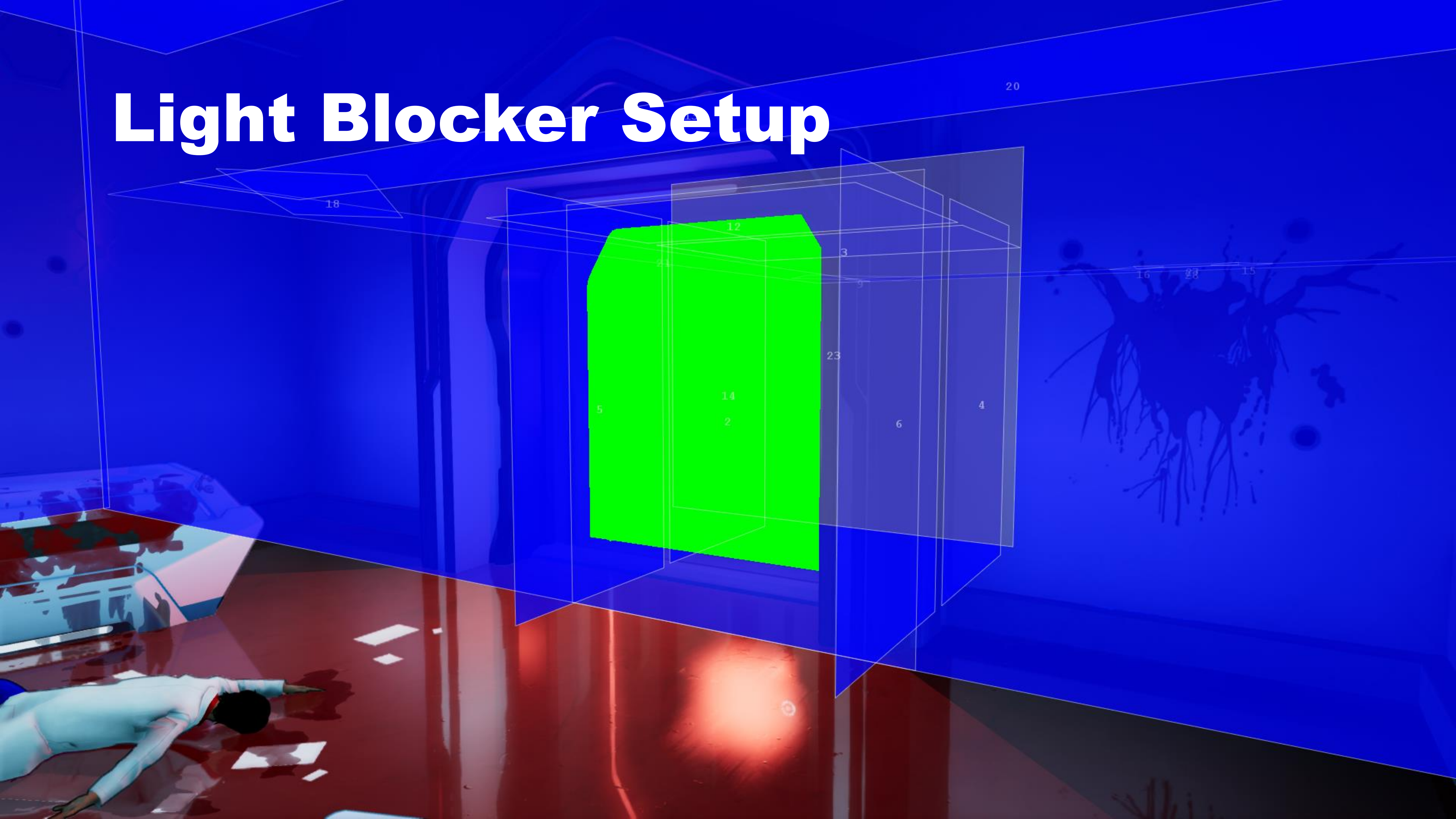
- With light blockers



Why Not Shadow Casters?

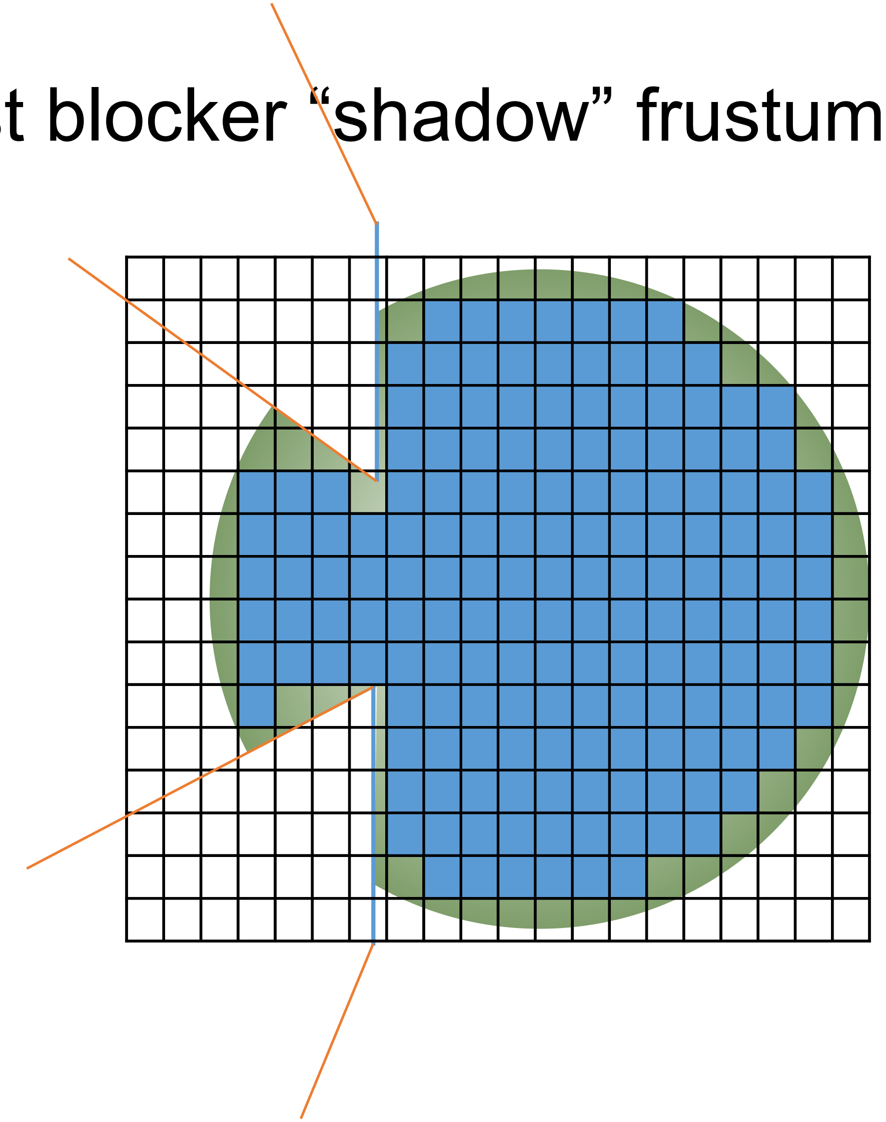
- Too many lights, some don't even support shadows

Light Blocker Setup



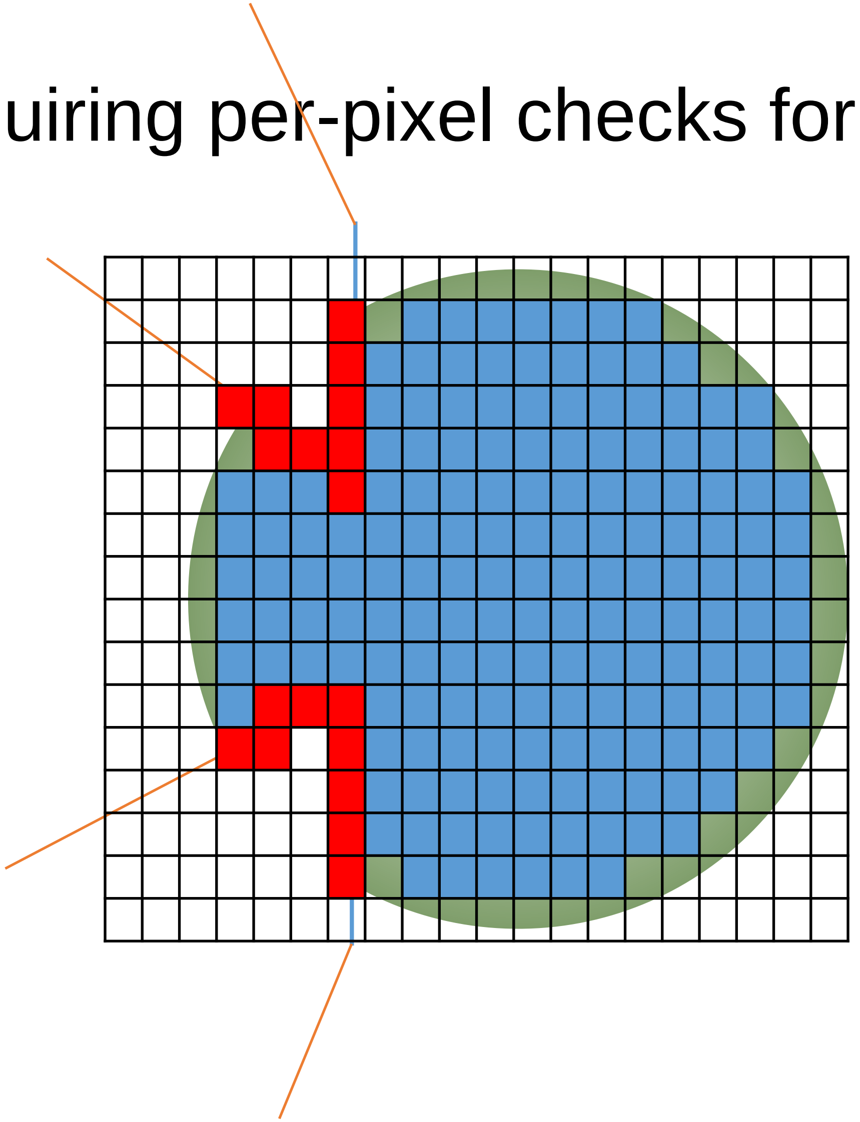
How It Works

- Cull tiles against blocker "shadow" frustums



How It Works

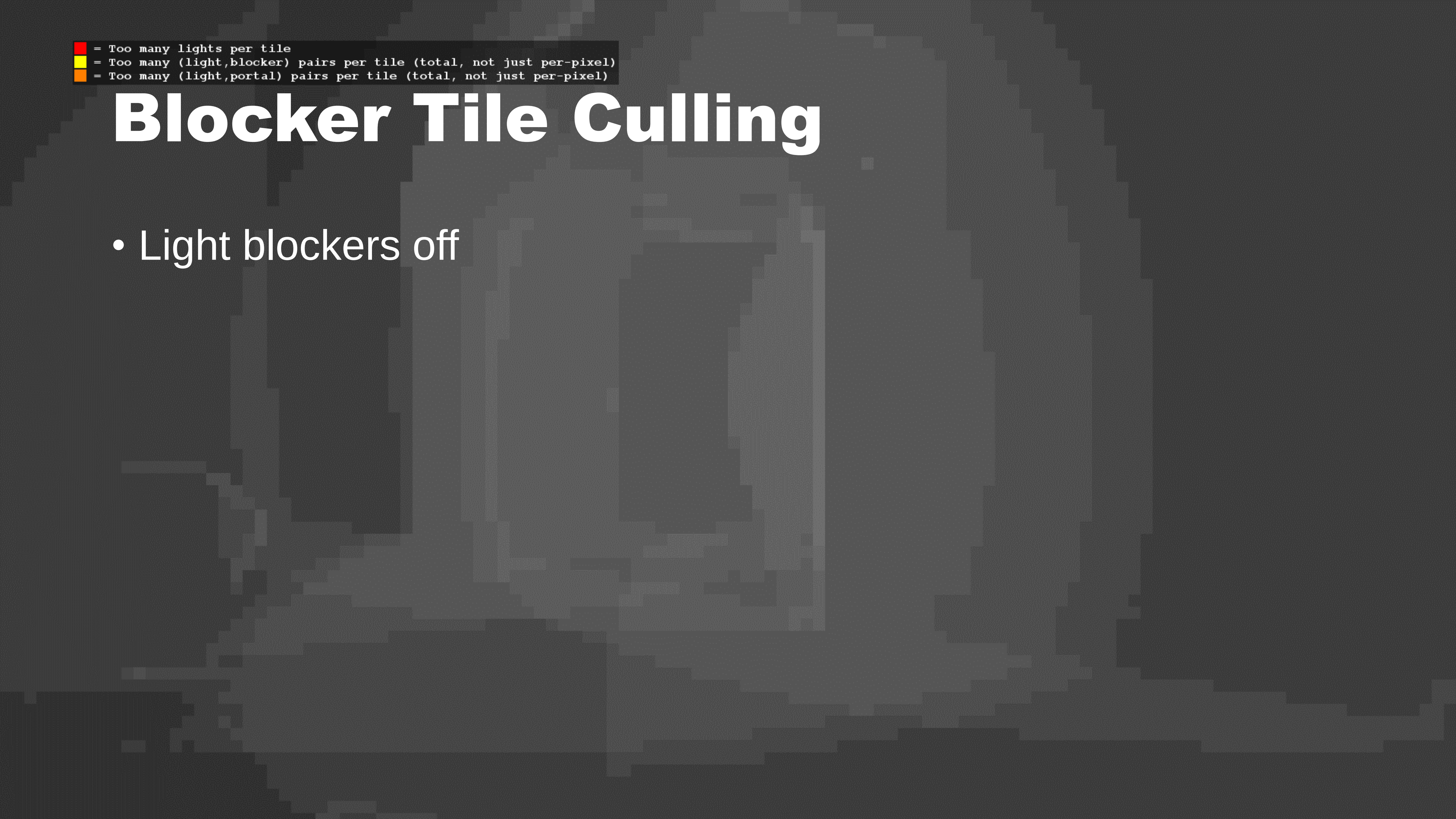
- List blockers requiring per-pixel checks for each light



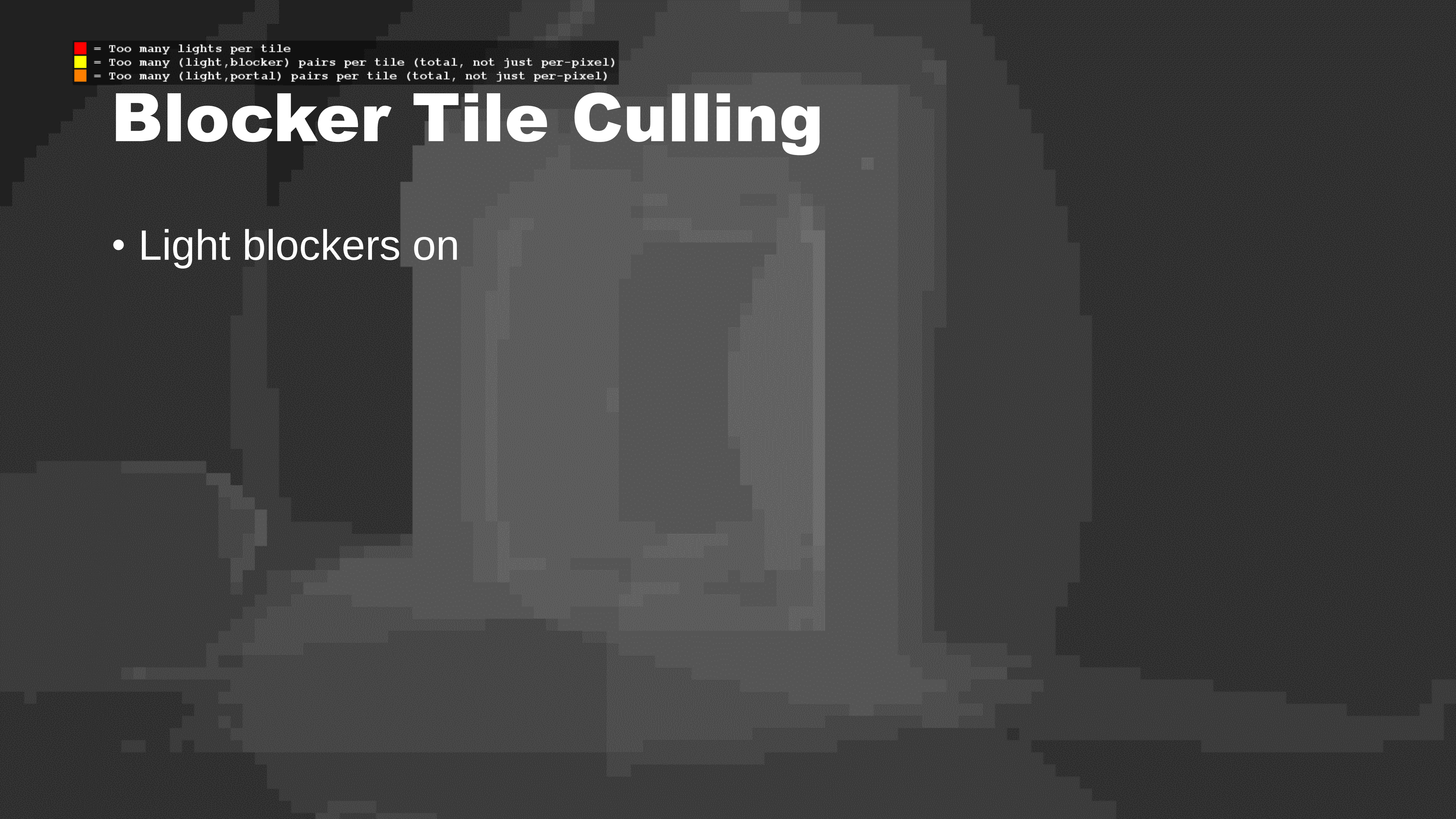
Returning To This Example

- For a moment

Blocker Tile Culling



- Light blockers off



Blocker Tile Culling

- Light blockers on

Blocker Tile Culling

- Tiles requiring per pixel checks

GAME DEVELOPERS

One threadgroup (256 threads) = one 16x16 pixel tile

Implementation

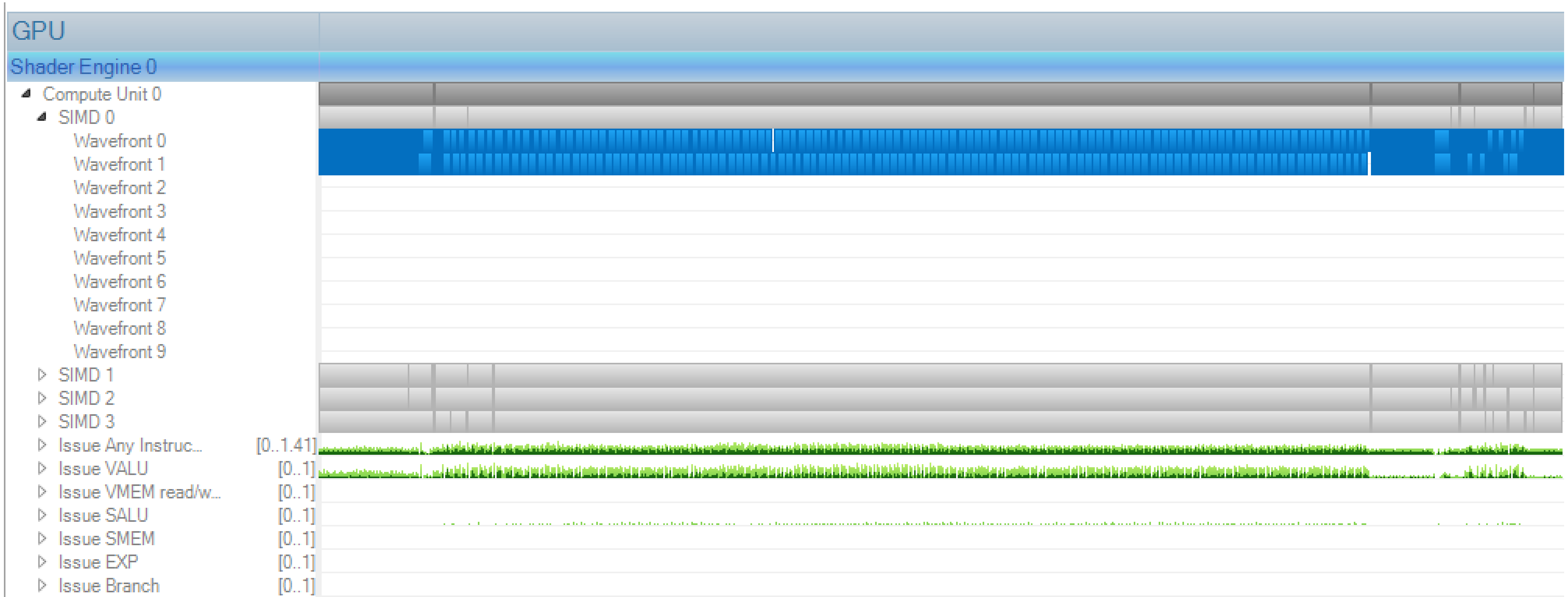
Process (groupsync between each phase)	Thread allocation	Groupshared memory (LDS)
Cull lights vs. tile	One per light	Light list
Build list of (light, portal) pairs	One per light	Light list (Light, portal) list
Classify (light, portal) pairs vs. tile	One per pair	Light list (Light, portal) list (Light, portal) bitarrays (per pixel / enclosed)
Build list of (light, blocker) pairs	One per light	Light list (Light, blocker) list (Light, portal) bitarrays (per pixel / enclosed)
Classify (light, blocker) pairs vs. tile	One per pair	Trimmed light list (Light, blocker) list (Light, portal) bitarray (per pixel test needed) (Light, blocker) bitarray (per pixel test needed)
Compact & sort surviving lights	One per light	

Feature Spectra

Lighting Compute Optimization

Remember That Feature List?

- Lots of features means lots of register usage
  - More registers per thread = less threads per shader unit
- Naïve implementation = BAD occupancy



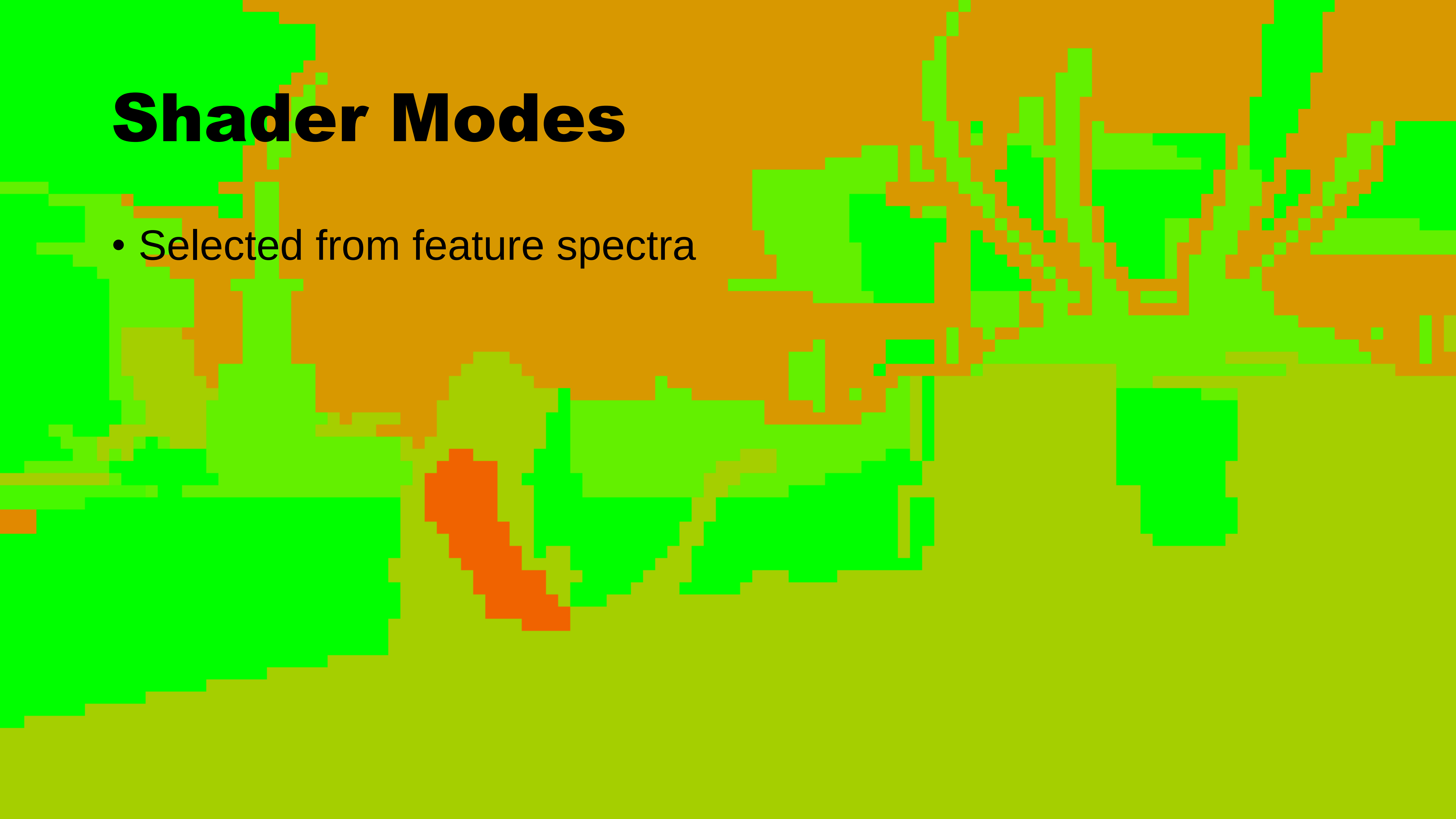
Main Idea

- Break shader into culling phase + different combinations of features
- Select feature set (or *spectrum*) based on needs, *per tile*
- Culling phase determines what shader to use for each tile

Feature Spectra

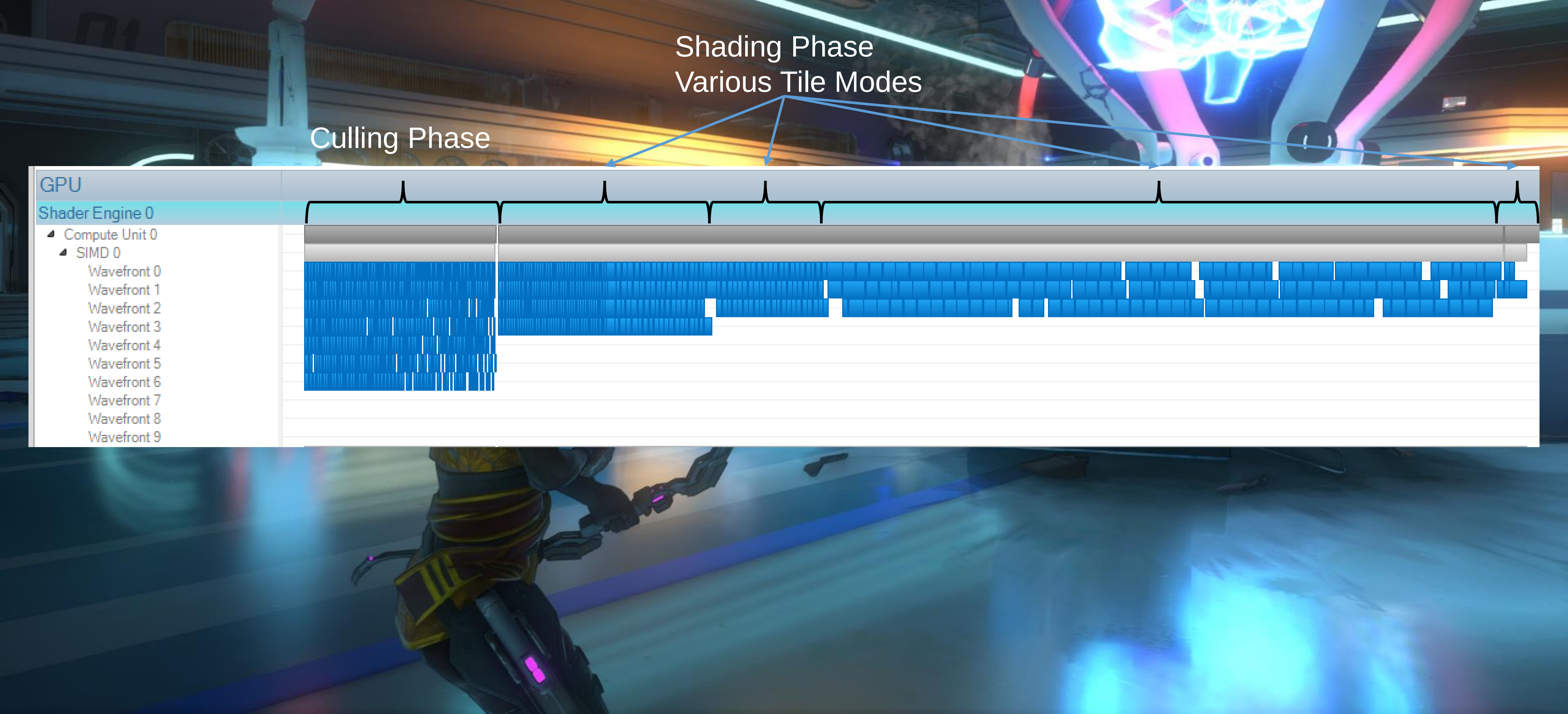
Shader Modes

- Selected from feature spectra



Investigating Feature Spectra

Result



Shading Phase
Various Tile Modes
Culling Phase
GPU
Shader Engine 0
Compute Unit 0
SIMD 0
Wavefront 0
Wavefront 1
Wavefront 2
Wavefront 3
Wavefront 4
Wavefront 5
Wavefront 6
Wavefront 7
Wavefront 8
Wavefront 9

Global Illumination

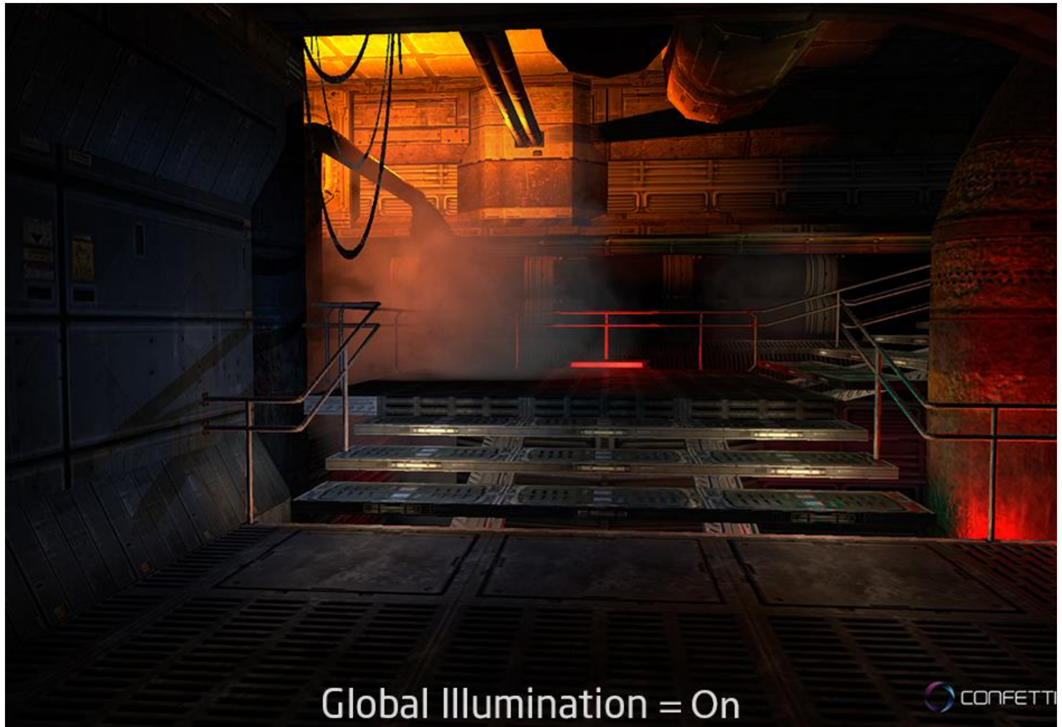
Light Propagation Volumes

- One of first real-time GI techniques
- Crytek

[Kaplanyan2010]

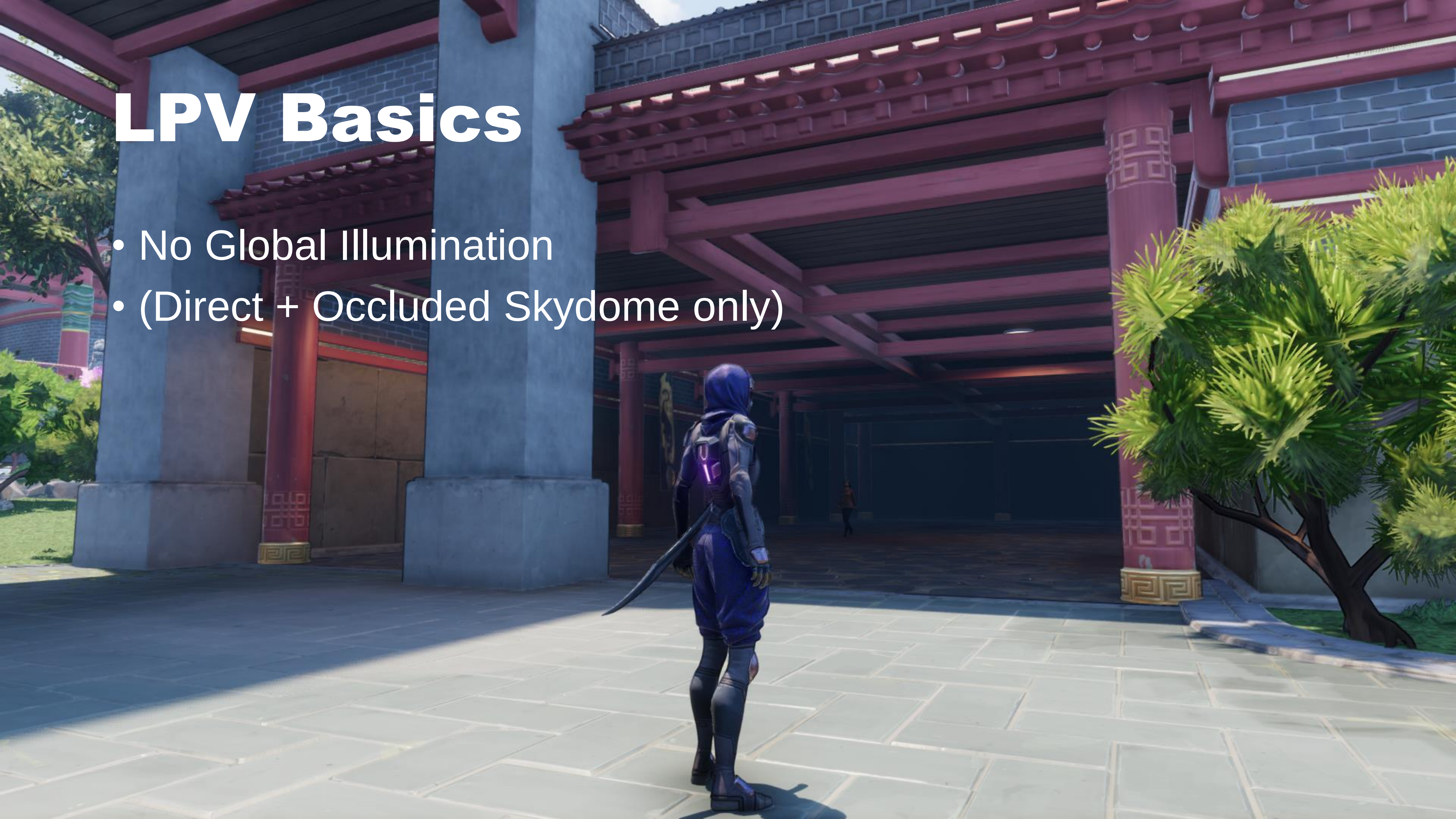
LPV Middleware

- Our starting point: Aura Library from CONFETTI
- Heavily modified (by Volition's own Mike Flavin)
- Modifications applicable to any LPV implementation



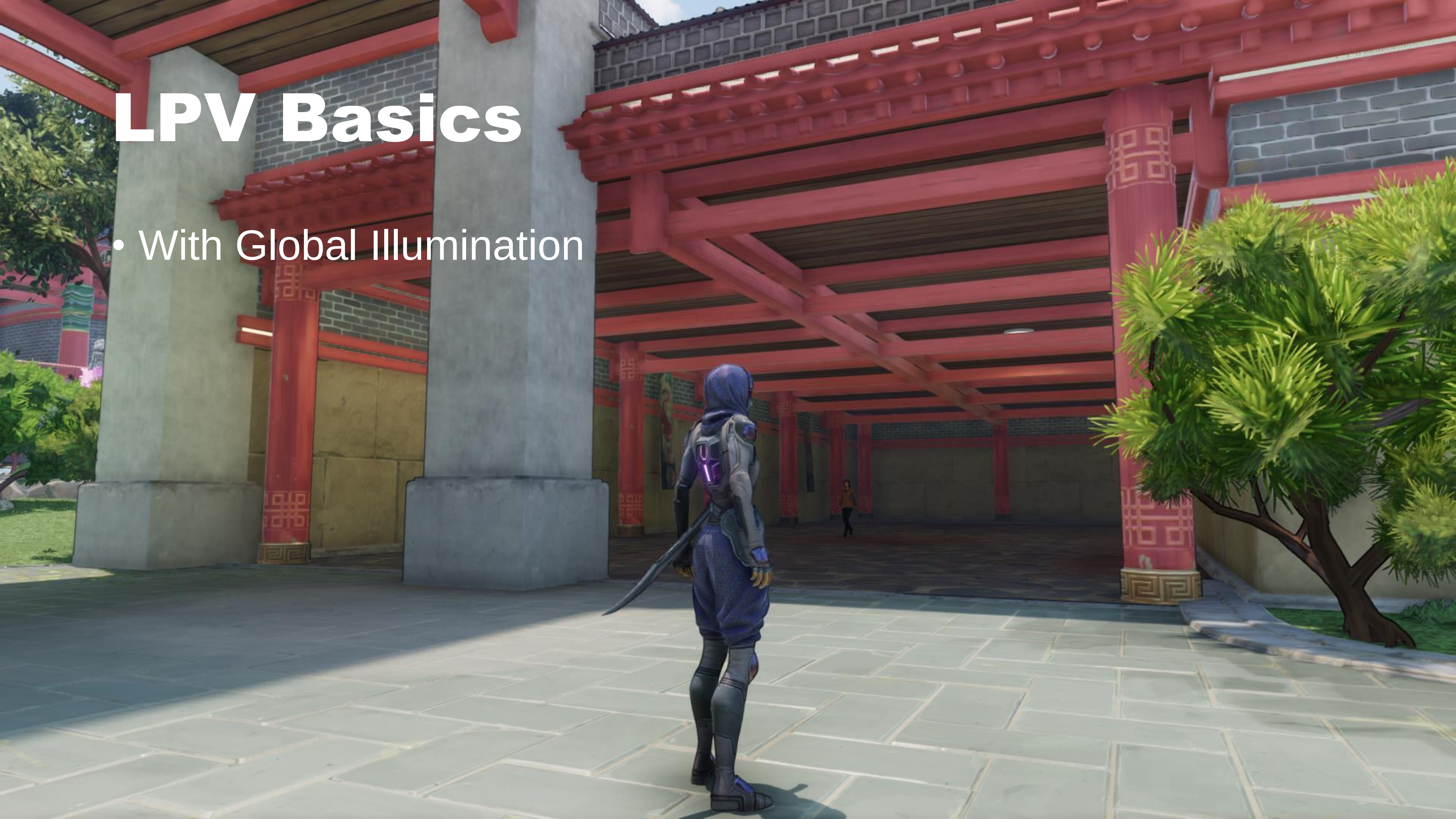
LPV Basics

- No Global Illumination
- (Direct + Occluded Skydome only)



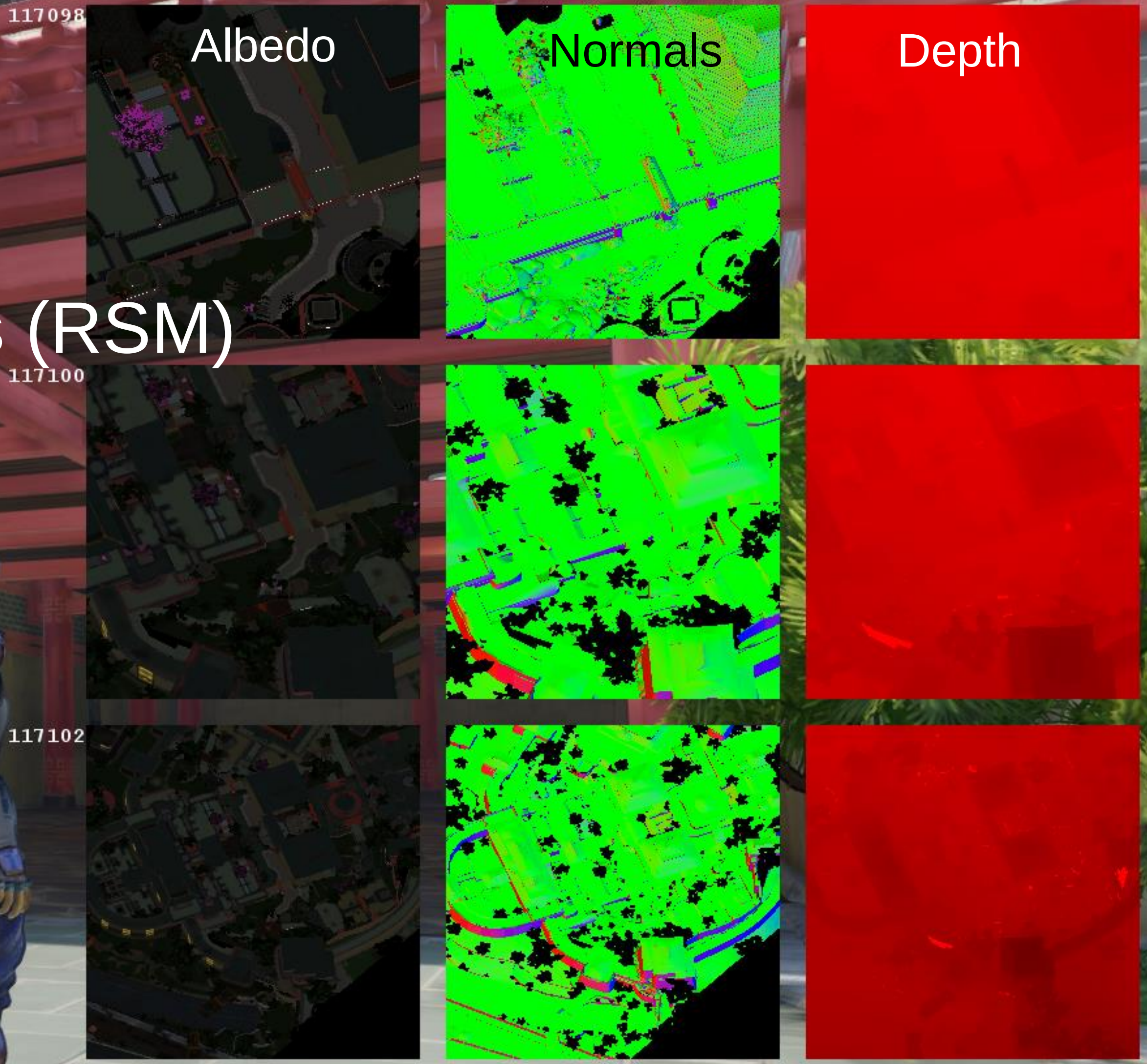
LPV Basics

- With Global Illumination



LPV Basics

- Render Reflective Shadow Maps (RSM)
- Inject into LPV volumes
- Propagate light through volume
- Apply to scene



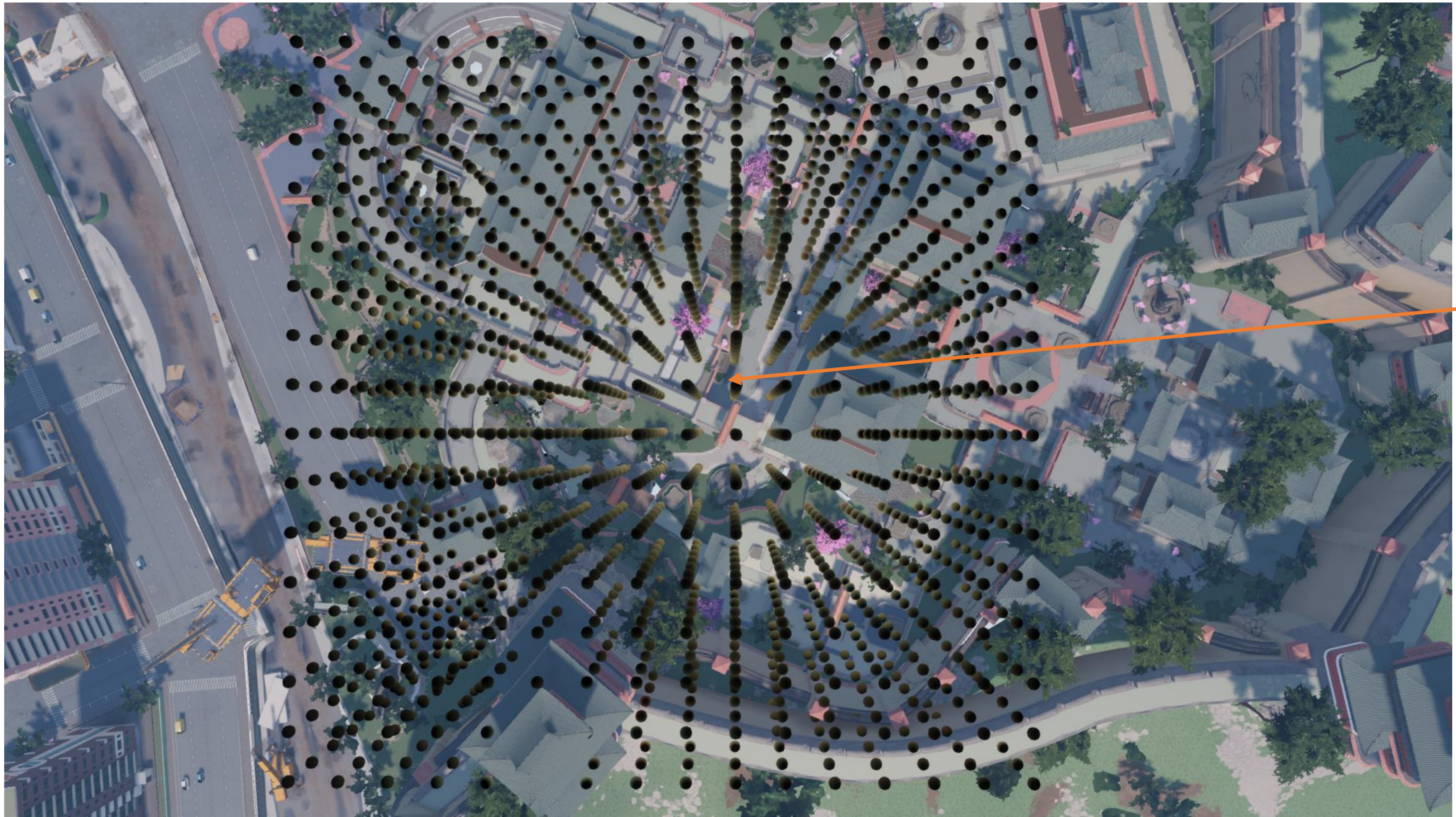
LPV Basics

- 3D LPV volumes store SH of radiant intensity function

Global vs. Local Volumes

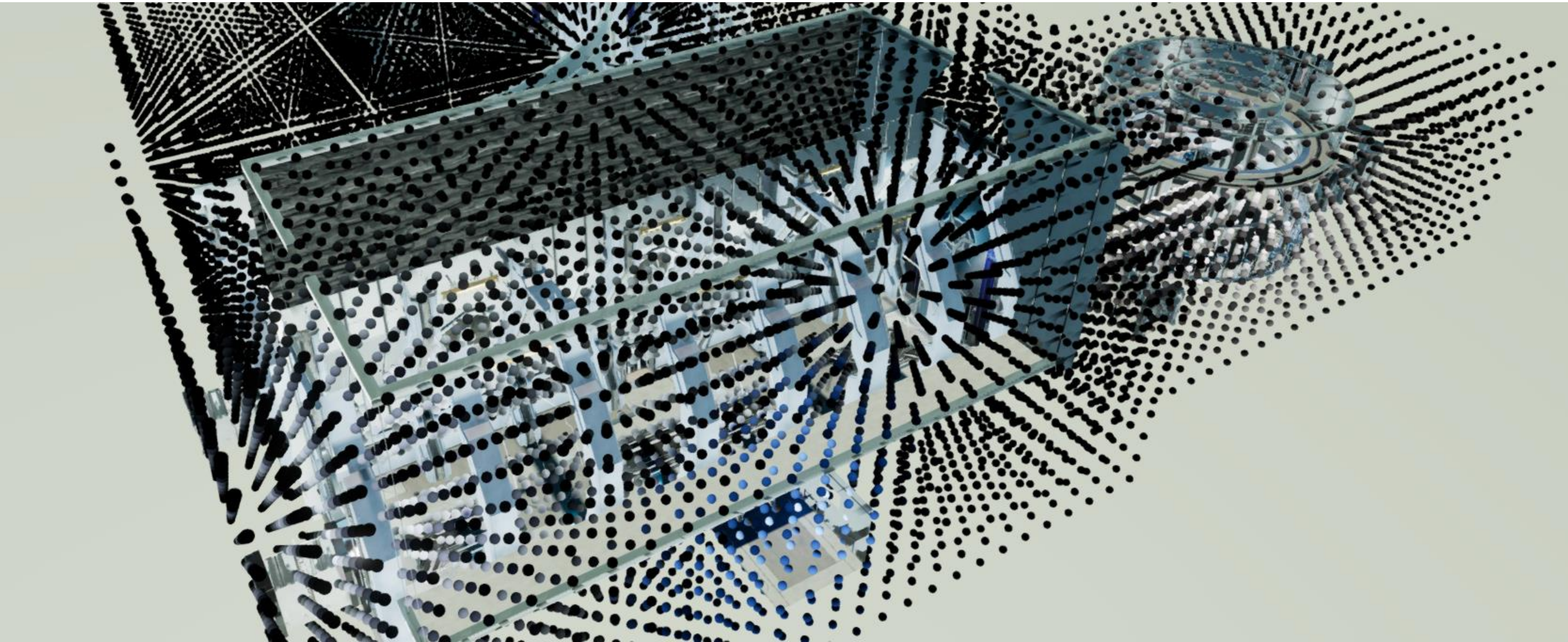
- Originally, only cascaded global volume
  - Follows camera

Cascade 0
Cascade 1
Cascade 2



Global vs. Local Volumes

- For interiors, we found fixed local volumes worked better
  - Higher quality
  - No need to inject & propagate every frame



Original LPV Occlusion

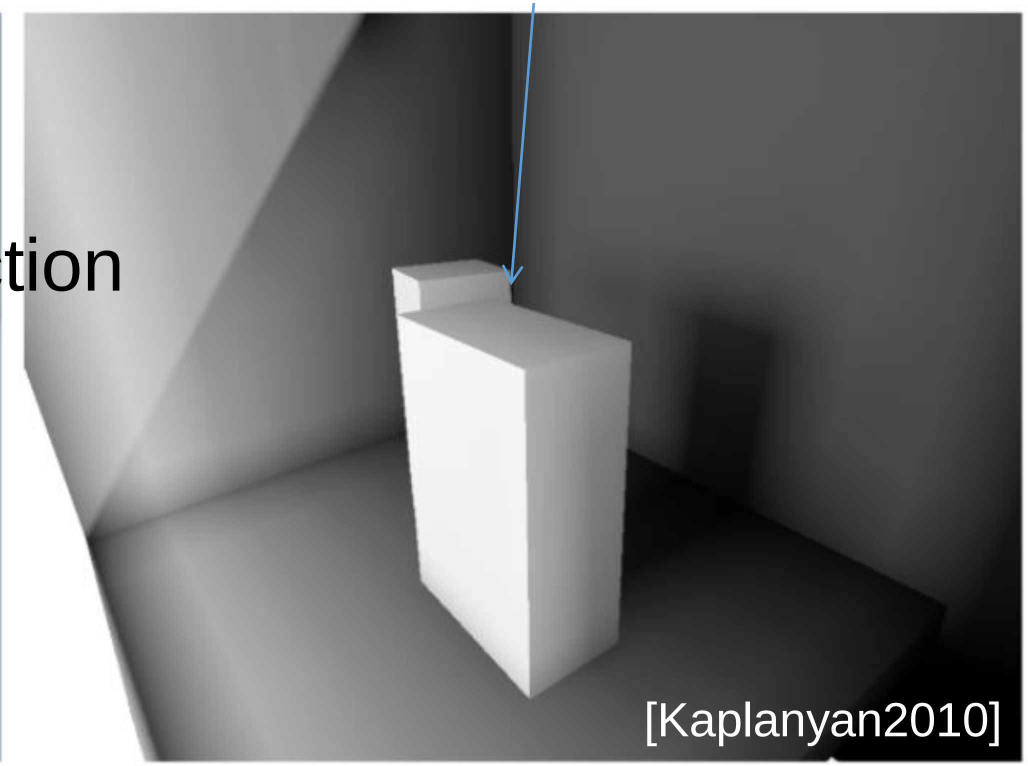
- Inject “occluders” into LPV volume from depth [Kaplanyan2010]
  - Main depth buffer
  - Auxiliary depth buffers (RSMs themselves, other shadow maps)
- Existed in original Confetti implementation

[Kaplanyan2010]

LPV Occlusion Problems

Light bleeding from coarse discretization

Missed geometry



Biggest Problems:

- Inconsistent results based on view direction
- Limited artist control!

Light Blockers for LPV

- Artists placing light blockers anyway, can use for GI too!

GI Only View

- Light blockers on

GI Only View

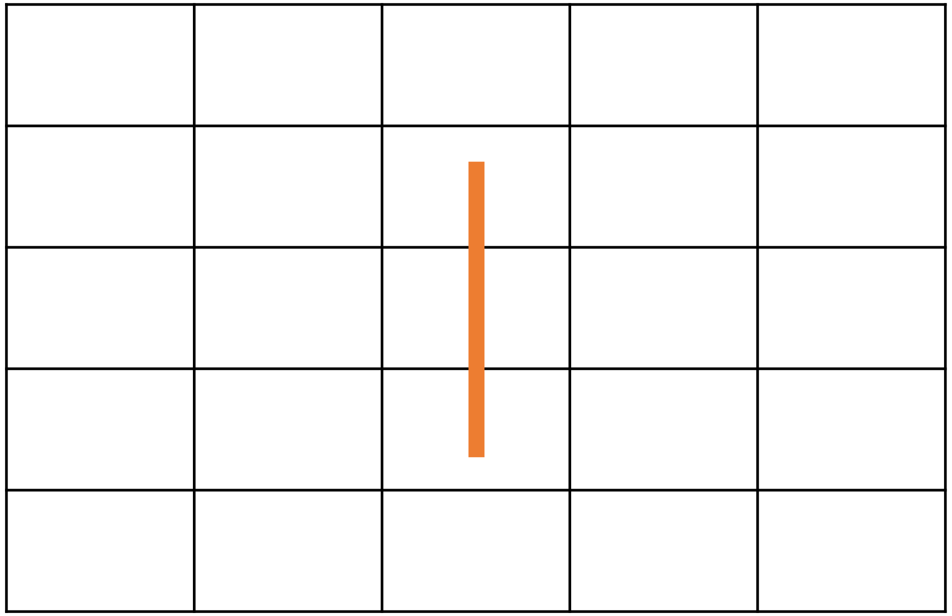
- Light blockers off

GI Only View

- Light blocker placement

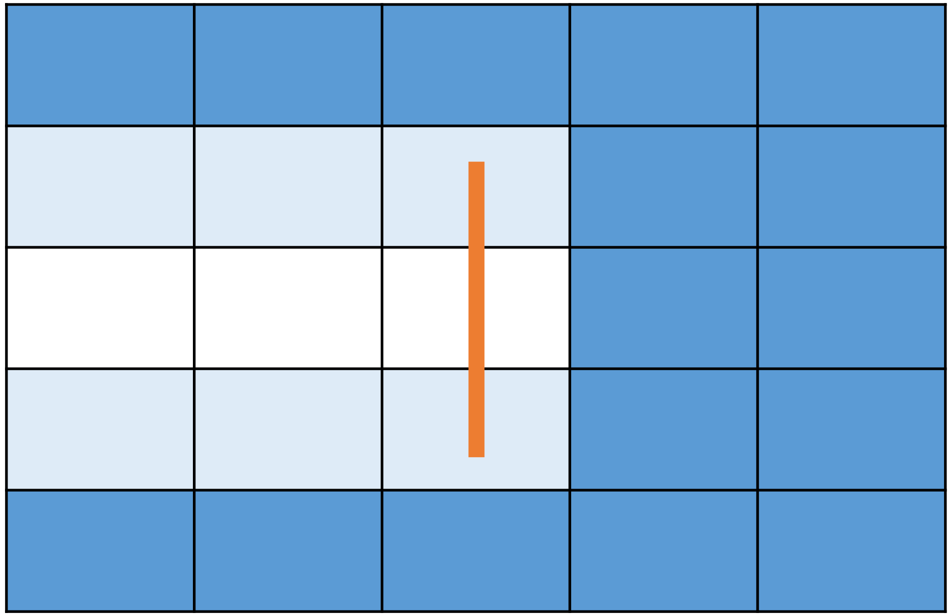
Blockers During Propagation

- Light blockers injected into volume
  - Stored as “axial” occlusion (amount of occlusion along each axis)



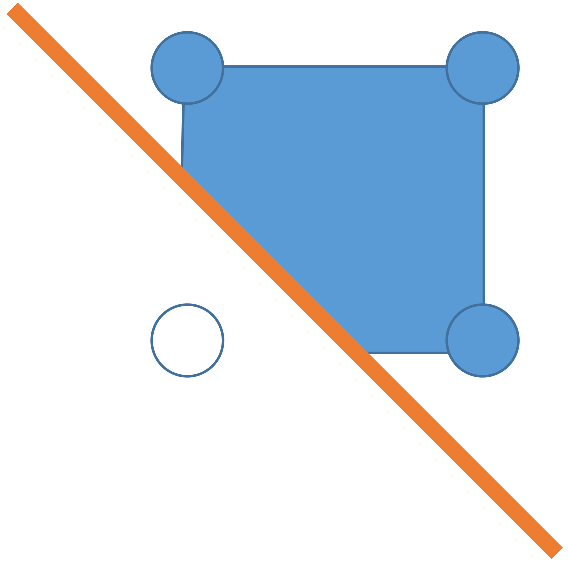
Blockers During Propagation

- Light blockers injected into volume
  - Stored as “axial” occlusion (amount of occlusion along each axis)
- Block light during propagation
  - Produces GI “shadows”



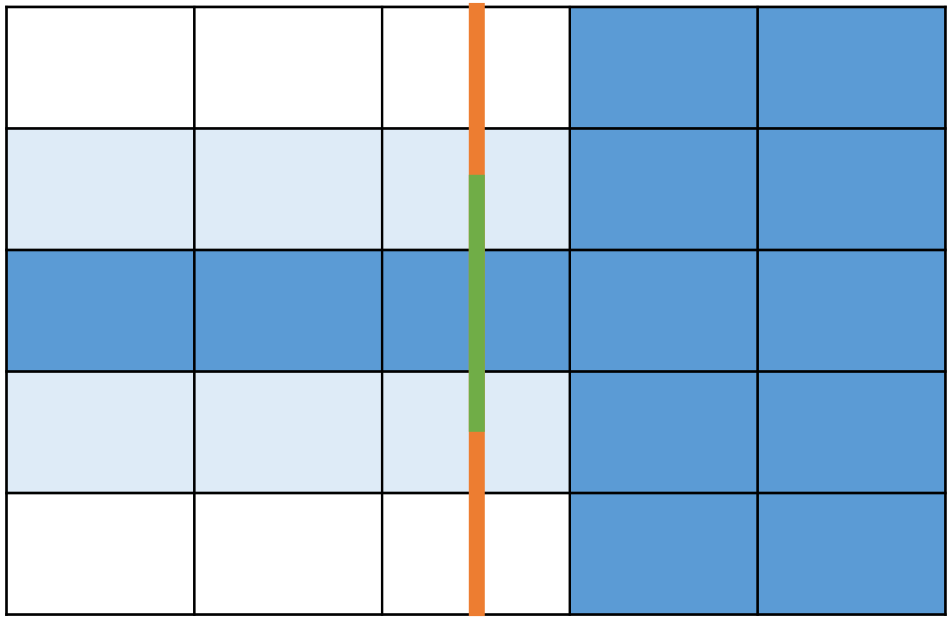
Blockers During Apply

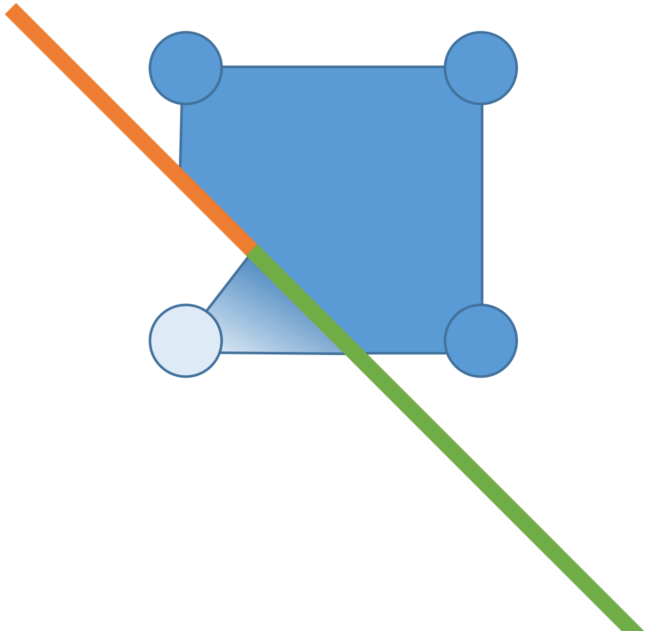
- Light blockers culled against 4x4x4 macro-cells
  - To reduce set of blockers considered in each LPV cell
- Block light from trilinear samples during apply
  - Eliminates light leaking from coarse grid

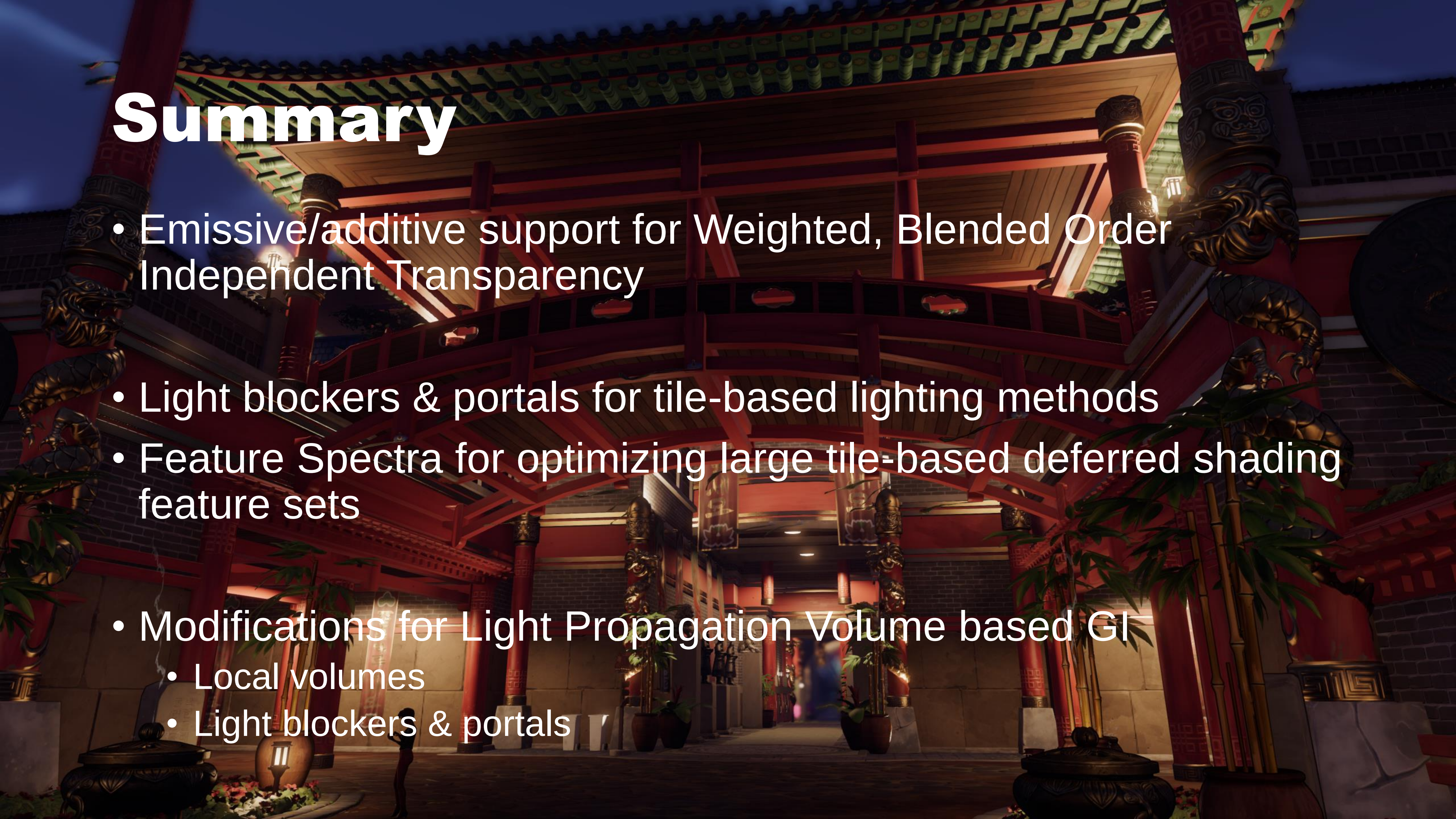


Light Portals for LPV

- Portals injected along with blockers as set of “holes” per blocker
- Modify axial occlusion for propagation
- Negate sample blockage in apply







Summary

- Emissive/additive support for Weighted, Blended Order Independent Transparency
- Light blockers & portals for tile-based lighting methods
- Feature Spectra for optimizing large tile-based deferred shading feature sets
- Modifications for Light Propagation Volume based GI
  - Local volumes
  - Light blockers & portals

Questions?

http://www.dsvolition.com/publications/

volition

References

- Andersson, *DirectX 11 Rendering in Battlefield 3*, Game Developers Conference, 2011
  - *https://www.slideshare.net/DICEStudio/directx-11-rendering-in-battlefield-3*
- Kaplanyan, Dachsbacher, *Cascaded Light Propagation Volumes for Real-Time Indirect Illumination,* Proceedings of the 2010 Symposium on Interactive 3D Graphics and Games.
  - *http://dl.acm.org/citation.cfm?id=1730821&CFID=989089912&CFTOKEN=24284118*
- McGuire, Bavoil, *Weighted Blended Order-Independent Transparency*, Journal of Computer Graphics Techniques, vol. 2, no. 2, 2013
  - *http://jcgt.org/published/0002/02/09/*
- McGuire, *Implementing Weighted, Blended Order-Independent Transparency*, Blog post, 2015
  - http://casual-effects.blogspot.com/2015/03/implemented-weighted-blended-order.html

volition http://www.dsvolition.com/

CONFETTI http://www.confettispecialfx.com/

Appendix

WBOIT implementation details + shader source code

Implementation

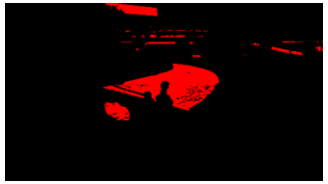
- MRT Setup

MRT0: FP16:16:16:16	**Red * Weight**	**Green * Weight**	**Blue * Weight**	**Weight**
MRT1: 8:8:8:8	Revealage	(unused)	(unused)	Additiveness

- Blend State
  - MRT0: (1)S + (1)D for all channels
  - MRT1: (0)S + (1-S)D for color channels, (1)S+(1)D for alpha channel
- Low-res and high-res alpha easily combined in composite
- See Appendix for Shader Source Code

CMASK Optimization

- Reading high-res targets can be expensive
- Fast-clear eliminate of high-res buffers also slow (~0.2ms)
- Read super-tiny CMASK buffer first and skip work if not written
  - Reduces “no-alpha” case from 0.7ms to 0.3ms on PS4



```
// This function is executed in alpha material shaders as the last step before writing out to the MRTs
void weighted_oit_process(out float4 accum, out float revealage, out float emissive_weight, float4 premultiplied_alpha_color, float raw_emissive_luminance, float view_depth, float current_camera_exposure)
{
const float opacity_sensitivity = 3.0; // Should be greater than 1, so that we only downweight nearly transparent things. Otherwise, everything at the same depth should get equal weight. Can be artist controlled
const float weight_bias = 5.0; //Must be greater than zero. Weight bias helps prevent distant things from getting hugely lower weight than near things, as well as preventing floating point underflow
const float precision_scalar = 10000.0;  //adjusts where the weights fall in the floating point range, used to balance precision to combat both underflow and overflow
const float maximum_weight = 20.0;  //Don't weight near things more than a certain amount to both combat overflow and reduce the "overpower" effect of very near vs. very far things
const float maximum_color_value = 1000.0;
const float additive_sensitivity = 10.0; //how much we amplify the emissive when deciding whether to consider this additively blended

// Exposure changes relative importance of emissive luminance (whereas it does not for opacity)
float relative_emissive_luminance = raw_emissive_luminance * current_camera_exposure;

//Emissive sensitivity is hard to pin down
//On the one hand, we want a low sensitivity so we don't get dark halos around "feathered" emissive alpha that overlap with eachother
//On the other hand, we want a high sensitivity so that dim emissive holograms don't get overly downweighted.
//We expose this to the artist to let them choose what is more important.
const float emissive_sensitivity = 1.0/<<artist controlled value between 0.01 and 1>>;

float clamped_emissive = saturate(relative_emissive_luminance);
float clamped_alpha = saturate(premultiplied_alpha_color.a);

// Intermediate terms to be cubed
// NOTE: This part differs from McGuire's sample code:
// since we're using premultiplied alpha in the composite, we want to
// keep emissive values that have low coverage weighted appropriately
// so, we'll add the emissive luminance to the alpha when computing the alpha portion of the weight
// NOTE: We also don't add a small value to a, we allow it to go all the way to zero, so that completely invisible portions do not influence the result
float a = saturate((clamped_alpha*opacity_sensitivity) + (clamped_emissive*emissive_sensitivity));

// NOTE: This differs from McGuire's sample code. In order to avoid having to tune the algorithm separately for different
// near/far plane values, we produce a "canonical" depth value from the view-depth, using an fixed near plane and a tunable far plane
const float canonical_near_z = 0.5;
const float canonical_far_z = 300.0;
float range = canonical_far_z-canonical_near_z;
float canonical_depth = saturate(canonical_far_z/range - (canonical_far_z*canonical_near_z)/(view_depth*range));
float b = 1.0 - canonical_depth;

// clamp color to combat overflow (weight will be clamped too)
float3 clamped_color = min(premultiplied_alpha_color.rgb, maximum_color_value);

float w = precision_scalar * b * b * b; //basic depth based weight
w += weight_bias; //NOTE: This differs from McGuire's code. It is an alternate way to prevent underflow and limits near/far weight ratio
w = min(w, maximum_weight); //clamp by maximum weight BEFORE multiplying by opacity weight (so that we'll properly reduce near faint stuff in weight)
w *= a * a * a; //incorporate opacity weight as the last step

accum     = float4(clamped_color*w, w); //NOTE: This differs from McGuire's sample code because we want to be able to handle fully additive alpha (e.g. emissive), which has a coverage of 0 (revealage of 1.0)
revealage = clamped_alpha; //blend state will invert this to produce actual revealage
emissive_weight = saturate(relative_emissive_luminance*additive_sensitivity)/8.0f; //we're going to store this into an 8-bit channel, so we divide by the maximum number of additive layers we can support
}
```

```
// Full-screen composite pixel shader
PS_OUTPUT main_ps(VS_OUTPUT input)
{
uint3 ipos = uint3(input.pos.xy, 0);

#if (defined(_PS4) || defined(_XBOX3)) && defined(USE_CMASK_OPT)
    // skip some work for pixels that we didn't write to at all
    const bool hires_written = decoded_cmask.Load(uint3(ipos.x/4,ipos.y/4,0))!=0.0f;
#else
    const bool hires_written = true;
#endif

float revealage = 1.0;
float additiveness = 0.0;
float4 accum = float4(0.0,0.0,0.0,0.0);

// high-res alpha
[branch]
if(hires_written) {
    float4 temp = input_accum2.Load(ipos);
    revealage = temp.r;
    additiveness = temp.w;
    accum = input_accum1.Load(ipos);
}

// low-res alpha
float4 temp = input_accum2_subpass.SampleLevel(Sampler_filter_clamp, input.uv, 0);
revealage = revealage * temp.r;
additiveness = additiveness + temp.w;

accum = accum + input_accum1_subpass.SampleLevel(Sampler_filter_clamp, input.uv, 0);

// weighted average (weights were applied during accumulation, and accum.a stores the sum of weights)
float3 average_color = accum.rgb / max(accum.a, 0.00001);

// Amplify based on additiveness to try and regain intensity we lost from averaging things that would formerly have been additive.
// Revealage gives a rough estimate of how much "alpha stuff" there is in the pixel, allowing us to reduce the additive amplification when mixed in with non-additive
float emissive_amplifier = (additiveness*8.0f); //The constant factor here must match the constant divisor in the material shaders!
emissive_amplifier = lerp(emissive_amplifier*0.25, emissive_amplifier, revealage); //lessen, but do not completely remove amplification when there's opaque stuff mixed in

// Also add in the opacity (1-revealage) to account for the fact that additive + non-additive should never be darker than the non-additive by itself
emissive_amplifier += saturate((1.0-revealage)*2.0); //constant factor here is an adjustable thing to indicate how "sensitive" we should be to the presence of opaque stuff

average_color *= max(emissive_amplifier,1.0); // NOTE: We max with 1 here so that this can only amplify, never darken, the result

// Suppress overflow (turns INF into bright white)
if (any(isinf(accum.rgb))) {
    average_color = 100.0f;
}

PS_OUTPUT OUT;
OUT.Color0 = float4(average_color, 1.0 - revealage);

return OUT;
}
```

Additional Bonus Slide

Light Blockers/Portals LDS Memory Analysis for Lighting Compute

Some Rough Numbers

- Max lights per tile: 64
- Max blockers per light: 32
- Max portals per light: 32
- Max portals per blocker: 32
- Max (light,portal) or (light,blocker) pairs per tile: 256
- Groupshared (LDS) memory requirements:
  - Initial & final lights in tile: 512 bytes
  - Various (light,blocker)/(light,portal) bitarrays: 1280 bytes
  - + Other miscellaneous counts, etc…
  - Total: ~2KB (max theoretical PS4 occupancy: 8 wavefronts/SIMD)