GPU TECHNOLOGY CONFERENCE

HIGH-QUALITY RASTERIZATION

CHRIS WYMAN

SENIOR RESEARCH SCIENTIST, NVIDIA

GPU TECHNOLOGY CONFERENCE

NEW RASTER METHODS USING MAXWELL

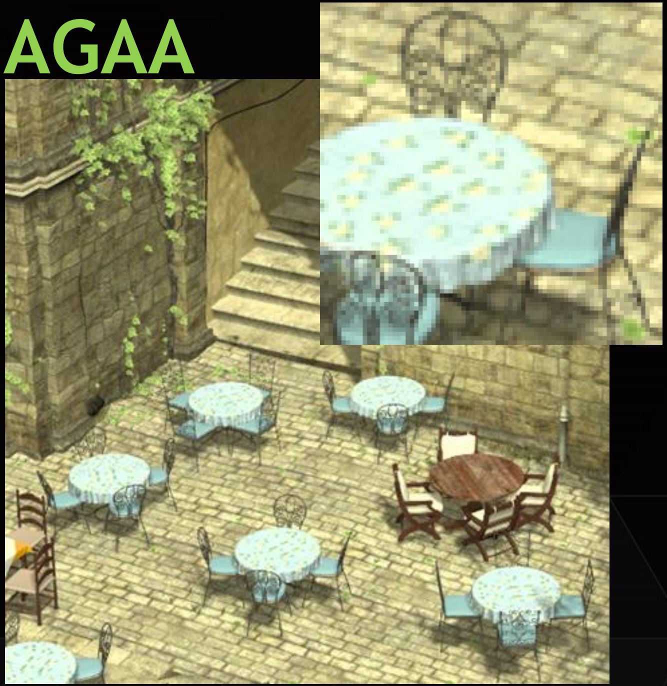
- Accumulative Anti-Aliasing (ACAA)
  - A simple improvement on forward MSAA using less memory and bandwidth
- Aggregate Anti-Aliasing (AGAA)
  - Create statistical aggregates from similar surfaces' G-buffer samples
  - Shade just once per aggregate, reducing shades per pixel and bandwidth costs
- Frustum-Traced Irregular Z-Buffer (FTIZB)
  - A raster method to render ray traced quality, 32 sample-per-pixel hard shadows
  - No spatial or temporal aliasing

COMMONALITIES

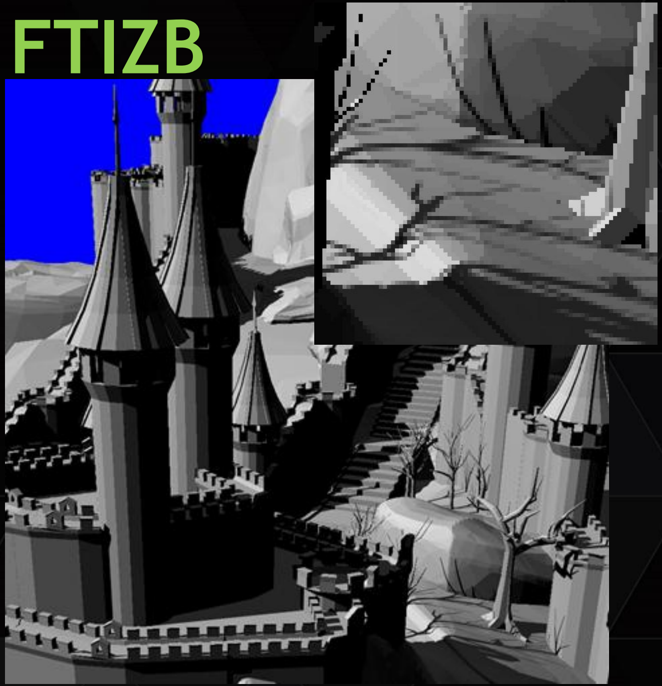
- High quality antialiasing
  - 8 samples or higher per pixel, lower cost than prior methods

ACAA

AGAA



FTIZB



COMMONALITIES

- High quality antialiasing
  - 8 samples or higher per pixel, lower cost than prior methods

- Leverage new Maxwell GPU features
  - Fast geometry shader (aka NV_geometry_shader_passthrough)
  - Target independent raster (aka NV_framebuffer_mixed_samples)
  - Post-depth coverage (aka EXT_post_depth_coverage)
  - Conservative rasterization (aka NV_conservative_raster)
  - Sample mask override (aka NV_sample_mask_override_coverage)

Accumulative Anti-Aliasing

Work by:

Eric Enderton, Eric Lum, Christian Rouet,

and Oleg Kouznetsov

WHAT IS ANTI-ALIASING?

$$F = \sum w_i c_i$$

- Usually:
  - Sample multiple times per pixel
  - Resolve to final color by appropriately weighting each color sample

ACCUMULATIVE ANTI-ALIASING (ACAA)

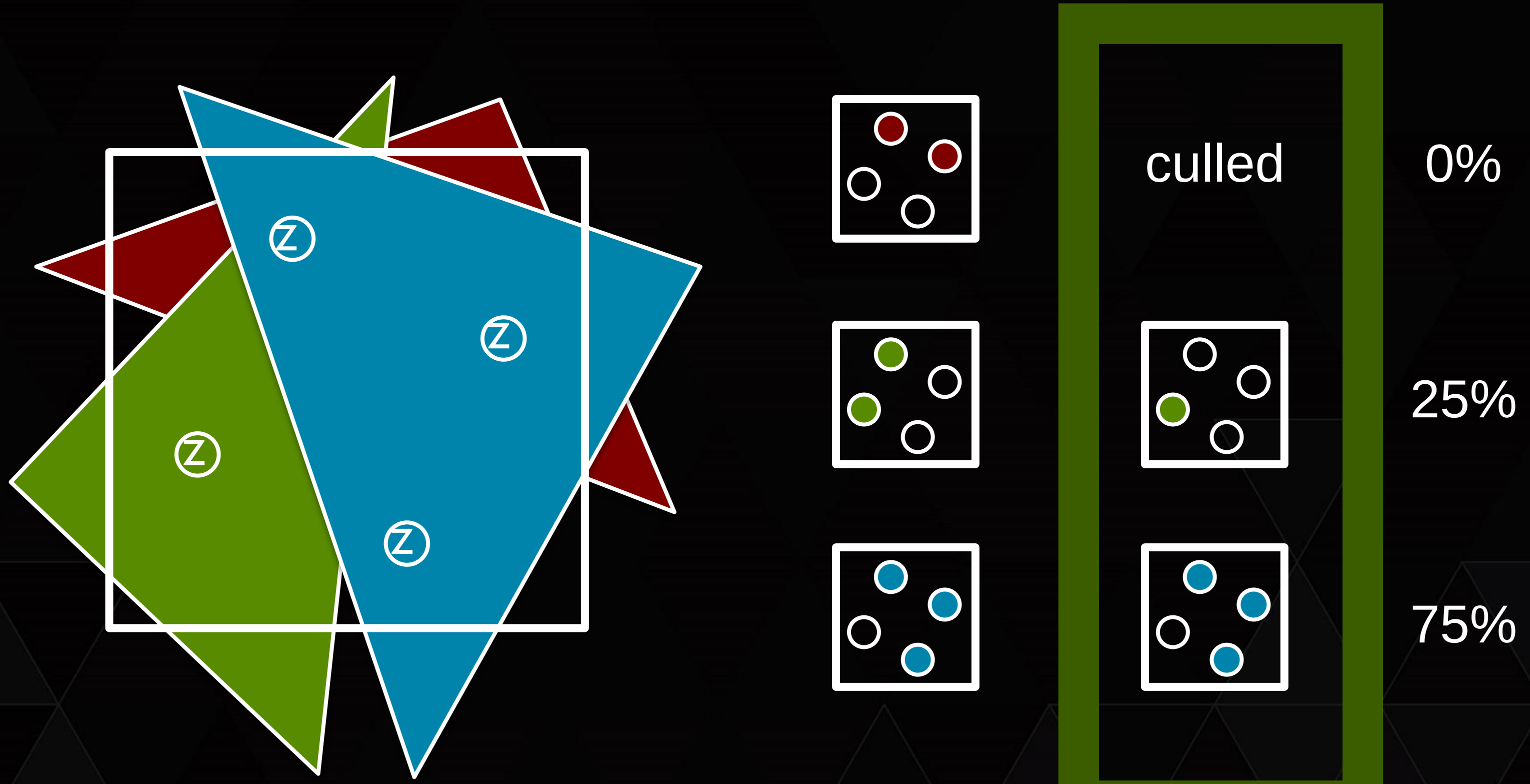
$$F = \sum w_i c_i$$

- Key ACAA insight:
  - If we pre-compute visibility (i.e., the weights),
  - Only need to store one color per pixel (the accumulated color)
- Gives full MSAA quality using alpha blending

ACCUMULATIVE ANTI-ALIASING (ACAA)

- Why don’t people already do this?
  - MSAA weight depends on samples covered
  - Not known in forward renderer until all geometry rendered
- ACAA does a z-prepass
  - Precomputes visibility, storing the closest surface per sample
  - During shading pass ask, “how many samples passed the z-test?”
- Requires shader to know post-z sample coverage
  - New with Maxwell GPUs

POST-Z COVERAGE




EXAMPLE OF 8X ACAA
Scene courtesy of Kishonti Informatics

COMPARED TO 8X MSAA

Scene courtesy of Kishonti Informatics

ACAA ALGORITHM

- Z prepass at 8x
- Rasterize at 8x
  - Shader uses post-z coverage to weight the fragment color
  - Accumulate into 1x color buffer

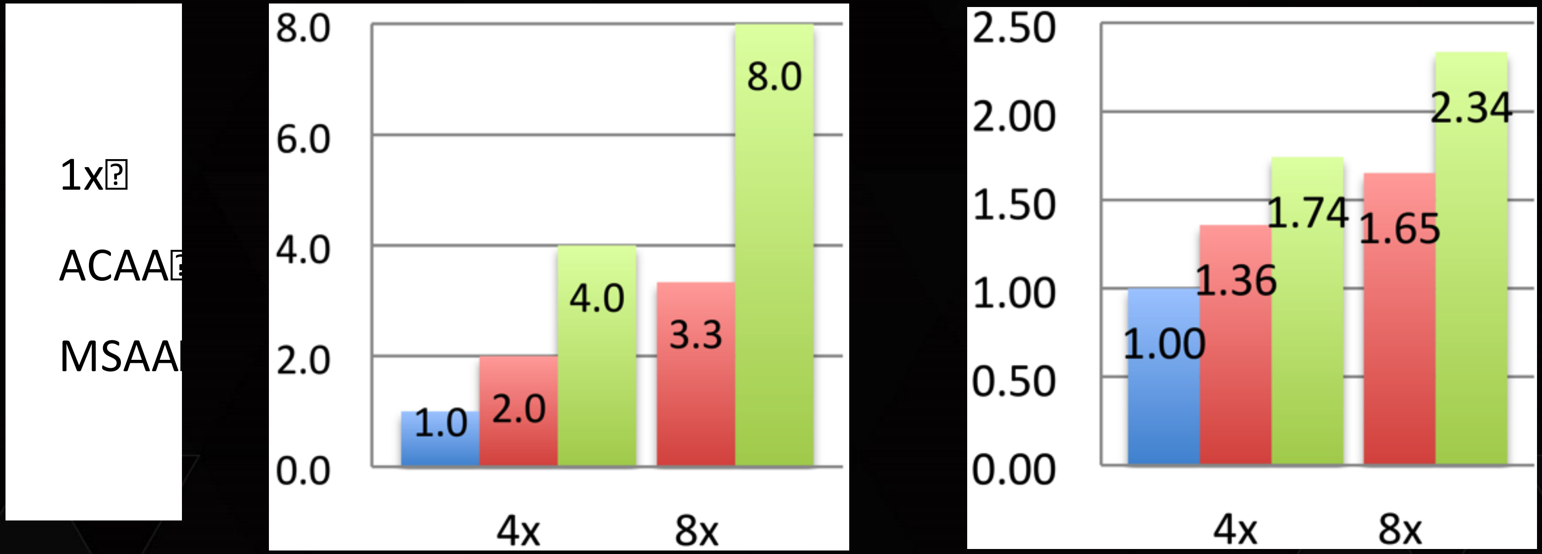
→ Same image quality as MSAA (*)

→ Less memory (in color buffer)

→ Less bandwidth (to color buffer)

ACAA BENEFITS

- Recovers most of the performance penalty of MSAA.



Memory required

Rendering cost

ACAA CAVEATS

- Assumes z test during shade passes only one fragment per sample
  - Fails when z-fighting occurs
  - Usually not an issue at 24-bit depths
  - Stenciling or saturated alpha blend can solve

- Transparency is not handled

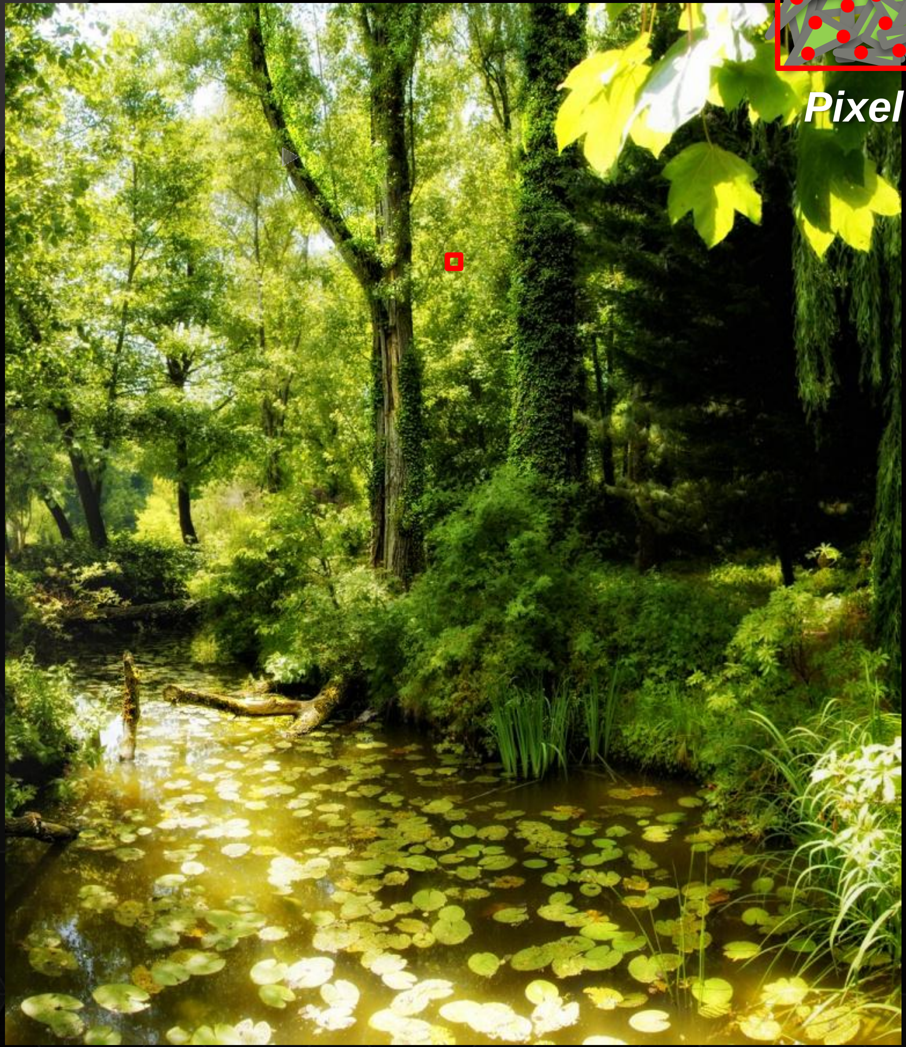
- Easily tested; part of NVIDIA GameWorks SDK:
  - https://github.com/NVIDIAGameWorks/OpenGLSamples
  - Sample called “Blended Antialising”

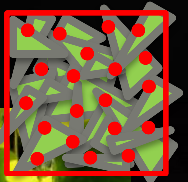
Aggregate G-Buffer Anti-Aliasing

Work by:

Cyril Crassin, Morgan McGuire, Kayvon Fatahalian and Aaron Lefohn

MOTIVATION





Pixel

Photograph © di9.in

The Mummy – [© Digital Domain / Rhythm&Hues]

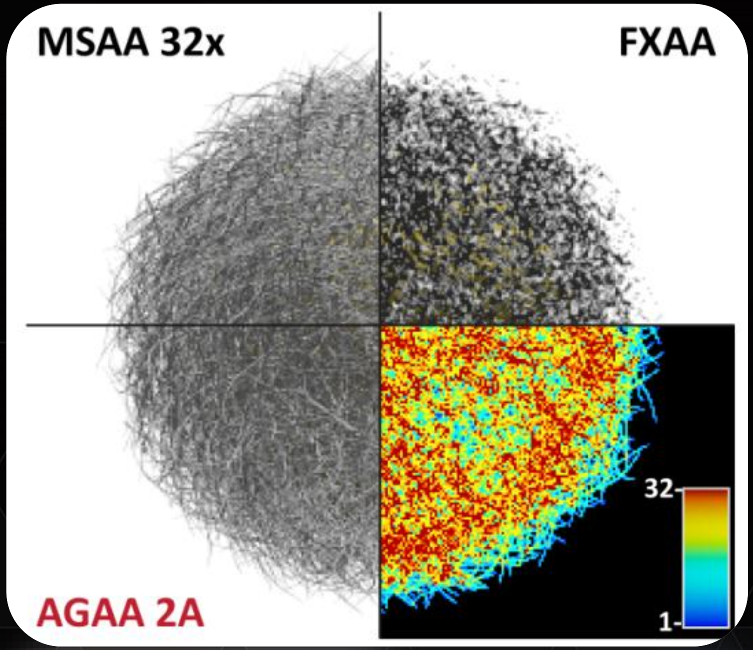
GPU TECHNOLOGY CONFERENCE

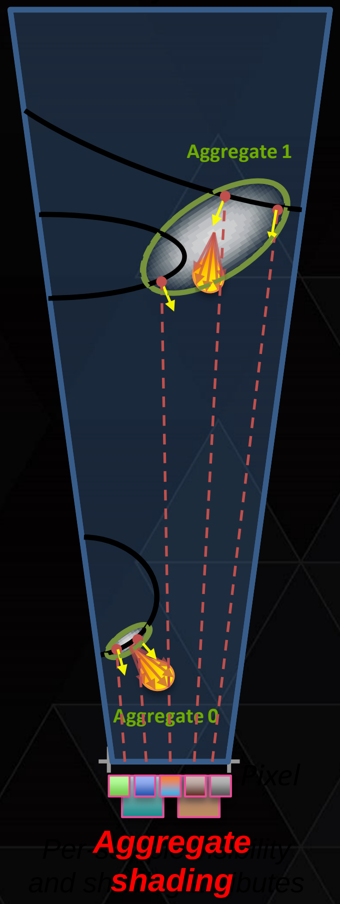
OVERVIEW

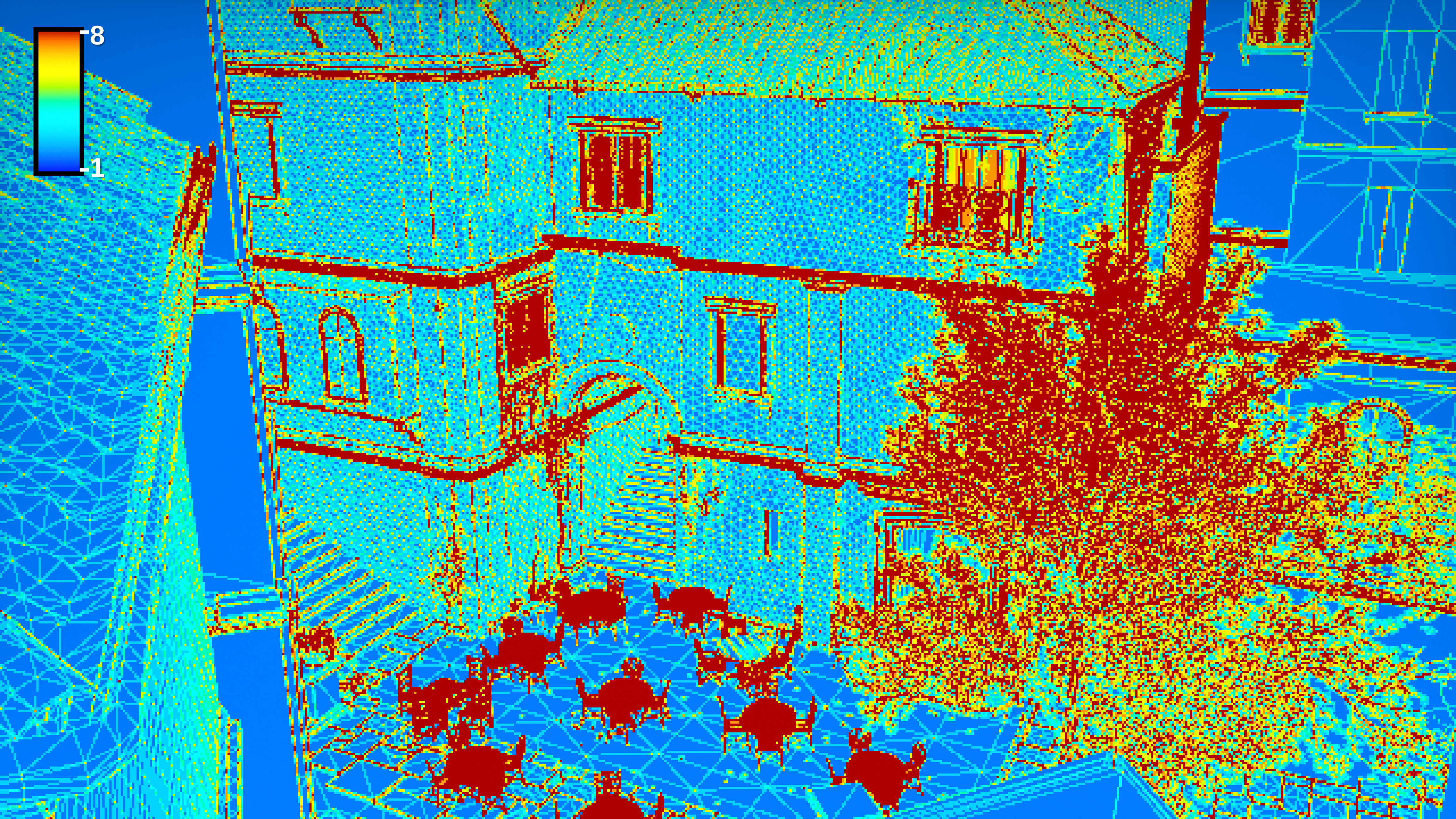
- High frequency shading is too costly
- Idea: Decouple shading rate from geometry
  - Shade statistical geometry distributions

OVERVIEW

- High frequency shading is too costly
- Idea: Decouple shading rate from geometry
  - Shade statistical geometry distributions







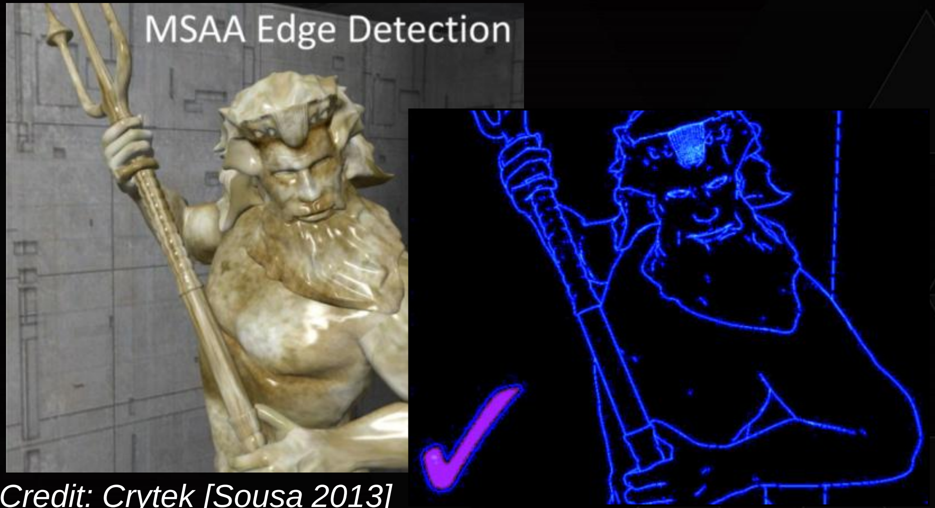
8
1


Single-sampled
MSAA 8x
AGAA (2A)


Single-sampled
MSAA 8x
AGAA (2A)

SIMILAR WORK

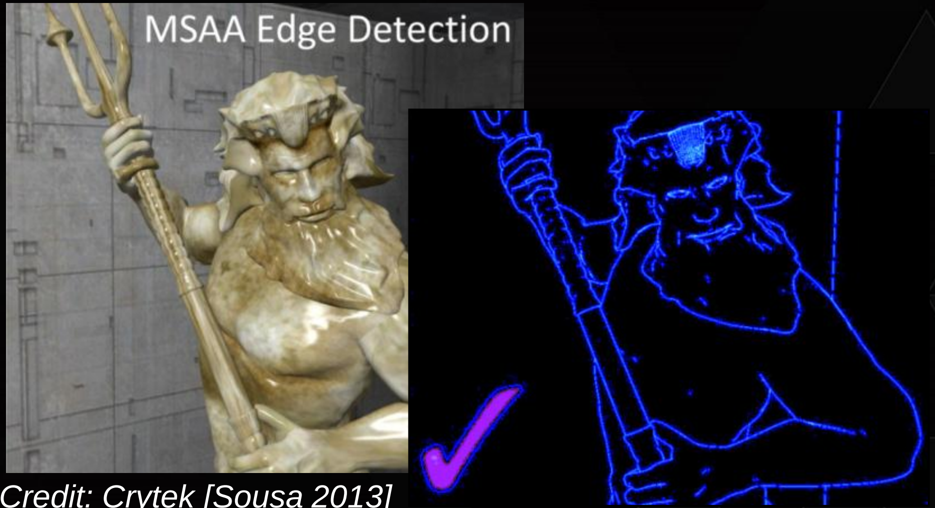
- Simple/Complex [Lauritzen 2010]
  - Segment image on per-pixel geometric complexity
  - Shade per-pixel for simple, per-sample for complex
  - Breaks when all pixels are complex



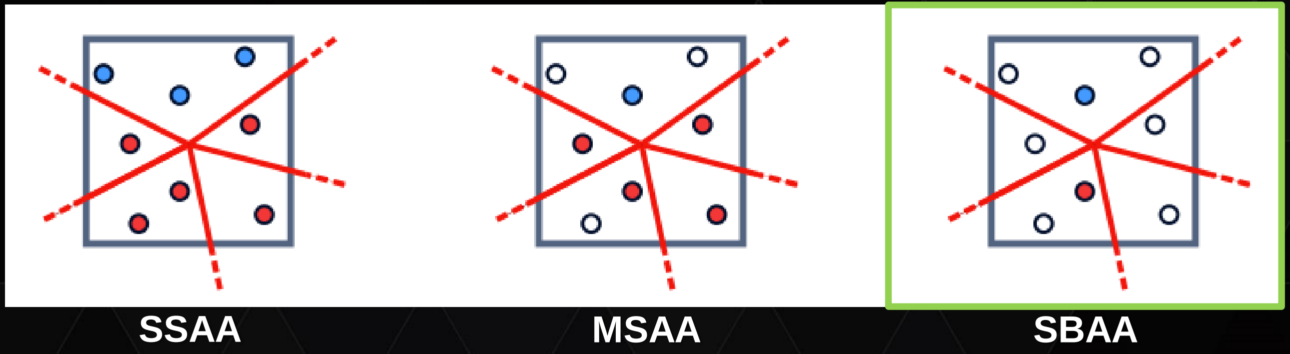
Credit: Crytek [Sousa 2013]

SIMILAR WORK

- Simple/Complex [Lauritzen 2010]
  - Segment image on per-pixel geometric complexity
  - Shade per-pixel for simple, per-sample for complex
  - Breaks when all pixels are complex
- Surface Based Anti-Aliasing (SBAA) [Salvi 2012]
  - Evaluate visibility per-sample in prepass
  - Only store and shade *N* most important surfaces
  - Discard other surfaces



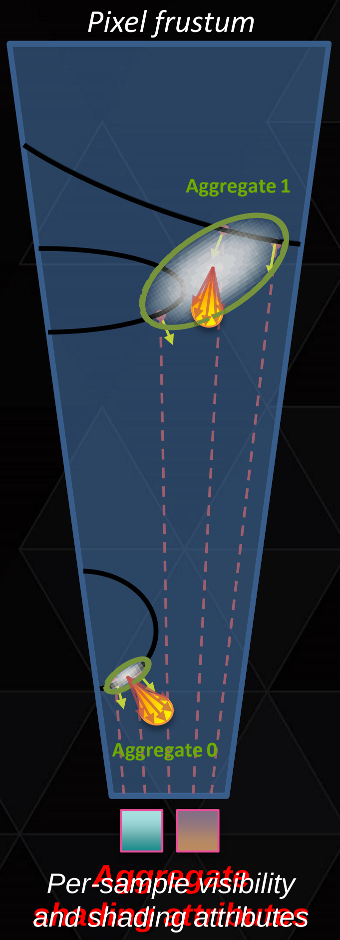
Credit: Crytek [Sousa 2013]



[Salvi 2012]

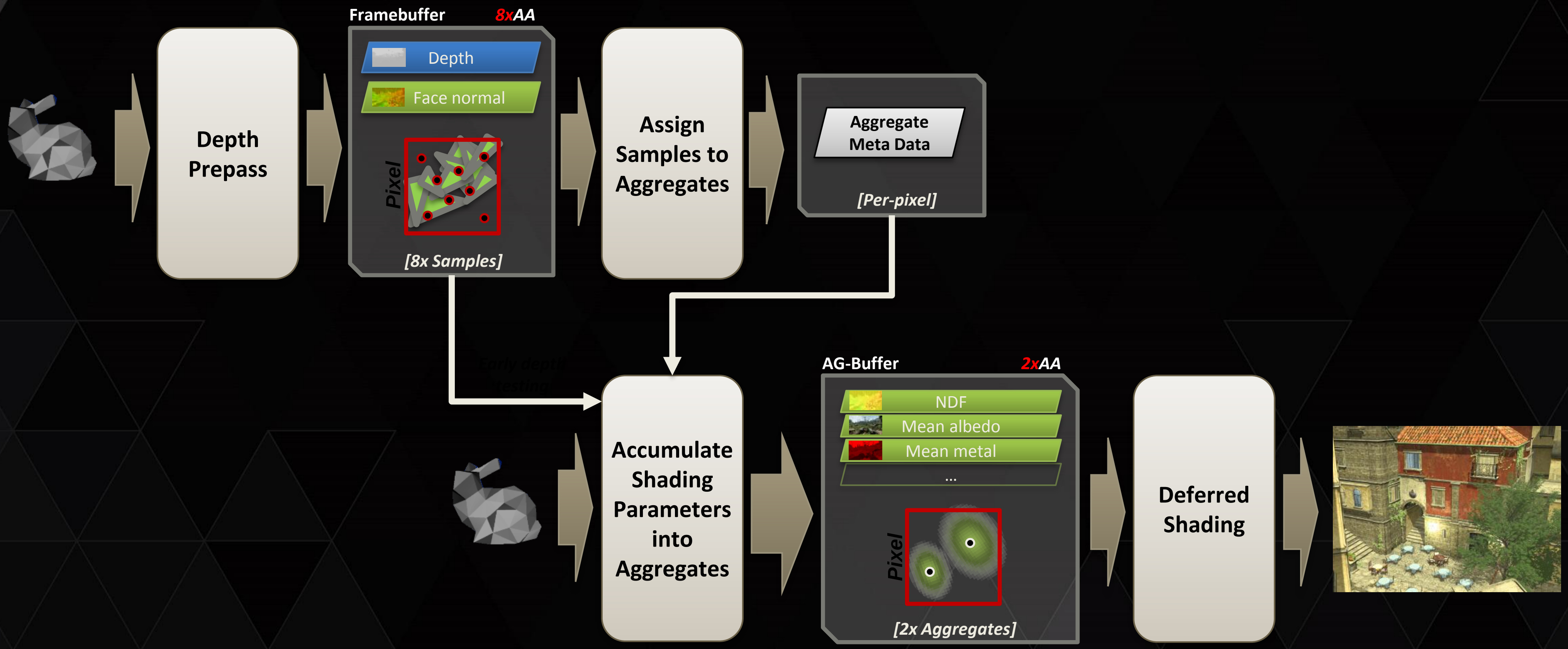
AGAA OVERVIEW

- Aggregate geometry in pixel-space before shading
  - Per-sample visibility (via Z-prepass)
  - Pre-filter shading attributes into aggregate g-buffer
    - Filter & aggregate on the fly
    - Inspired by texture pre-filtering
  - Aggregates store:
    - Normal distrib (NDF) & sub-pixel sample positions
    - Average albedo, specular coef, emissive, other mat’l info

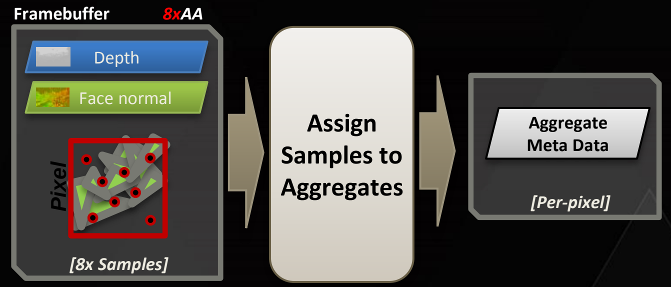


RENDERING WITH AGGREGATES

RENDERING WITH AGGREGATES

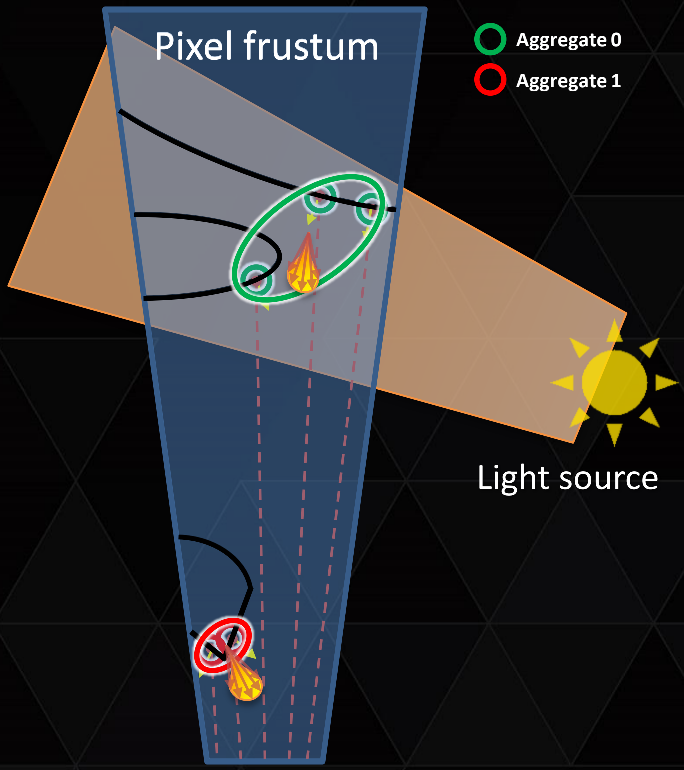


AGGREGATE DEFINITION



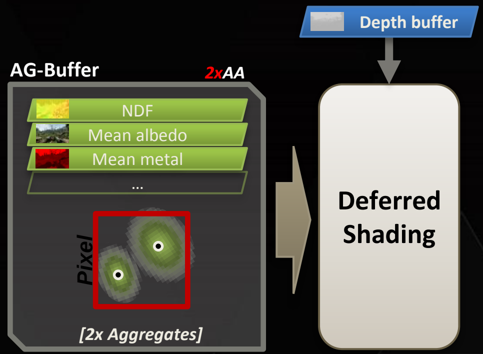
- Assign each visibility sample to one aggregate:
  - Allow for cross-primitive aggregates
  - Allow for aggregates over disjoint surfaces

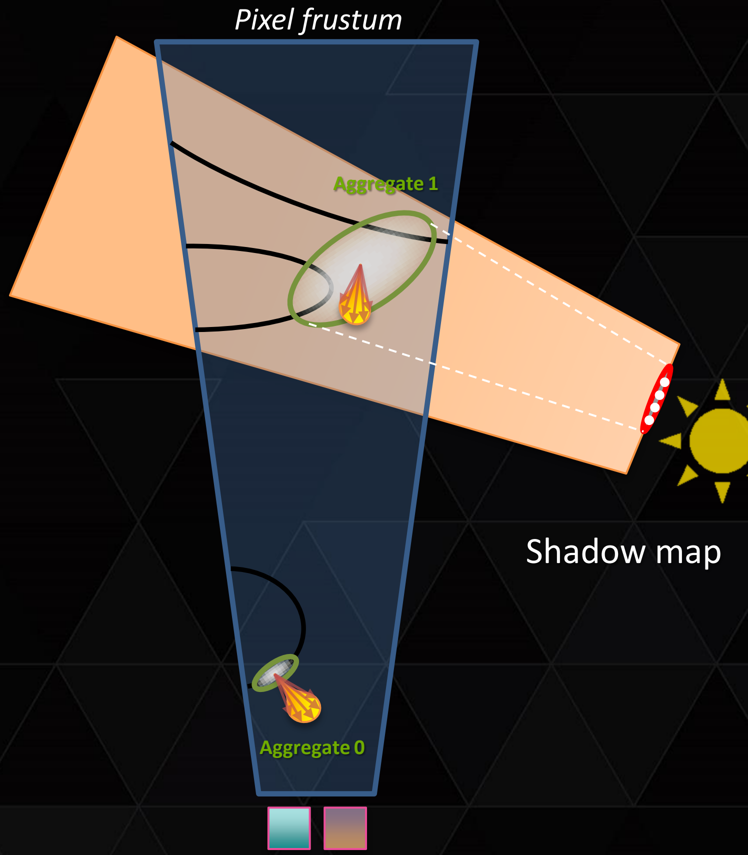
- **Goal:** Minimize errors from correlated attribs [Bruneton and Neyret 2012]
  - Prefer to cluster samples with similar normals
  - Prefer to cluster samples with similar shadows
    - Expensive to compute
    - Approx. with distance metric, assuming low frequency shadows



DEFERRED SHADING

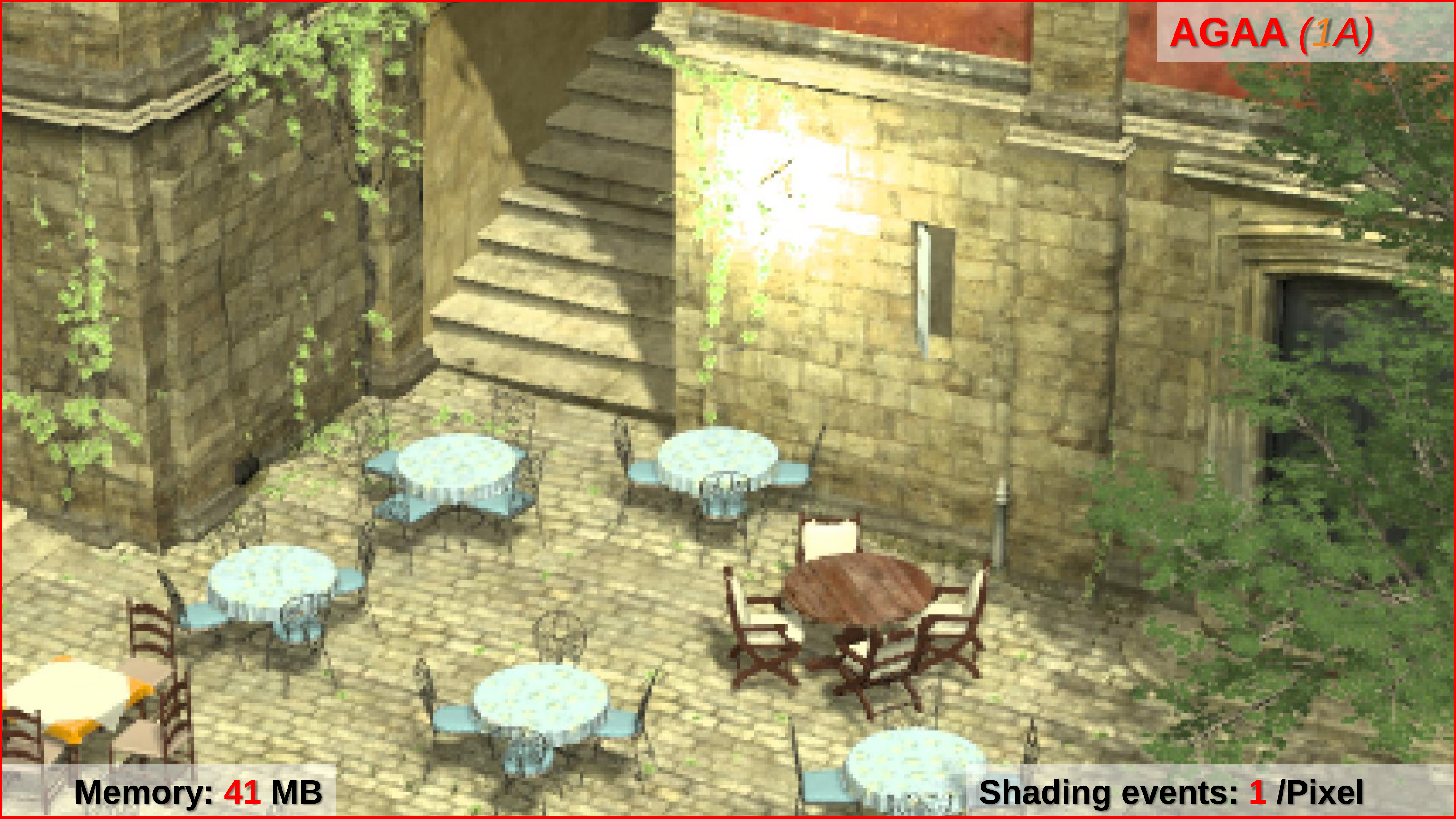
- Similar to using filtered textures for inputs to shade
  - AGAA is *independent* from the shading model
  - Assumes model inputs are linearly filterable
- Prototype uses Blinn-Phong BRDF model
  - Filtering specular component via Toksvig [Toksvig 2005]
  - Analytic approx. from Toksvig for diffuse [Baker and Hill 2012]
- Shadowing must be filtered
  - Account for aggregate depth extent
  - Avoids temporal issues when sample's cluster changes

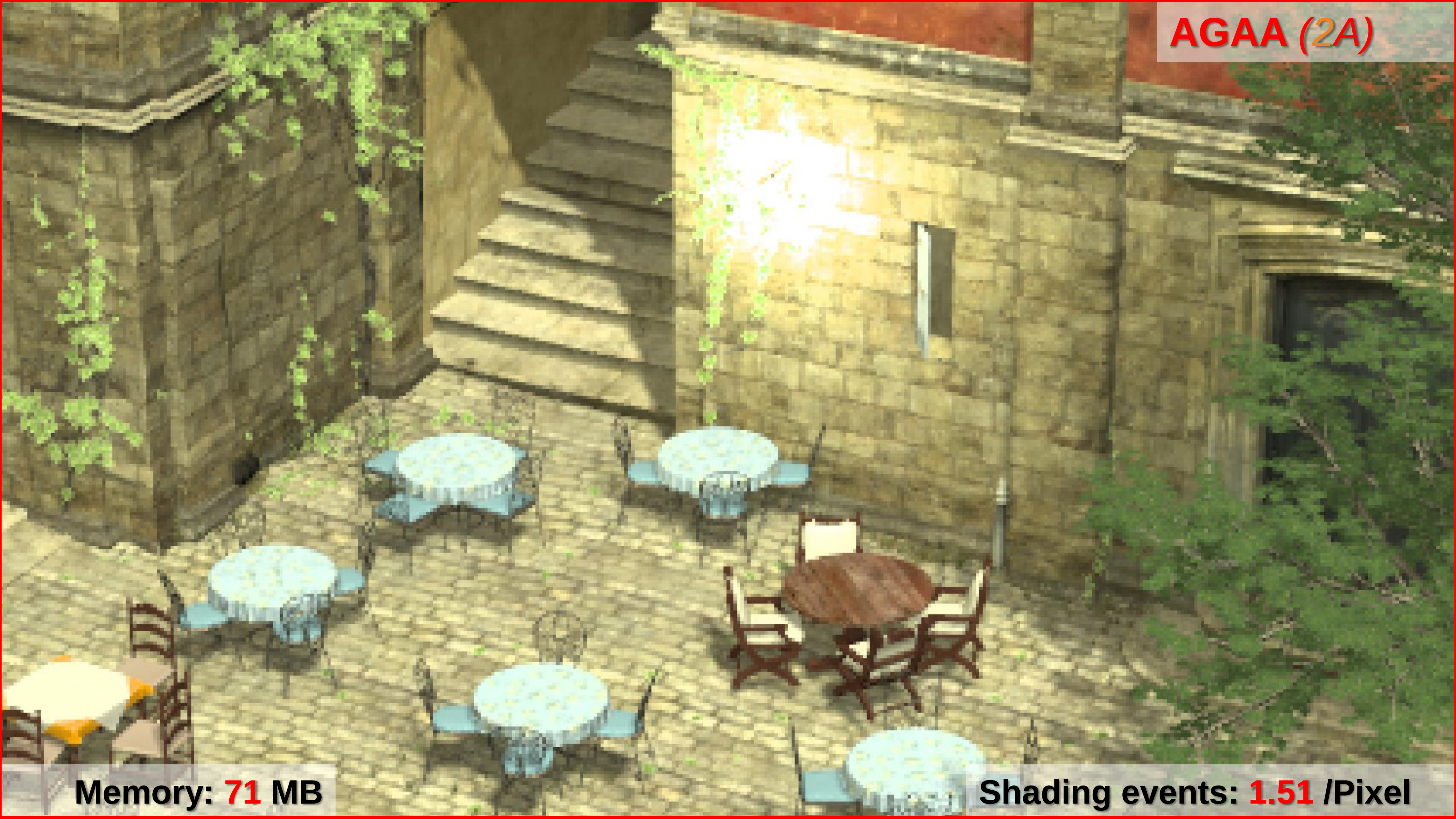




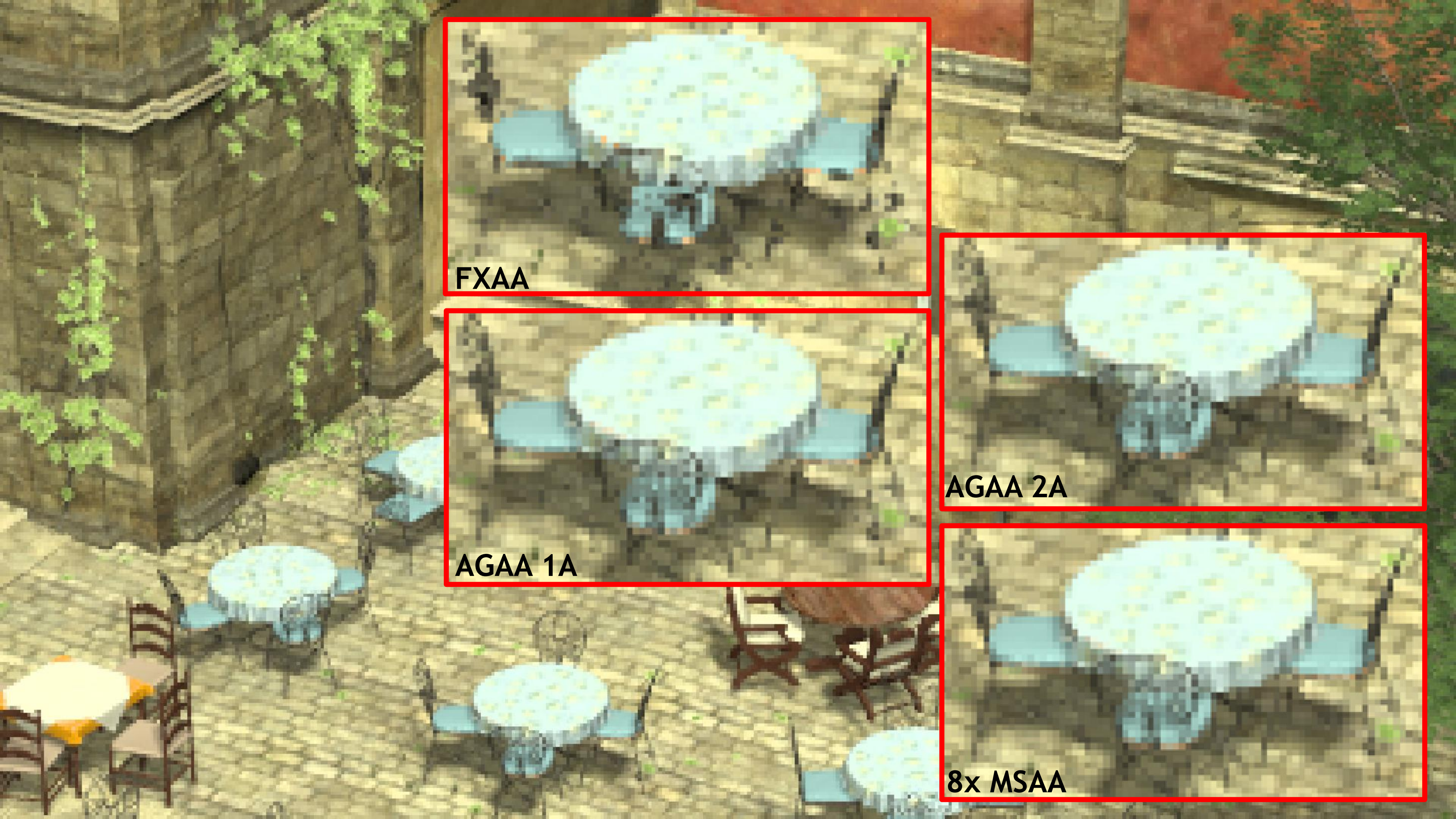

Deferred 8x (reference)


FXAA


AGAA (1A)
Memory: 41 MB
Shading events: 1 /Pixel


AGAA (2A)
Memory: 71 MB
Shading events: 1.51 /Pixel

Deferred 8x (reference)
Memory: 112 MB
Shading events: 6.68 /Pixel (Simple/Complex)


FXAA
AGAA 1A
AGAA 2A
8x MSAA

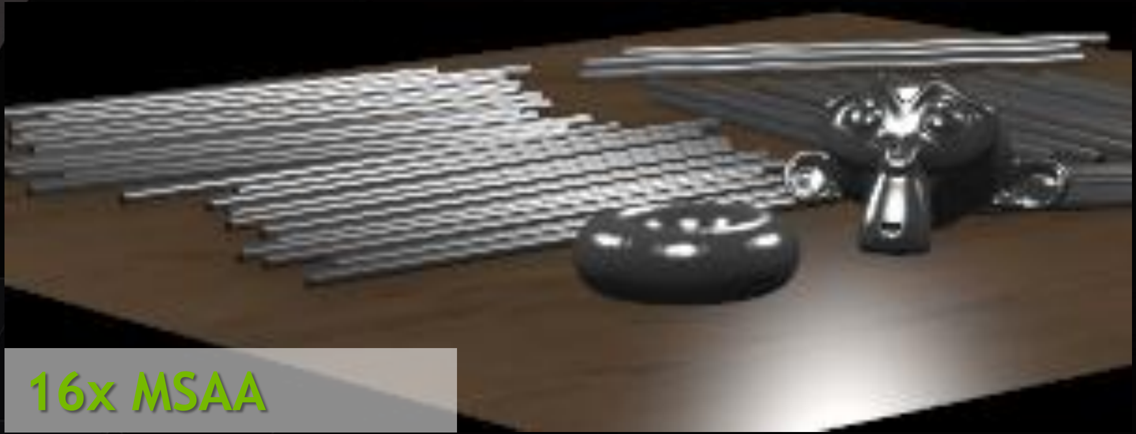

MSAA 32x
SBAA (2S)
AGAA (2A)

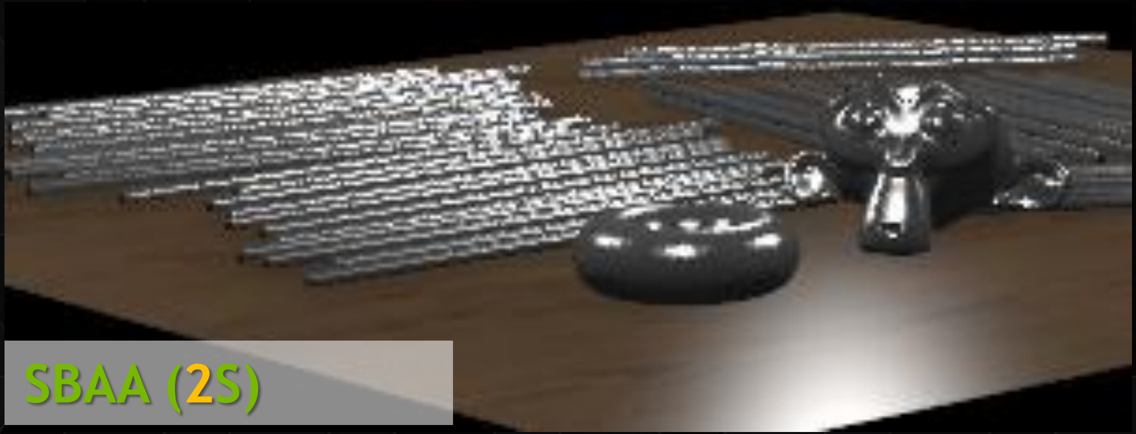
Old City (8x Zoom)

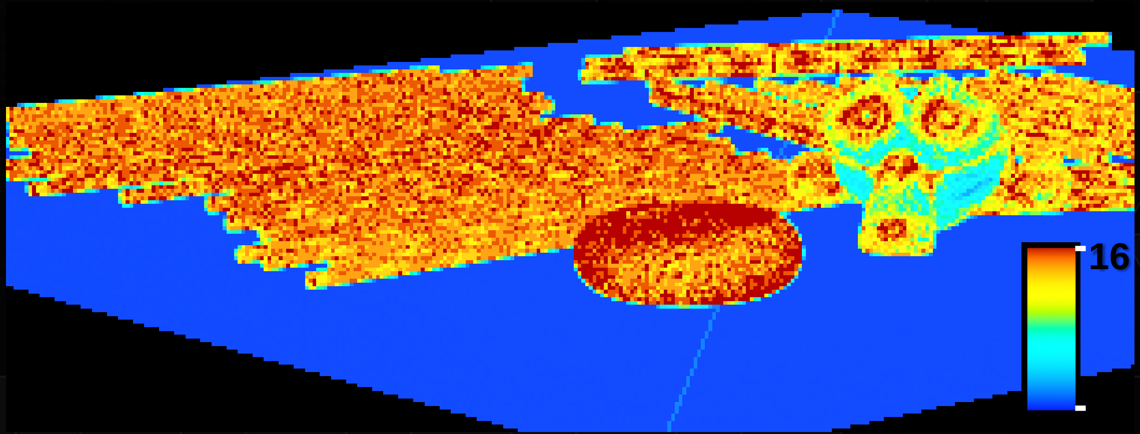
AGAA vs Deferred MSAA (Simple/Complex)

1280x736, 8x MSAA
NVIDIA GeForce GTX 980
Average performance:
AGAA, 2 aggregates: ~146 FPS
MSAA: ~125 FPS

SPECULAR ALIASING







RESULTS: PERFORMANCE

Deferred shading @8x MSAA 720p - Comparison with Simple/Complex [Lauritzen 2010] - NVIDIA GTX980 (Maxwell GM204)

Old City

54% Faster rendering than **Simple/Complex**

(2.84x Faster shading)

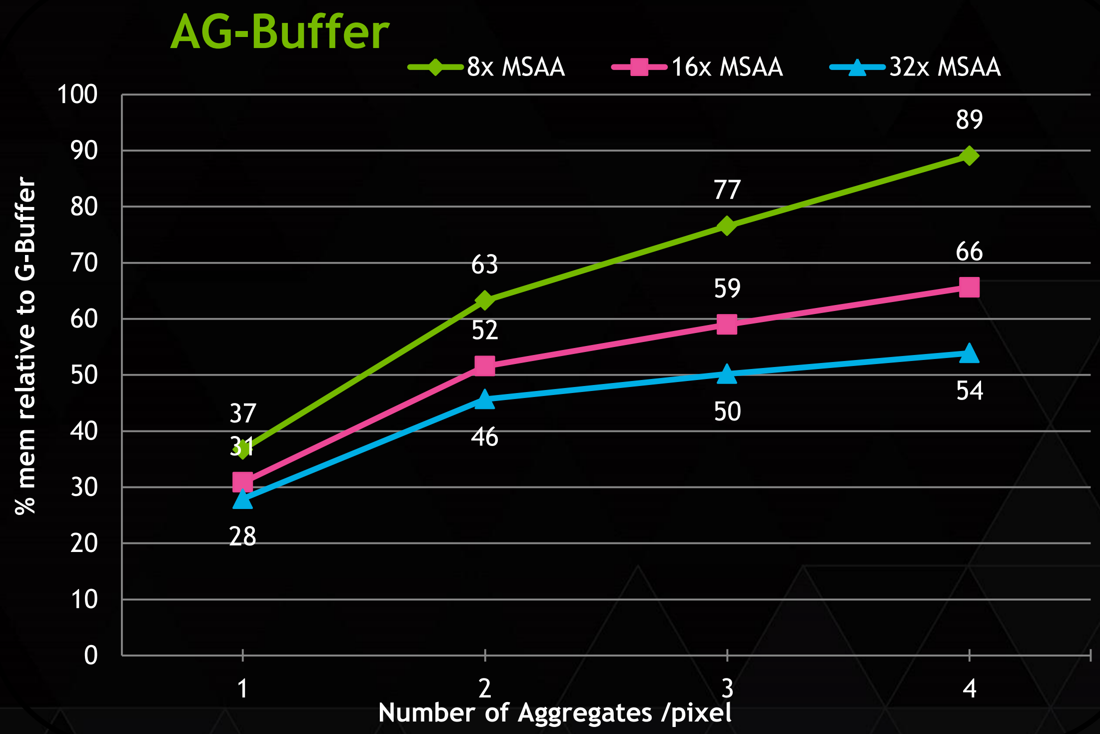
EPIC UE3 Foliage Map

74% Faster rendering than **Simple/Complex**

(2.85x Faster shading)

RESULTS: MEMORY

- Compared with super sampled G-buffer
  - Requires significantly less memory (**37% less** with 2 aggregates v.s. 8x MSAA)



LIMITATIONS

- Assumes all materials use same model
  - Not explored switching materials sub-pixel
  - All shader inputs assumed filterable
- Normal precision issues using few aggregates
  - Pixels with many prims & very different normal
  - Use a single lobe Gaussian distribution
    - Can cause some specular sparkling
- Correlation issues:
  - Lit green foliage over shadowed red wall

Both with 1 aggregate/ pixel

GPU TECHNOLOGY CONFERENCE

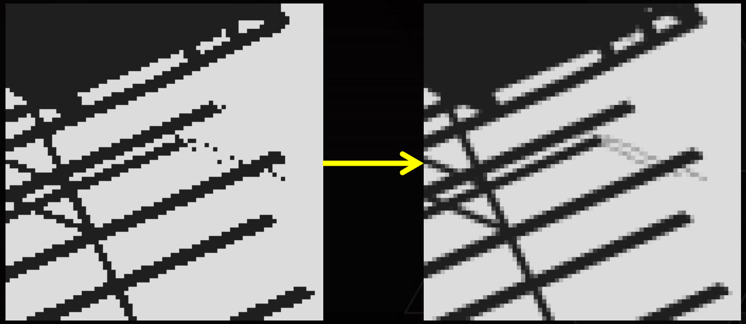
Frustum-Traced Irregular Z-Buffer Shadows

Work by:

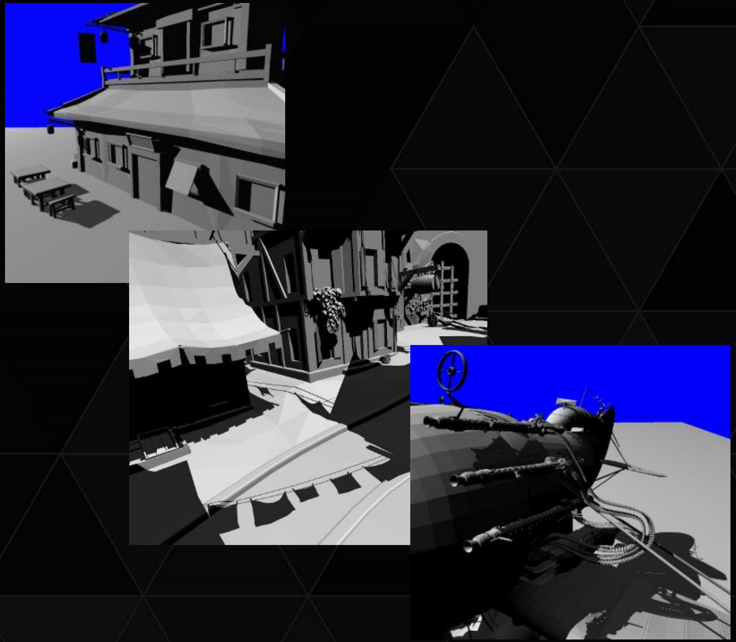
Chris Wyman, Rama Hoetzlein, and Aaron Lefohn

FEATURES

- Full scene, fully dynamic alias-free hard shadows
  - Show 32 spp shadows are under 2x cost of 1 spp shadows



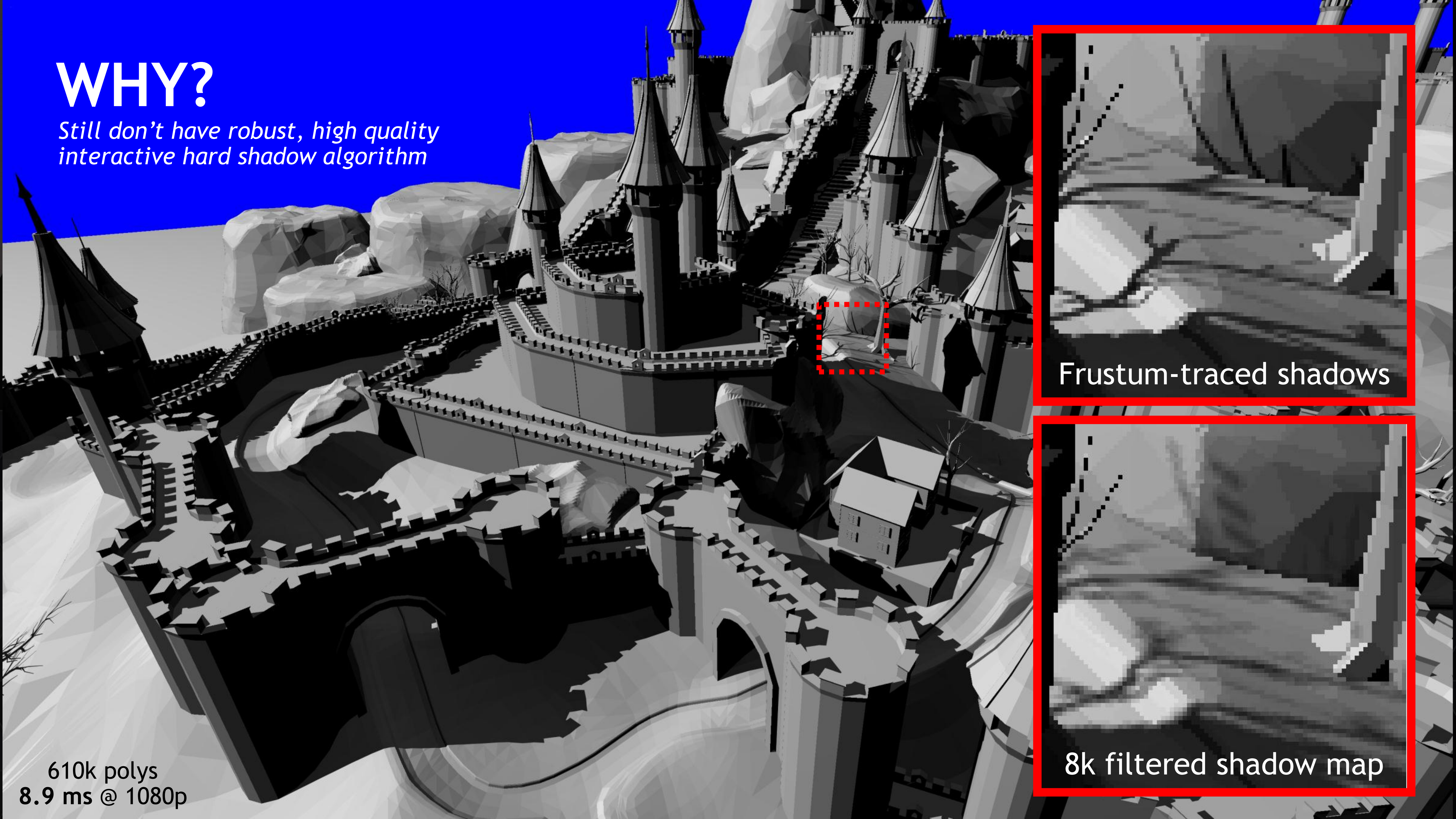
- Evolution of irregular z-buffering
  - For modern game-quality and CAD-quality assets
  - Builds on existing graphics hardware & pipeline
  - Demonstrate efficient frustum intersection for 32 spp



- Key takeaway:
  - Convert shadow map aliasing into irregular workload
  - Identify and remove perf bottlenecks from this workload

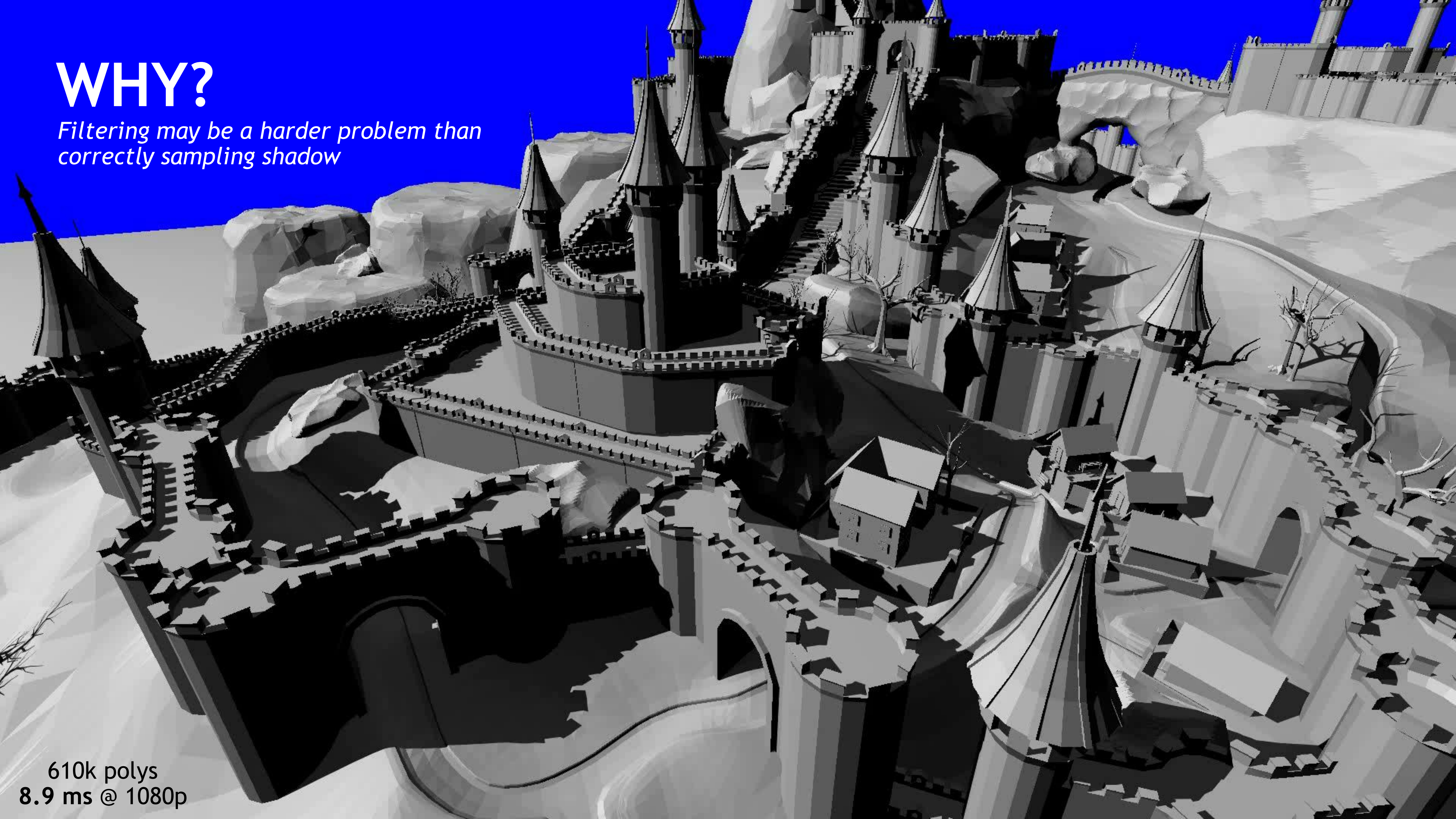
WHY?

Still don't have robust, high quality interactive hard shadow algorithm



WHY?

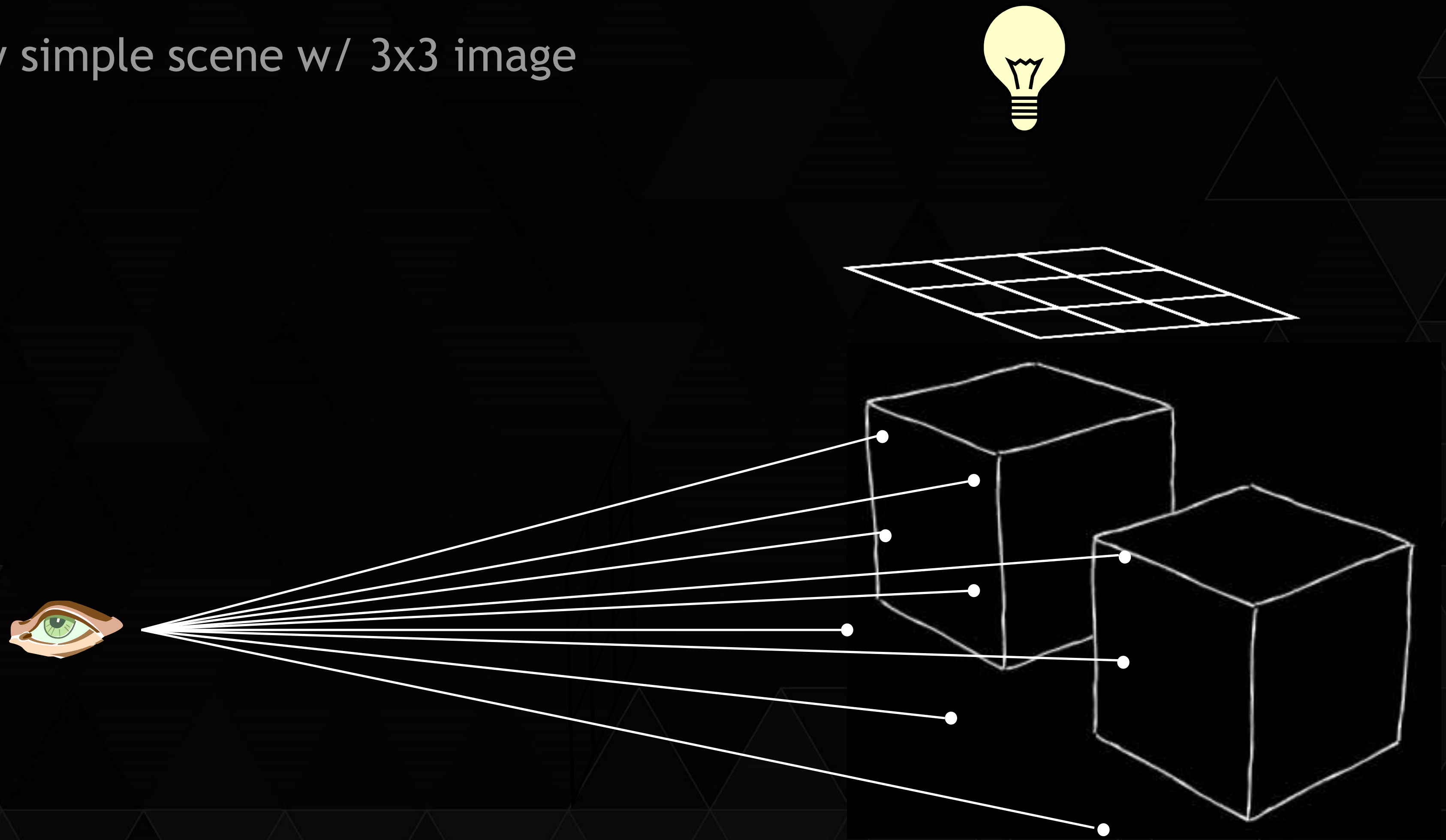
Filtering may be a harder problem than correctly sampling shadow



610k polys
8.9 ms @ 1080p

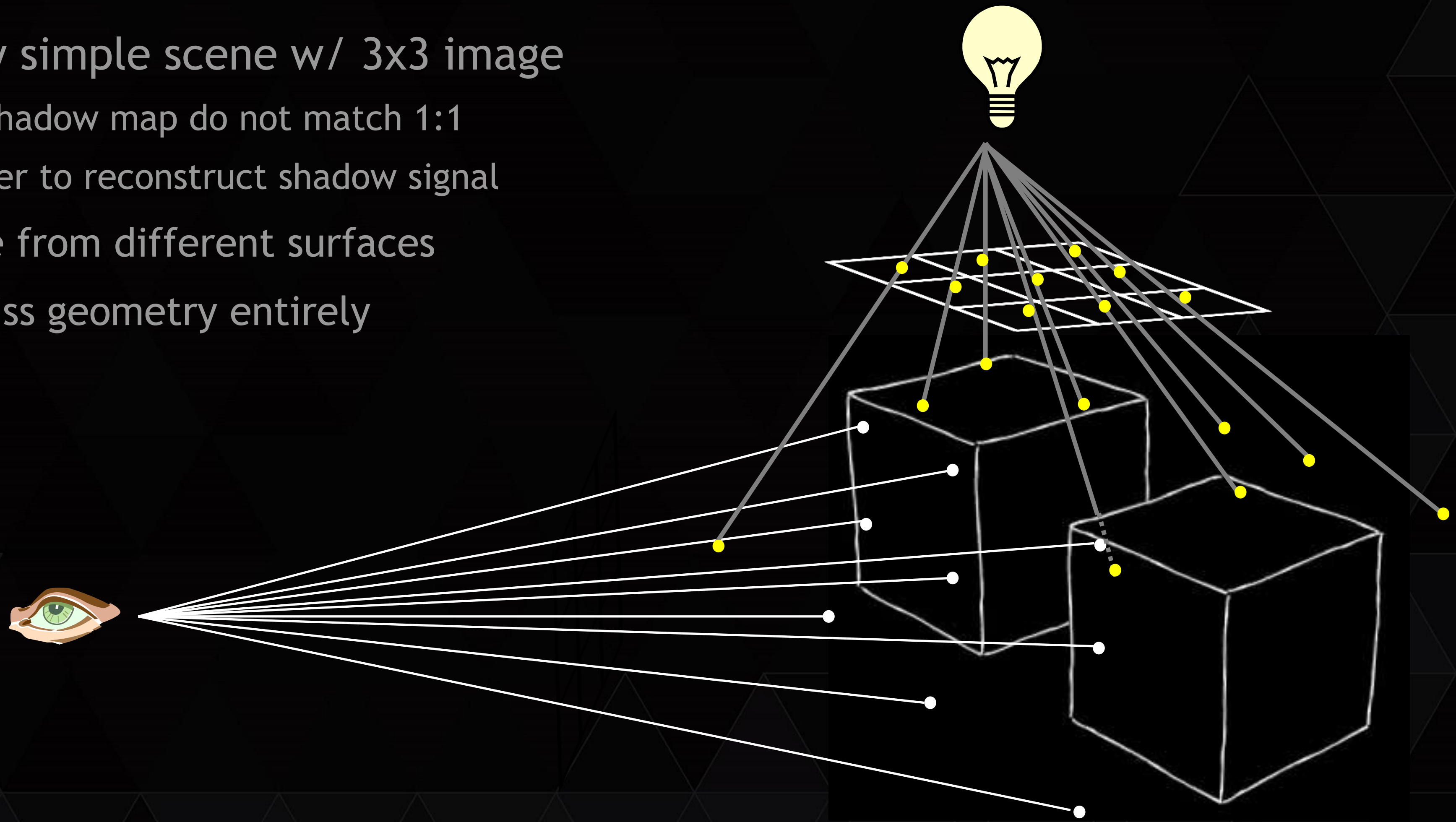
WHAT'S WRONG WITH EXISTING SHADOWS?

- Consider a very simple scene w/ 3x3 image



WHAT'S WRONG WITH EXISTING SHADOWS?

- Consider a very simple scene w/ 3x3 image
  - Samples in shadow map do not match 1:1
  - Requires filter to reconstruct shadow signal
    - May be from different surfaces
    - Can miss geometry entirely



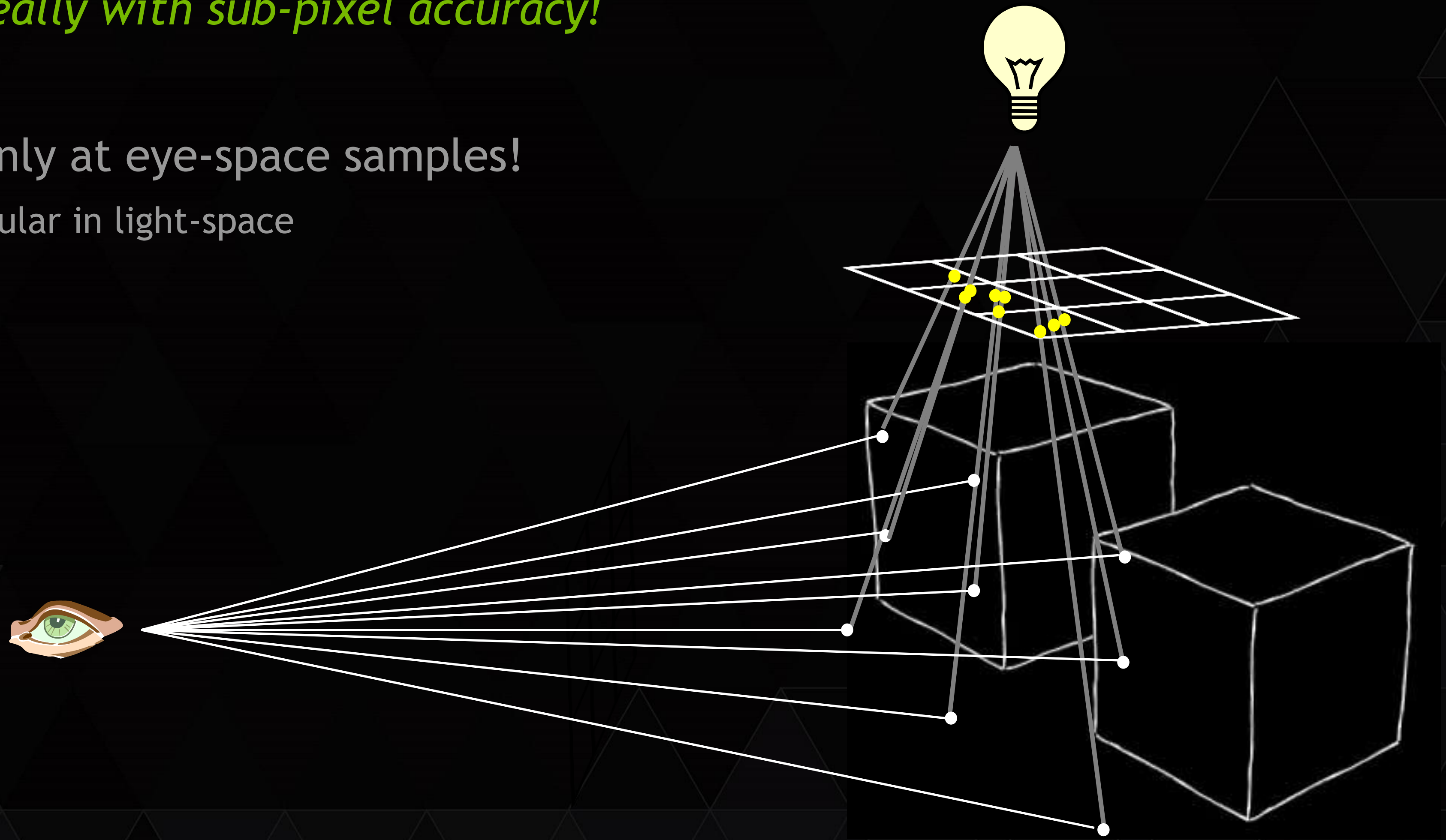
PRIOR WORK ON SHADOW MAPS

- Does one of two things:
  - Filter better (*e.g.*, *[Peters15] [Donnelly06] [Fernando05]*
    - Filtering is very hard; we still have problem antialiasing other signals
  - Better match eye & light-space samples (*e.g.*, *[Fernando01] [Stamminger02] [Lloyd08]*)
    - Perfect match impossible if requiring regular sampling in both eye & light space

THE GOAL: ALIAS-FREE SHADOWS

Ideally with sub-pixel accuracy!

- Want to light only at eye-space samples!
  - Will be irregular in light-space



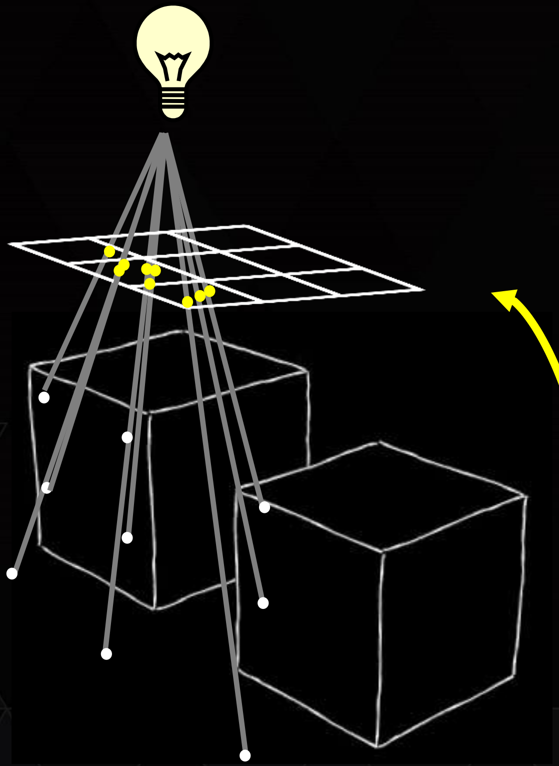
HOW TO DO THIS?

- Test triangle occlusion at these irregular sample points
  - Ray trace *(e.g., [Whitted80], [Parker10], [Mittring14])*
    - Query visibility at each ray, march through acceleration structure
  - Shadow volumes *(e.g., [Crow77], [Sintorn14], [Gerhards15])*
    - Test shadow quads to query if samples are in shadow
  - Irregular z-buffer *(e.g., [Johnson05], [Sintorn08], [Pan09])*
    - Rasterize over irregular sample points

- Converged on irregular z-buffering
  - Why? Allows us to leverage aspects of graphics pipe (e.g., culling)

WHAT IS AN IRREGULAR Z-BUFFER?

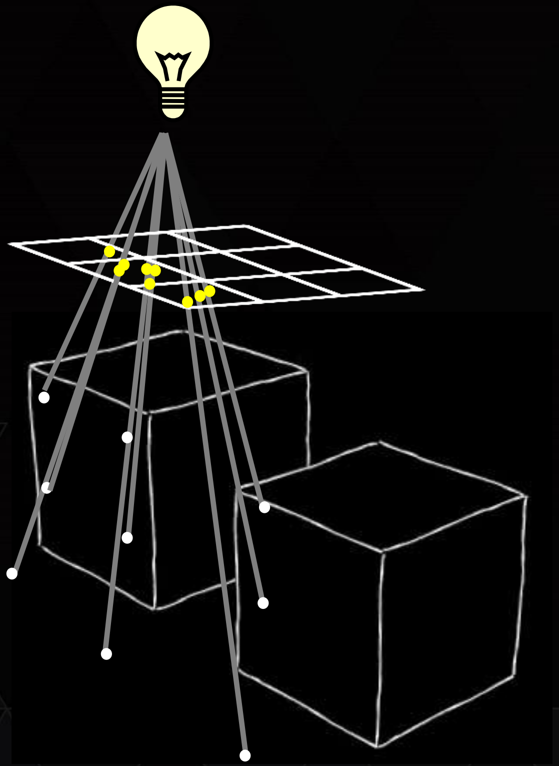
- Insert pixel samples (white dots) into light space grid at yellow samples

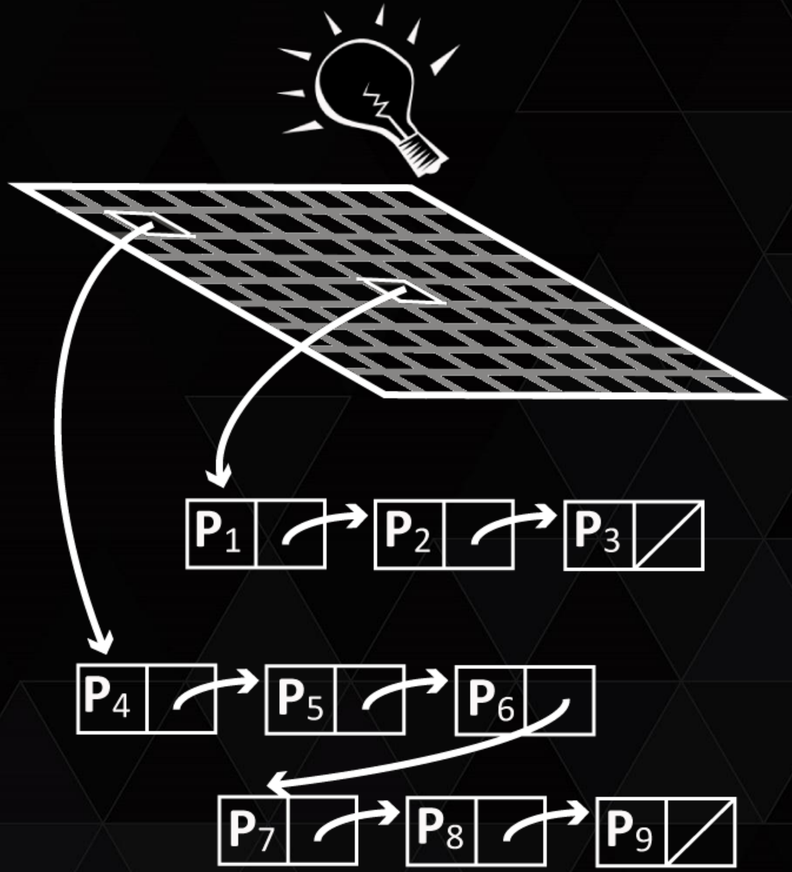


A standard shadow map projection matrix

WHAT IS AN IRREGULAR Z-BUFFER?

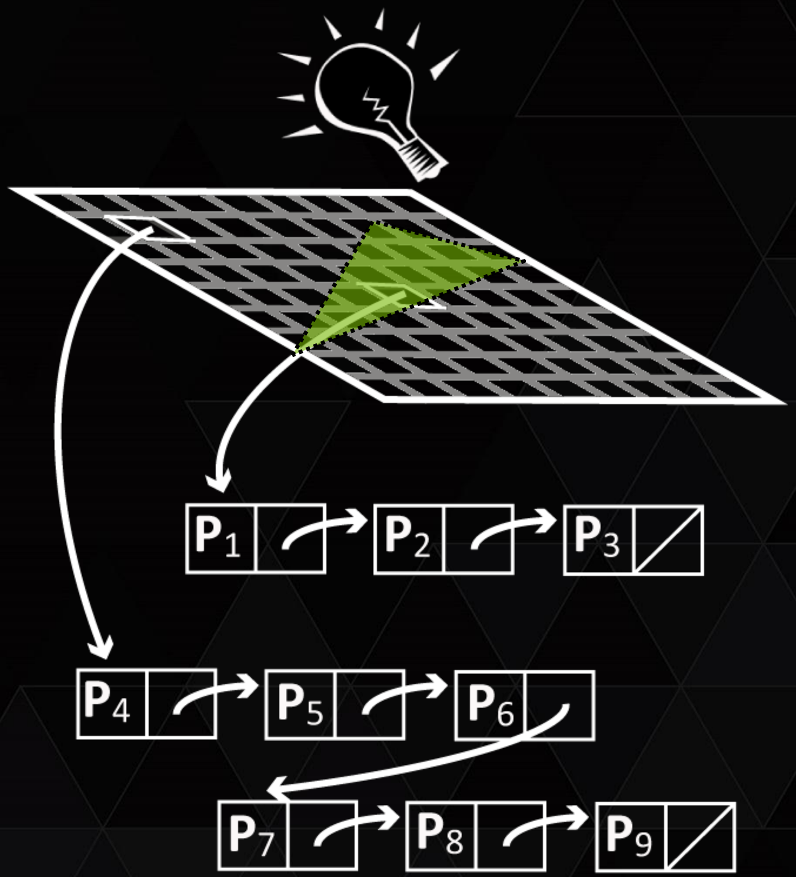
- Insert pixel samples (white dots) into light space grid at yellow samples
  - Creates grid-of-lists data structure





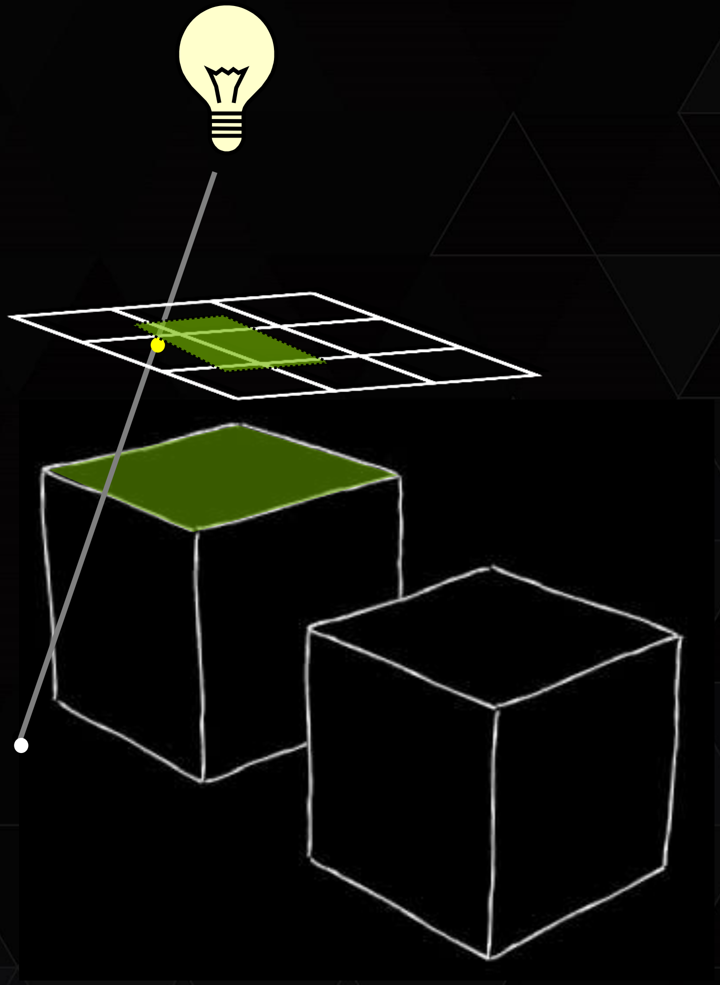
HOW DO YOU USE AN IZB?

- Rasterize from light view
  - For each texel (partially) covered
    - Walk through list of eye-space pixels P_i
    - Test ray from P_i to the light
    - Update visibility at P_i
- Store visibility for pixels P_i in eye-space buffer



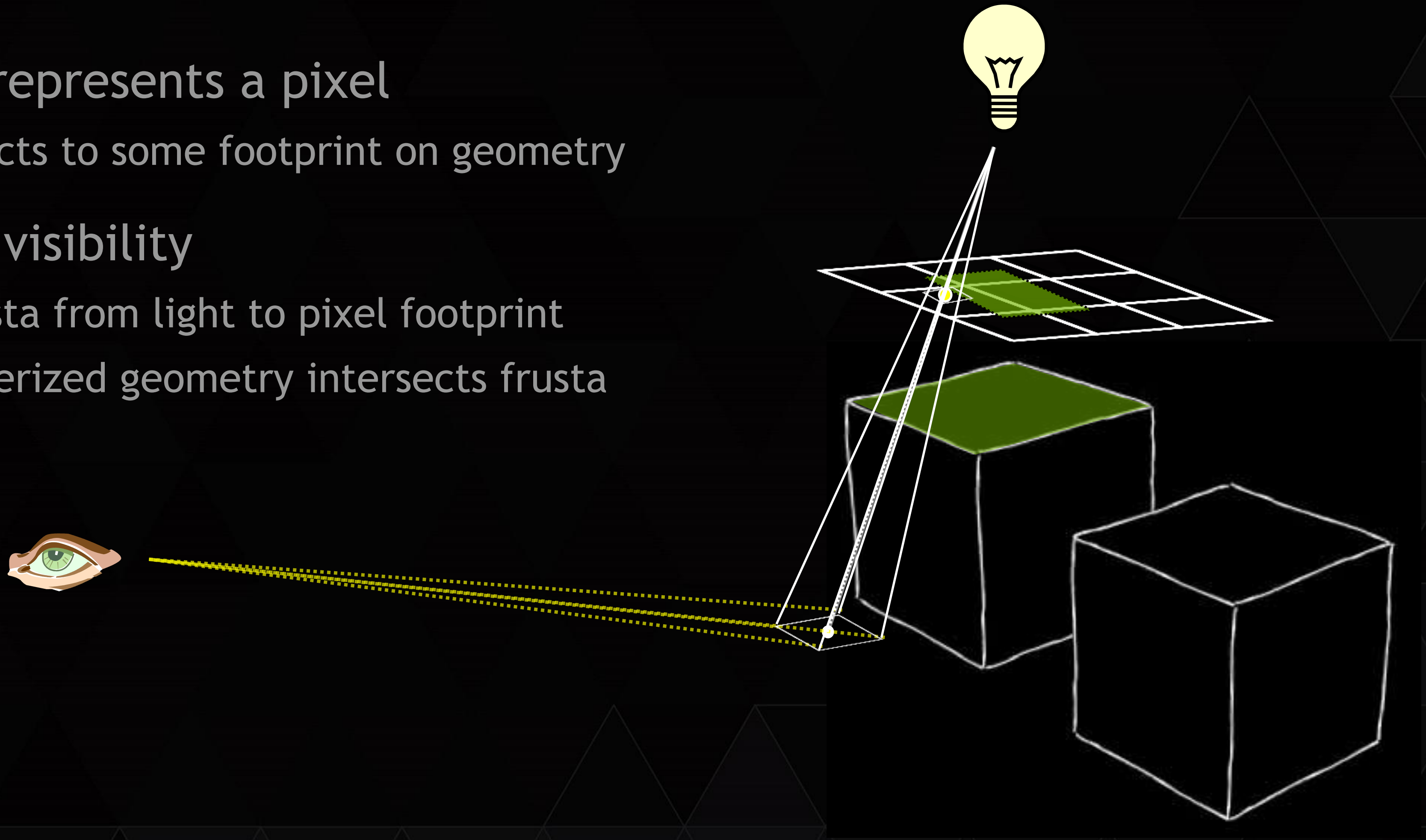
HOW DO YOU USE AN IZB?

- In simple cube example
  - When rendering top of box to light space
  - Partially covers texel containing a sample
  - Analytically test visibility for list of samples
  - This sample ends up unshadowed



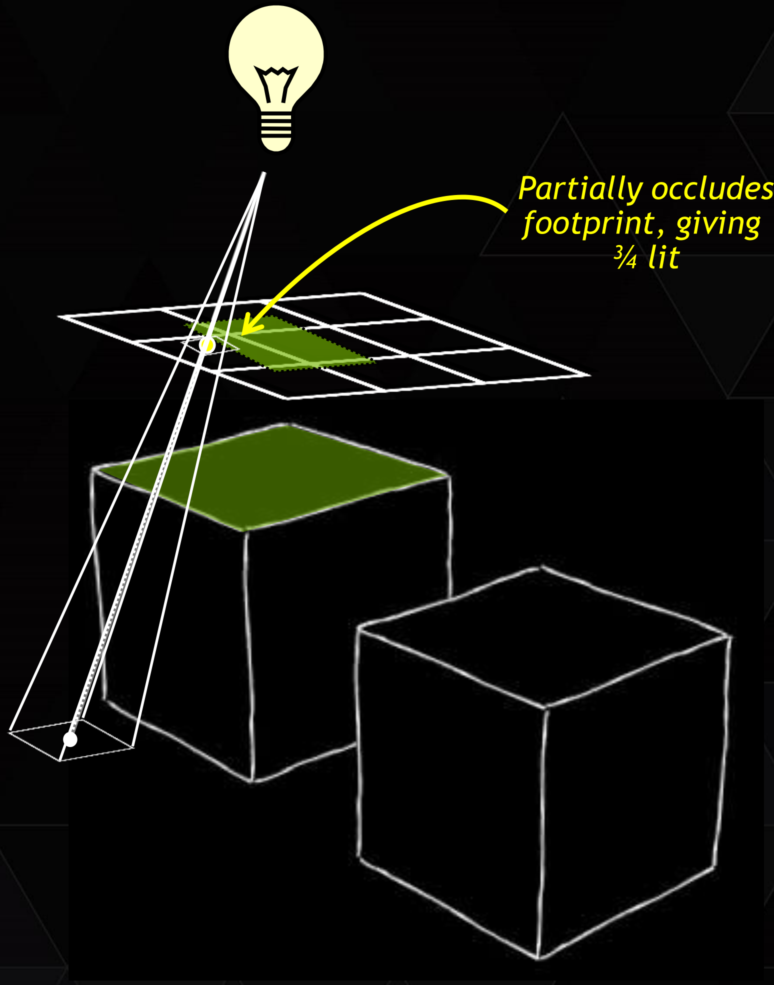
ADDING MULTIPLE SAMPLES PER PIXEL

- Each sample represents a pixel
  - Pixel projects to some footprint on geometry
- When testing visibility
  - Create frusta from light to pixel footprint
  - Test if rasterized geometry intersects frusta



ADDING MULTIPLE SAMPLES PER PIXEL

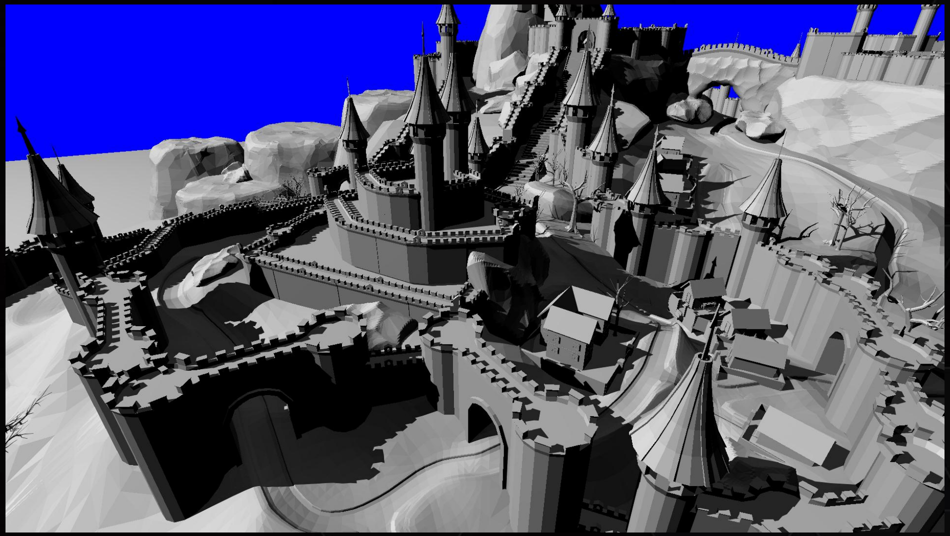
- Each sample represents a pixel
  - Pixel projects to some footprint on geometry
- When testing visibility
  - Create frusta from light to pixel footprint
  - Test if rasterized geometry intersects frusta
- Discretize visibility sampling on quad
  - Prototype uses 32 samples
  - Developer specified (currently a lookup table)
  - Each sample stores binary visibility

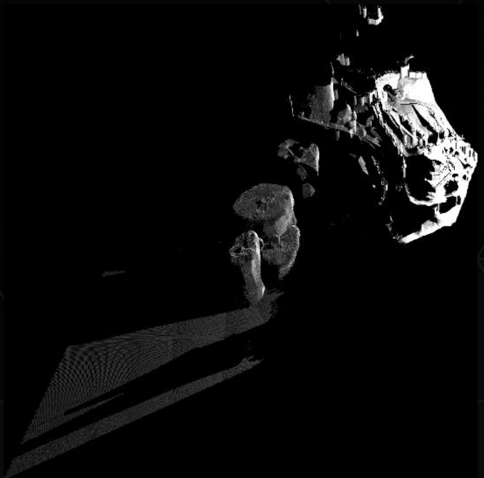


Problem with Irregular Z-Buffering

IRREGULARITY: BAD FOR GPU UTILIZATION

- By construction:
  - Introduce irregular workloads
  - As variable-length light-space lists
- When rasterizing in light space
  - Some frags test visibility of no pixels
  - Some frags test at 1000’s of pixels
- Naïve implementation
  - Leads to 100:1 variation in frame time





Light-space visualization

Intensity represents number of list elements per light space texel

IZB Complexity Considerations

WHAT WORK ACTUALLY OCCURS?

- Complexity is simple: $O(N)$
  - N = # of frusta-triangle visibility tests
- More usefully, complexity is: $O(f_{ls} * L_{avg})$
  - f_{ls} = # of light-space fragments from rasterizer
  - L_{avg} = average list length (i.e., # of pixels tested)
- For poorly utilized GPU, complexity is roughly: $O(f_{ls} * L_{max})$
  - L_{max} = # of pixels tested by slowest thread

HOW TO REDUCE COST?

- Reduce the number of fragments, f_{ls}.
- Reduce the list length, L_{avg}.
- Reduce the variance, to reduce gap between L_{max} and L_{avg}.

HOW TO REDUCE COST?

- Reduce the number of fragments, f_{ls}.
  - Reduce *number* of occluder triangles
    - Front/back face culling, z-culling, frustum culling, artistic culling

HOW TO REDUCE COST?

- Reduce the number of fragments, f_{ls}.
  - Reduce *number* of occluder triangles
    - Front/back face culling, z-culling, frustum culling, artistic culling
  - Reduce rasterized *size* of occluder triangles (i.e., change grid size)
    - But this increases L_{avg}, L_{max}, and other overheads; find the broad sweet spot per scene.

GPU TECHNOLOGY CONFERENCE

HOW TO REDUCE COST?

- Reduce the number of fragments, f_{ls}.
  - Reduce *number* of occluder triangles
  - Reduce rasterized *size* of occluder triangles (i.e., change grid size)
- Reduce the list length, L_{avg}.
- Reduce the variance, to reduce gap between L_{max} and L_{avg}.

HOW TO REDUCE COST?

- Reduce the number of fragments, f_{ls}.
  - Reduce *number* of occluder triangles
  - Reduce rasterized *size* of occluder triangles (i.e., change grid size)
- Reduce the list length, L_{avg}.
  - Reduce # of pixels *inserted* into IZB
    - Z-prepass, skip N•L < 0, skip known lit pixels, avoid duplicates, use approx IZB insertion

HOW TO REDUCE COST?

- Reduce the number of fragments, f_{ls}.
  - Reduce *number* of occluder triangles
  - Reduce rasterized *size* of occluder triangles (i.e., change grid size)
- Reduce the list length, L_{avg}.
  - Reduce # of pixels *inserted* into IZB
    - Z-prepass, skip N•L < 0, skip known lit pixels, avoid duplicates, use approx IZB insertion
  - *Remove* fully shadowed pixels from IZB
    - Gradually reduces L_{avg} and L_{max} over the frame

HOW TO REDUCE COST?

- Reduce the number of fragments, f_{ls}.
  - Reduce *number* of occluder triangles
  - Reduce rasterized *size* of occluder triangles (i.e., change grid size)
- Reduce the list length, L_{avg}.
  - Reduce # of pixels *inserted* into IZB
  - *Remove* fully shadowed pixels from IZB
- Reduce the variance, to reduce gap between L_{max} and L_{avg}.

HOW TO REDUCE COST?

- Reduce the number of fragments, f_{ls}.
  - Reduce *number* of occluder triangles
  - Reduce rasterized *size* of occluder triangles (i.e., change grid size)
- Reduce the list length, L_{avg}.
  - Reduce # of pixels *inserted* into IZB
  - *Remove* fully shadowed pixels from IZB
- Reduce the variance, to reduce gap between L_{max} and L_{avg}.
  - Ideally: match samples 1:1 between eye- & light-space
  - The *key goal* for fast GPU implementation

GPU TECHNOLOGY CONFERENCE

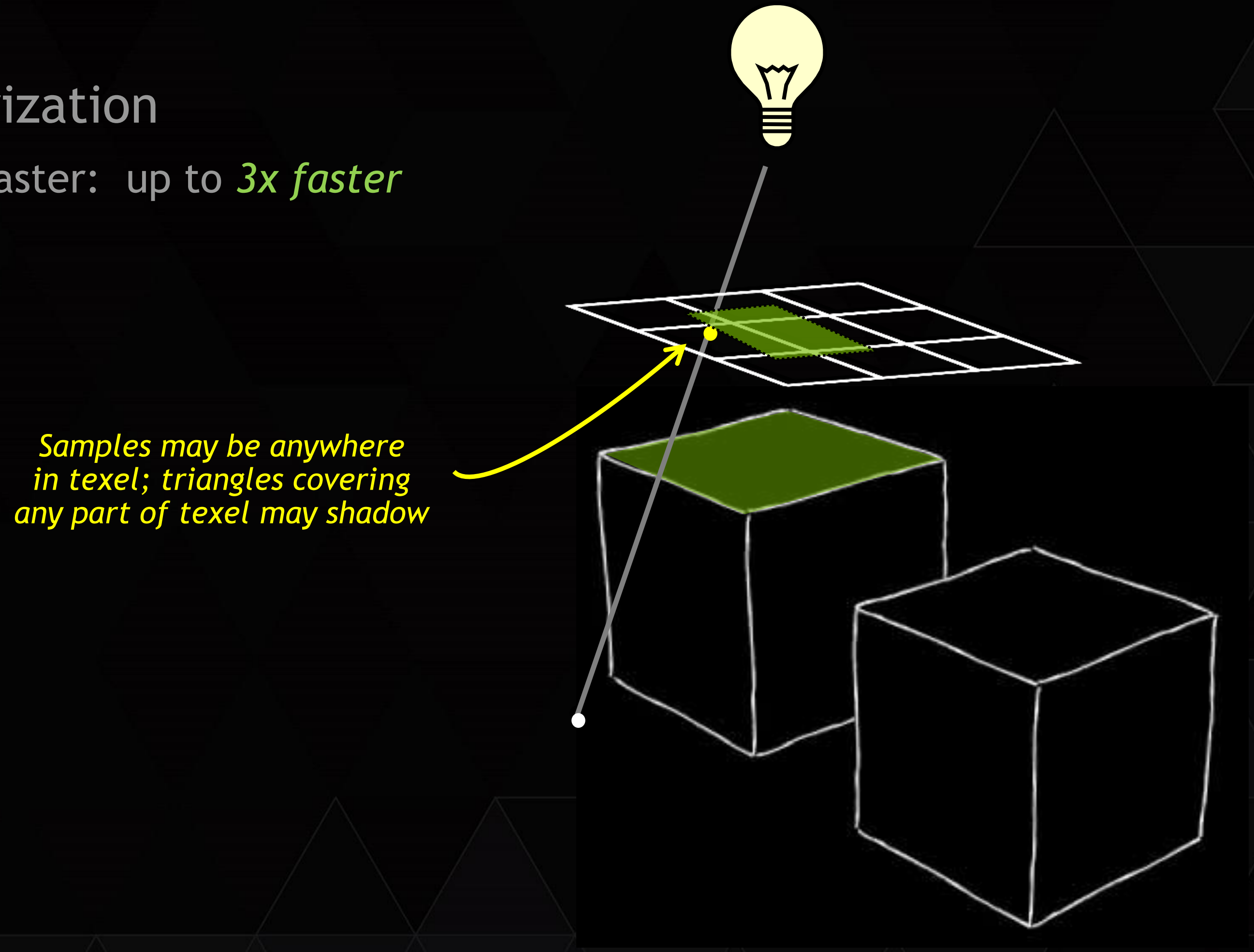
HOW TO REDUCE COST?

- Reduce the number of fragments, f_{ls}.
  - Reduce *number* of occluder triangles
  - Reduce rasterized *size* of occluder triangles (i.e., change grid size)
- Reduce the list length, L_{avg}.
  - Reduce # of pixels *inserted* into IZB
  - *Remove* fully shadowed pixels from IZB
- Reduce the variance, to reduce gap between L_{max} and L_{avg}.
  - Ideally: match samples 1:1 between eye- & light-space
  - The *key goal* for fast GPU implementation
  - We use *cascaded irregular z-buffers*

Miscellaneous Optimizations

GENERAL GPU OPTIMIZATIONS

- IZBs require conservative rasterization
  - Maxwell hardware conservative raster: up to *3x faster*



GENERAL GPU OPTIMIZATIONS

- IZBs require conservative rasterization
  - Maxwell hardware conservative raster: up to *3x faster*
- Memory contention / atomics are slower
  - Only update visibility mask *if change occurs*
  - Use *implicit indices*; skip global memory pools
  - Structure traversal to *avoid atomics*

GENERAL GPU OPTIMIZATIONS

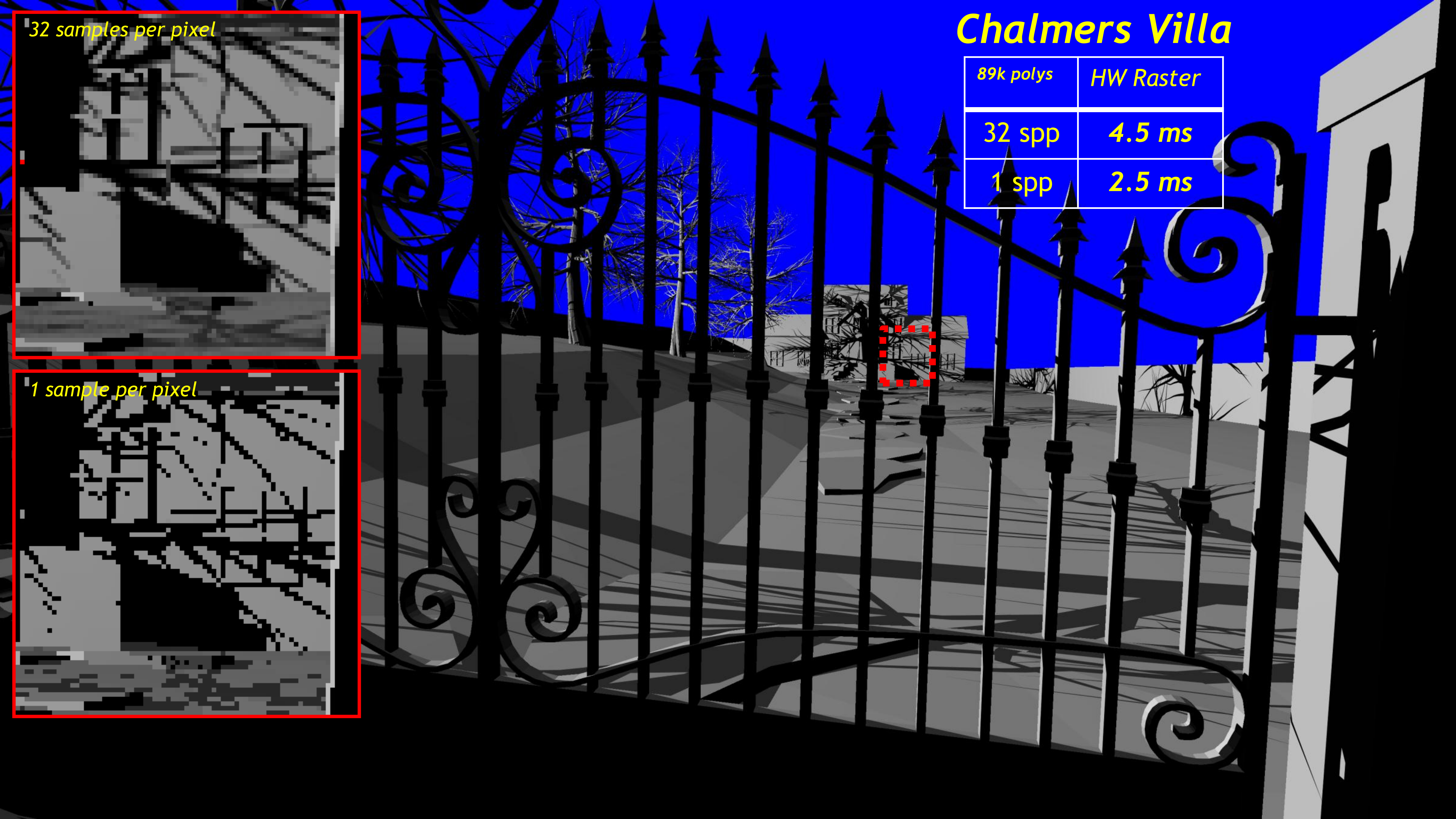
- IZBs require conservative rasterization
  - Maxwell hardware conservative raster: up to *3x faster*
- Memory contention / atomics are slower
  - Only update visibility mask *if change occurs*
  - Use *implicit indices*; skip global memory pools
  - Structure traversal to *avoid atomics*
- List traversal induces long dependency chains
  - Hide latency via *software pipelining*
  - *Avoid long latency* operations (e.g., int divide, modulo)
- Reduce SIMD divergence
  - *Flatten control flow* as much as possible

Results

(All at 1080p on a GeForce GTX 980)

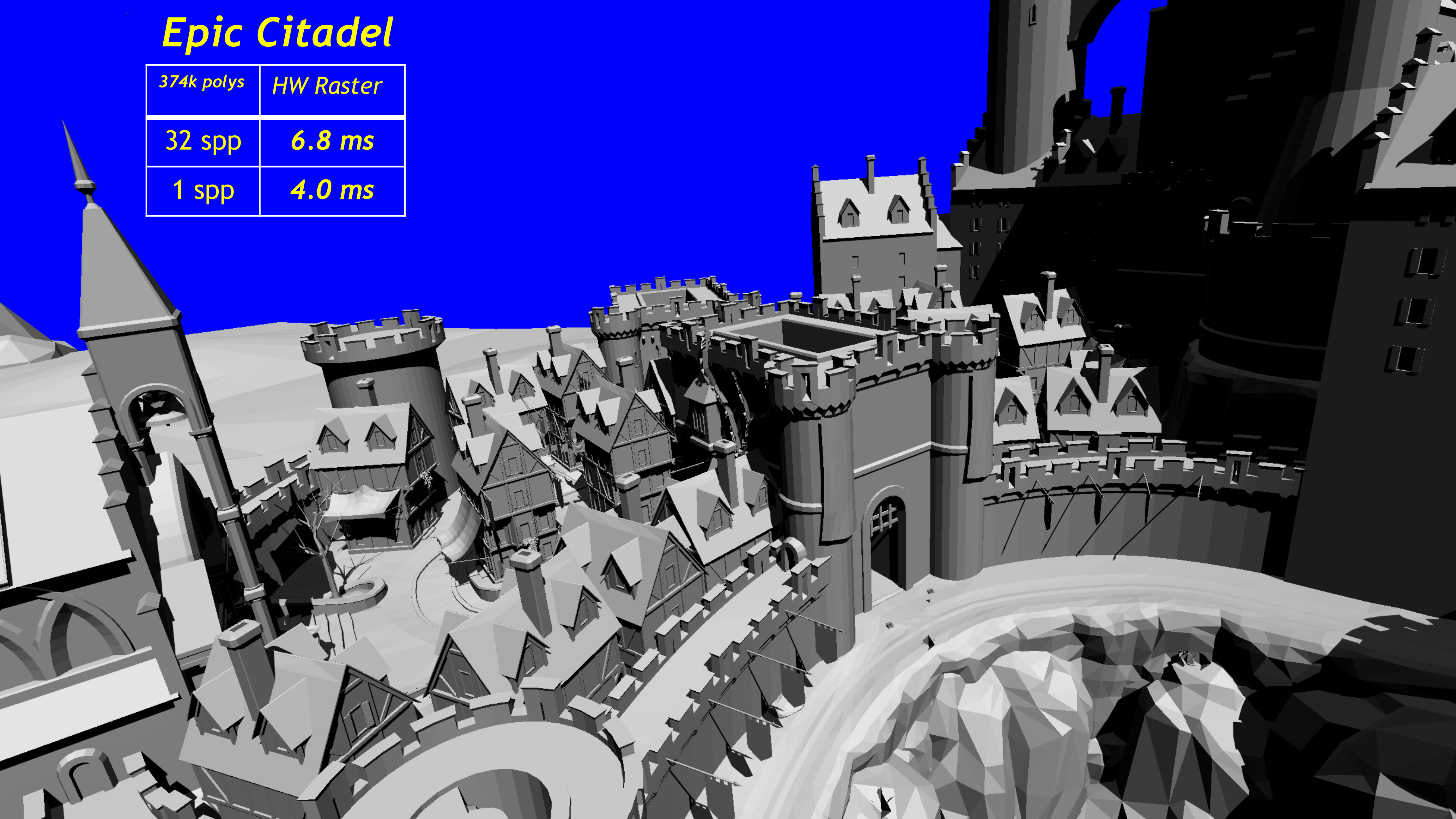
Chalmers Villa

89k polys	HW Raster
32 spp	4.5 ms
1 spp	2.5 ms



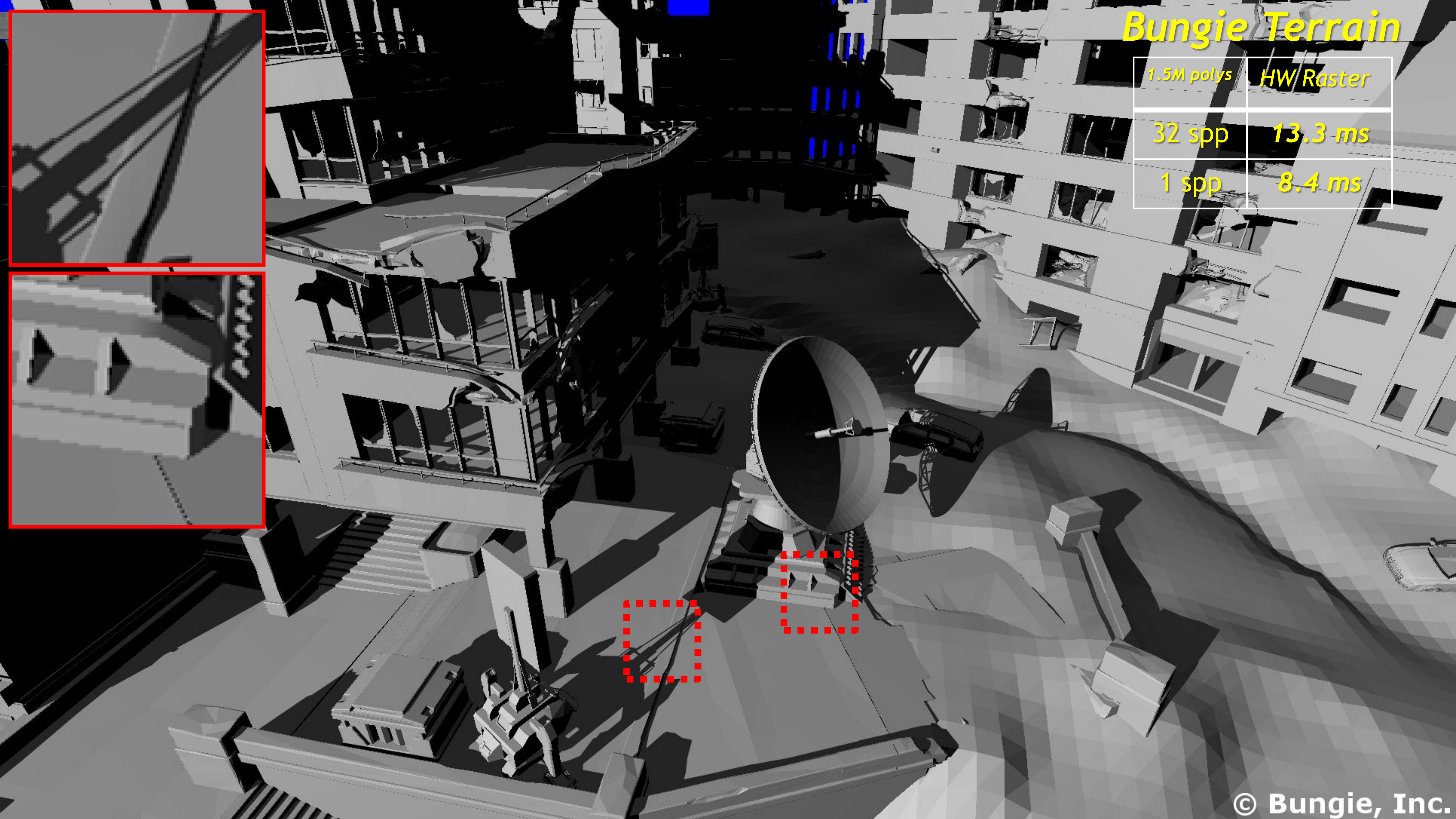
Epic Citadel

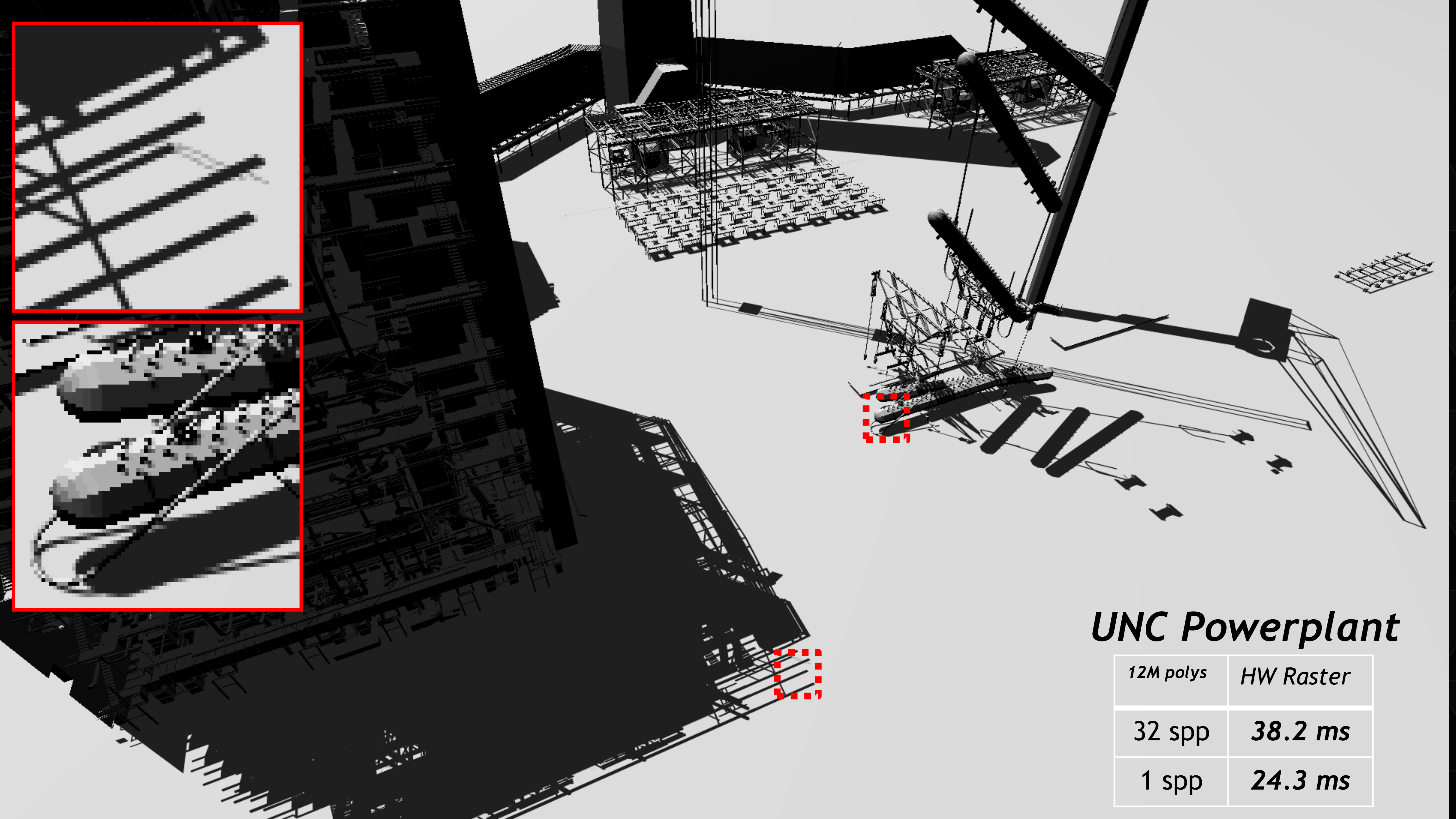
374k polys	HW Raster
32 spp	6.8 ms
1 spp	4.0 ms



Bungie Terrain

1.5M polys	HW Raster
32 spp	13.3 ms
1 spp	8.4 ms





UNC Powerplant

12M polys	HW Raster
32 spp	***38.2 ms***
1 spp	***24.3 ms***

UNC Powerplant

12M polys	HW Raster
32 spp	***38.2 ms***
1 spp	***24.3 ms***

FTIZB LIMITATIONS

- Requires an epsilon
  - In world space, to avoid self shadows; roughly same as ray tracing
- Performance still variable (around 2x)
  - We’re still working on this
- One approx used for performance for 32 spp shadows can break
  - If using non-LoD, highly tessellated models in distance (i.e., not closest cascade)
- Some sub-pixel robustness tricks needed for 32 spp

TALK SUMMARY

CONCLUSION

- Presented 3 new high quality raster algorithms:
  - ACAA improves MSAA for forward renderers
  - AGAA reduces costs for higher sampling rates in a deferred renderer
  - FTIZB renders smoothly anti-aliased hard shadows, avoiding shadow map sampling problems
- Leverage new Maxwell GPU features
  - Post-z coverage, target independent raster, conservative raster, fast geometry shader
- These simple hardware changes open up many new and exciting algorithms!

GPU TECHNOLOGY CONFERENCE

THANK YOU

JOIN THE CONVERSATION

#GTC15

cwyman@nvidia.com

@_cwyman_

BACKUP SLIDES

JOIN THE CONVERSATION
#GTC15

REDUCING NUMBER OF FRAGMENTS

- Reduce *number* of occluder triangles
  - Front/back face culling (we do this)
  - Z-culling (we do this, partially)
  - Frustum culling (we do not do this)
  - Artistic direction (we do not do this)

REDUCING NUMBER OF FRAGMENTS

- Reduce *number* of occluder triangles
  - Front/back face culling (we do this)
  - Z-culling (we do this, partially)
  - Frustum culling (we do <u>not</u> do this)
  - Artistic direction (we do <u>not</u> do this)

- Reduce rasterized *size* of occluder triangles (i.e., change grid size)
  - But this increases L_{avg}, L_{max}, and other overheads
  - A broad resolution "sweet spot" per scene for optimal

REDUCING LIST LENGTH L_{avg} AND L_{max}

- Reduce # of pixels *inserted* into IZB
  - Use z-prepass to insert only visible pixels (we do this)
  - Skip known shadowed pixels (N•L < 0) (we do this)
  - Skip known lit pixels (e.g., artistic direction) (we do <u>not</u> do this)
  - Avoid duplicates nodes (e.g., when using 32spp) (we do this)
  - For 32spp, use approximate insertion (we do this; see paper)

REDUCING LIST LENGTH L_{avg} AND L_{max}

- Reduce # of pixels *inserted* into IZB
  - Use z-prepass to insert only visible pixels (we do this)
  - Skip known shadowed pixels (N•L < 0) (we do this)
  - Skip known lit pixels (e.g., artistic direction) (we do not do this)
  - Avoid duplicates nodes (e.g., when using 32spp) (we do this)
  - For 32spp, use approximate insertion (we do this; see paper)

- *Remove* fully shadowed pixels from IZB
  - Gradually reduces L_{avg} and L_{max} over the frame (we do this)

REDUCING LIST LENGTH VARIANCE

- Causes $L_{max} \rightarrow L_{avg}$
- Ideally: match samples 1:1 between eye- & light-space
  - Same goal as perspective, logarithm, adaptive, and cascaded shadow maps
- The *key goal* for fast GPU implementation

REDUCING LIST LENGTH VARIANCE

- Causes $L_{max} \rightarrow L_{avg}$
- Ideally: match samples 1:1 between eye- & light-space
  - Same goal as perspective, logarithm, adaptive, and cascaded shadow maps
- The *key goal* for fast GPU implementation
  - Use these shadow map techniques (we use cascades)
  - Tightly bound light frustum to visible scene (we do this)