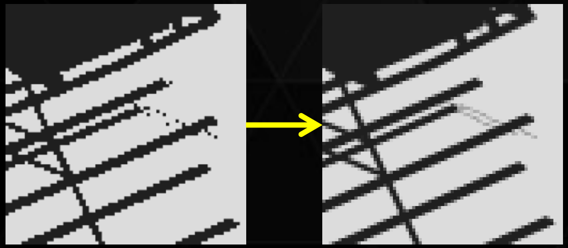
FRUSTUM-TRACED RASTER SHADOWS: REVISITING IRREGULAR Z-BUFFERS

Chris Wyman, Rama Hoetzlein, Aaron Lefohn

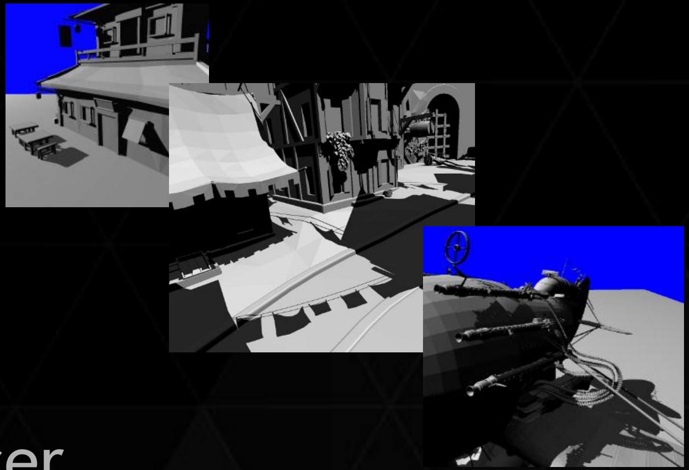
2015 Symposium on Interactive 3D Graphics & Games

CONTRIBUTIONS

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



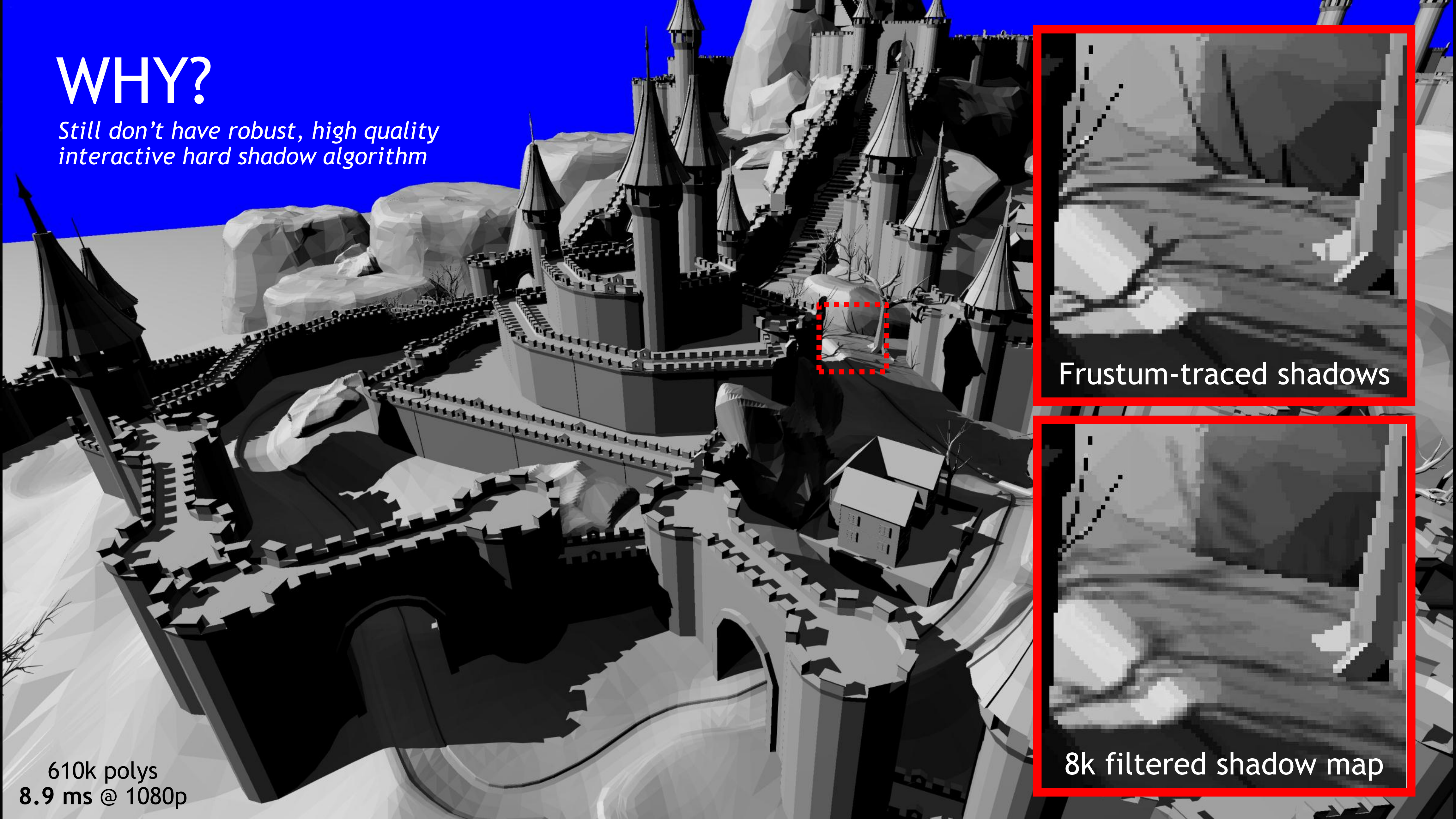
- Evolution of irregular z-buffering
  - For modern game-quality and CAD-quality assets
  - Builds on existing graphics hardware & pipeline
  - Demonstrate efficient frustum intersection for 32 spp



- # frustum-triangle tests competitive with ray tracer
  - We build our data structure in ~2 ms per frame

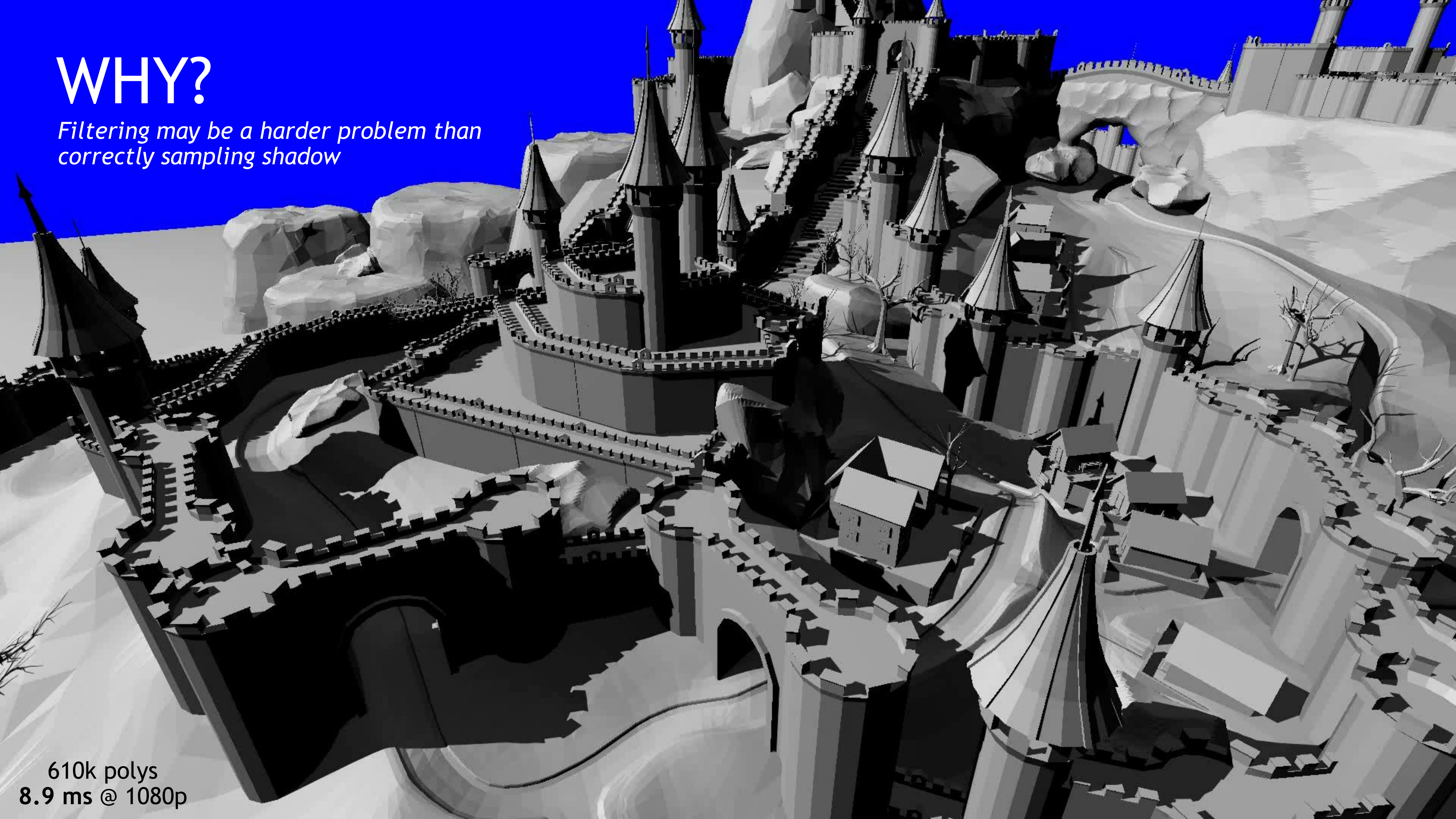
WHY?

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



WHY?

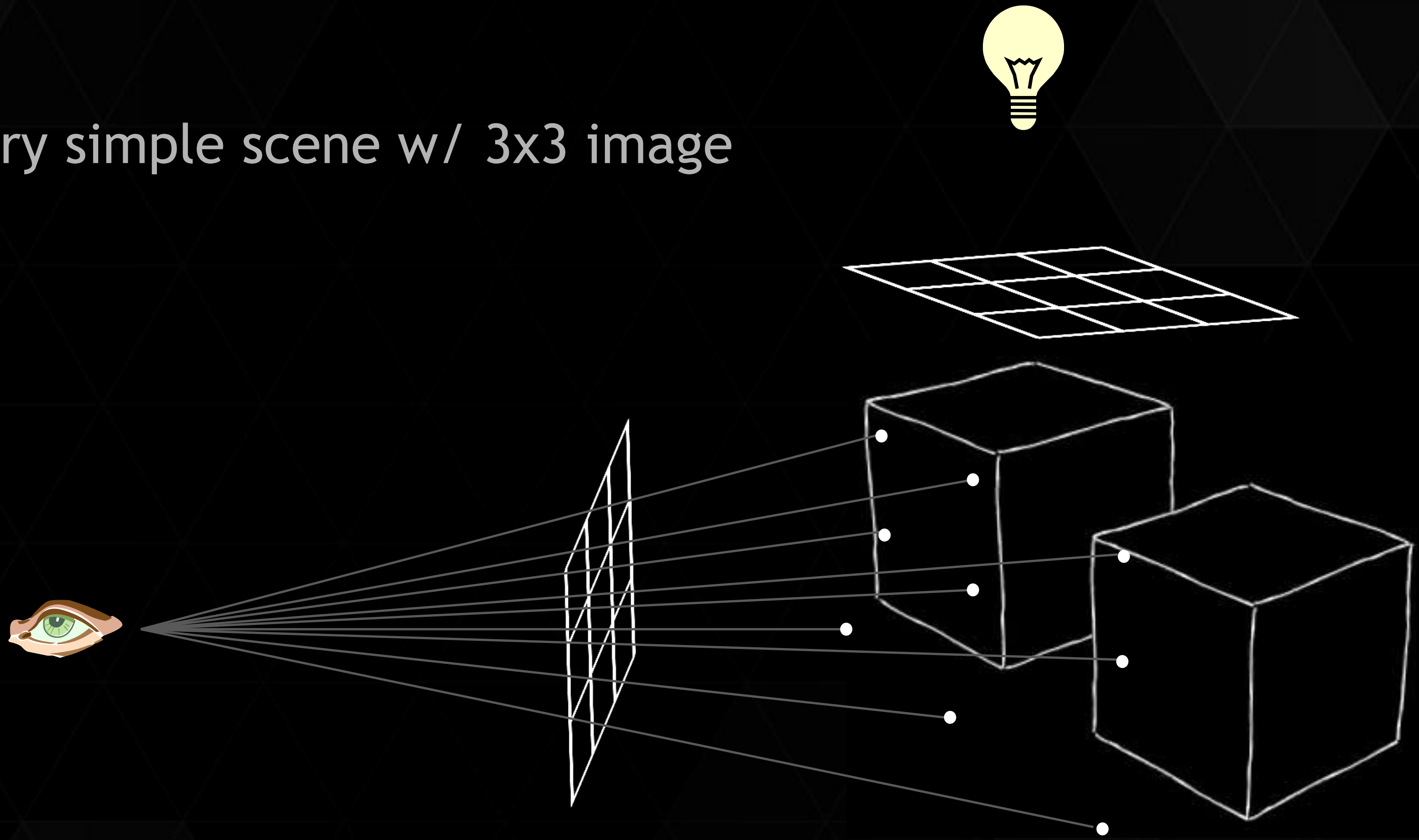
Filtering may be a harder problem than correctly sampling shadow



610k polys
8.9 ms @ 1080p

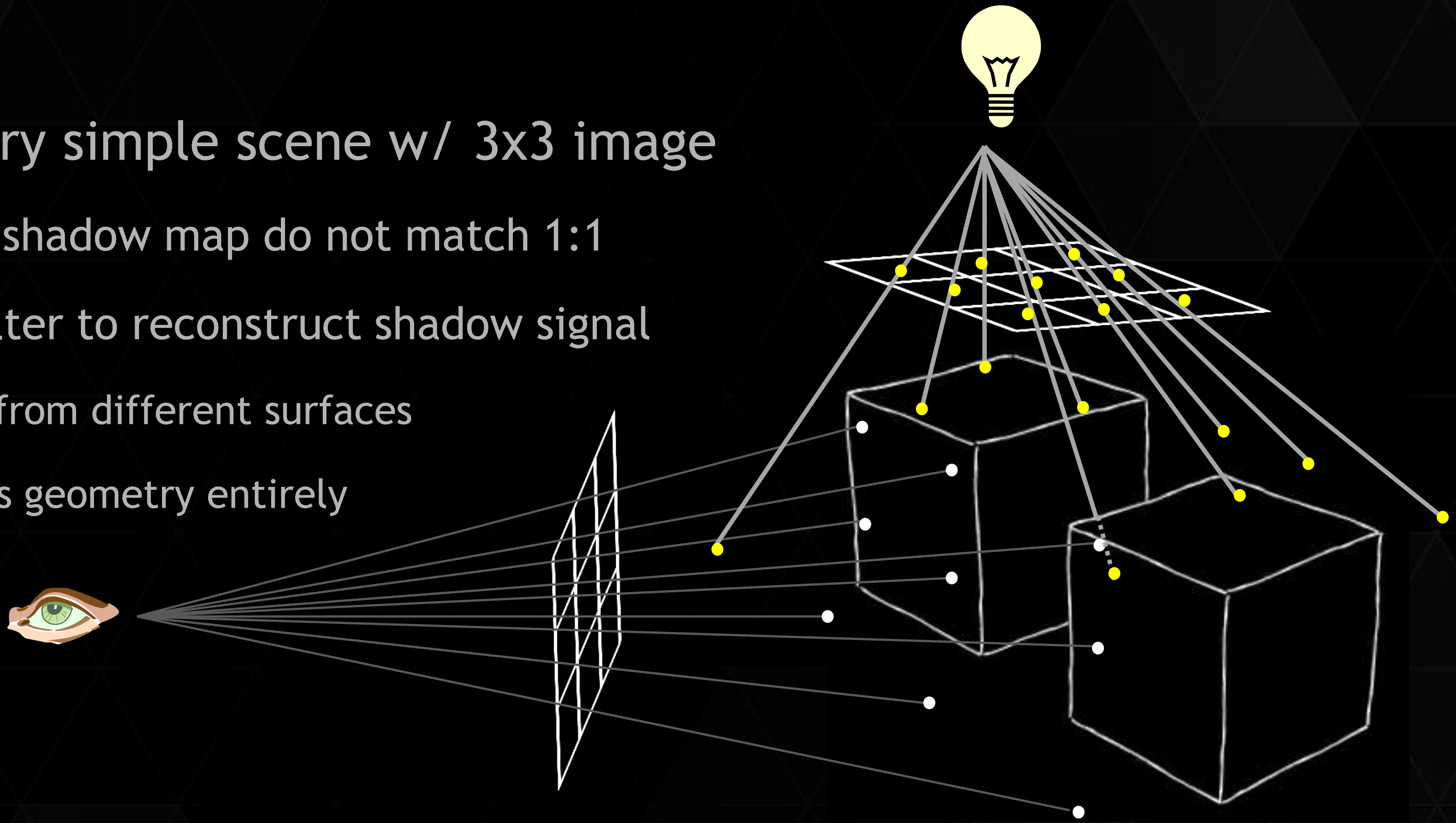
WHAT'S WRONG WITH EXISTING SHADOWS?

- 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



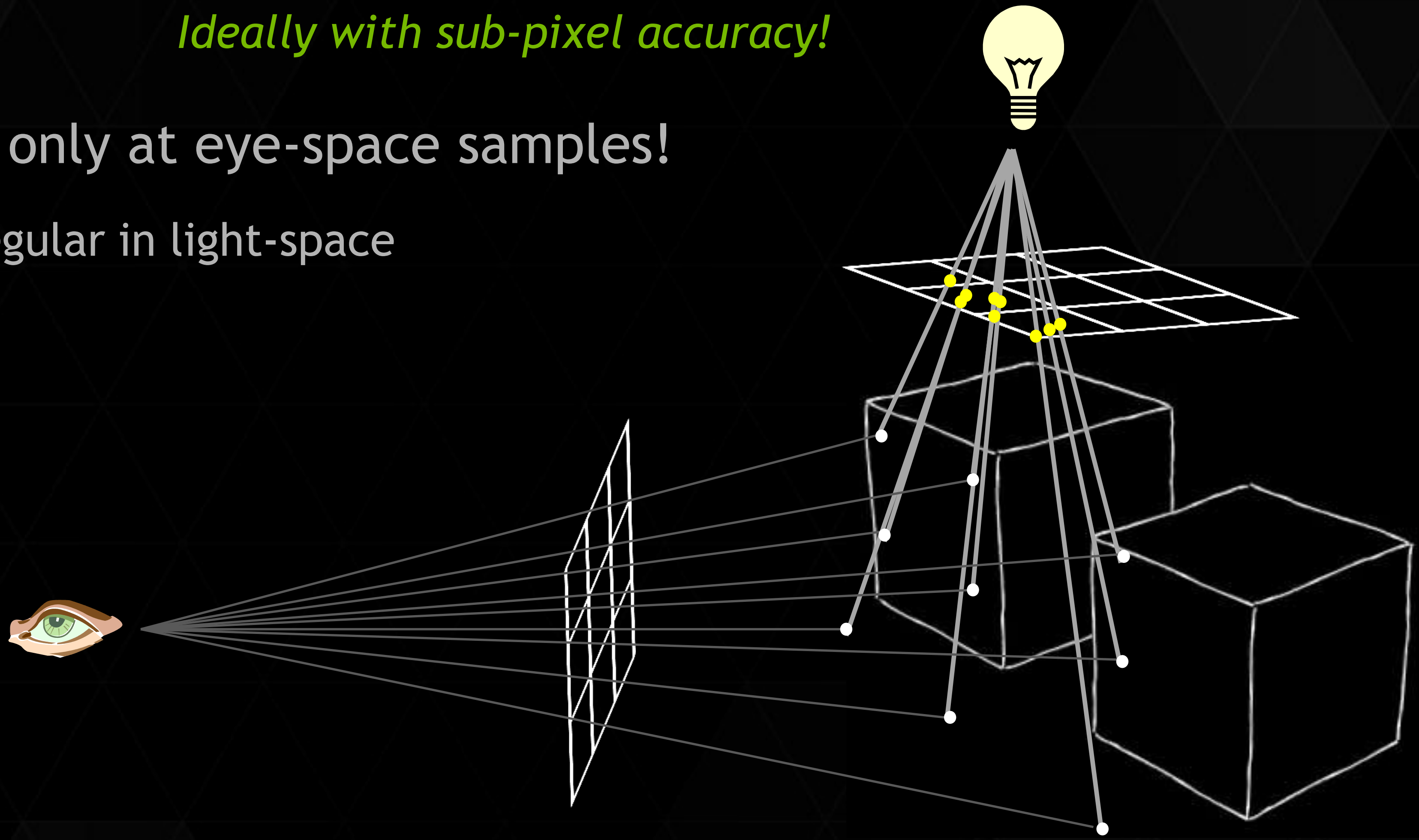
PRIOR WORK ON SHADOW MAPS

- 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

OUR GOAL: ALIAS-FREE SHADOWS

Ideally with sub-pixel accuracy!

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

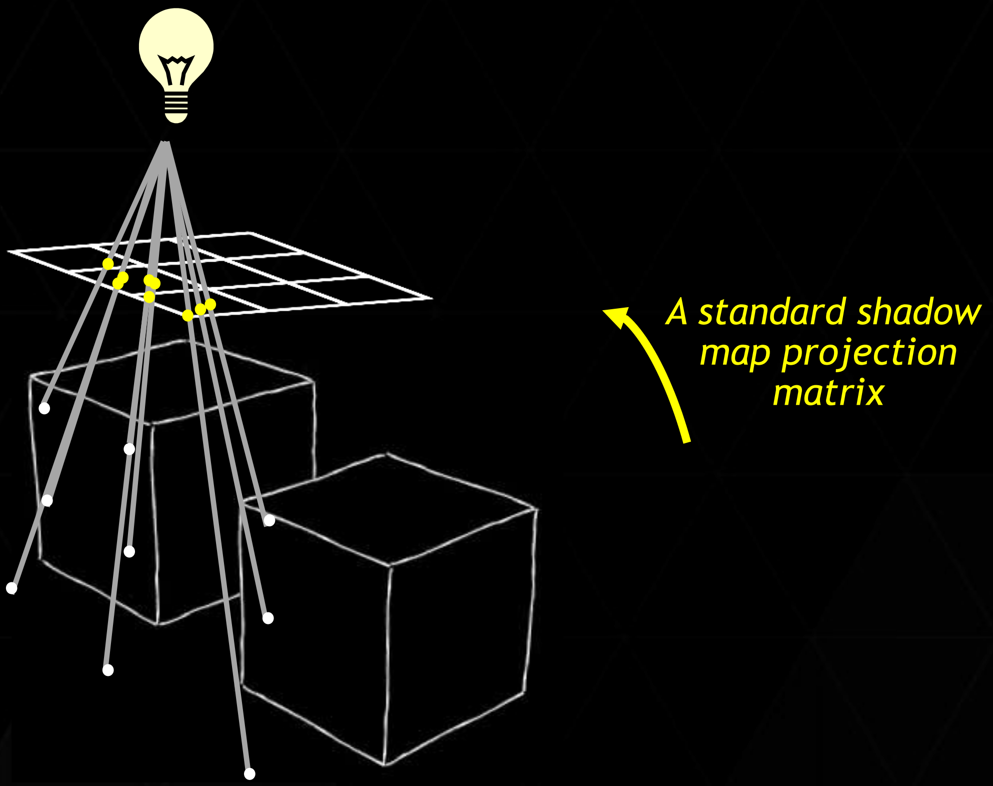


HOW TO DO THIS?

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

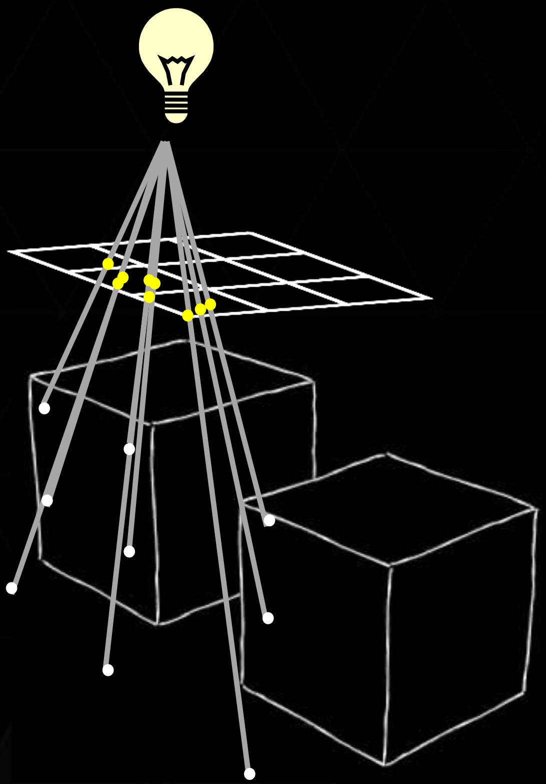
WHAT IS AN IRREGULAR Z-BUFFER?

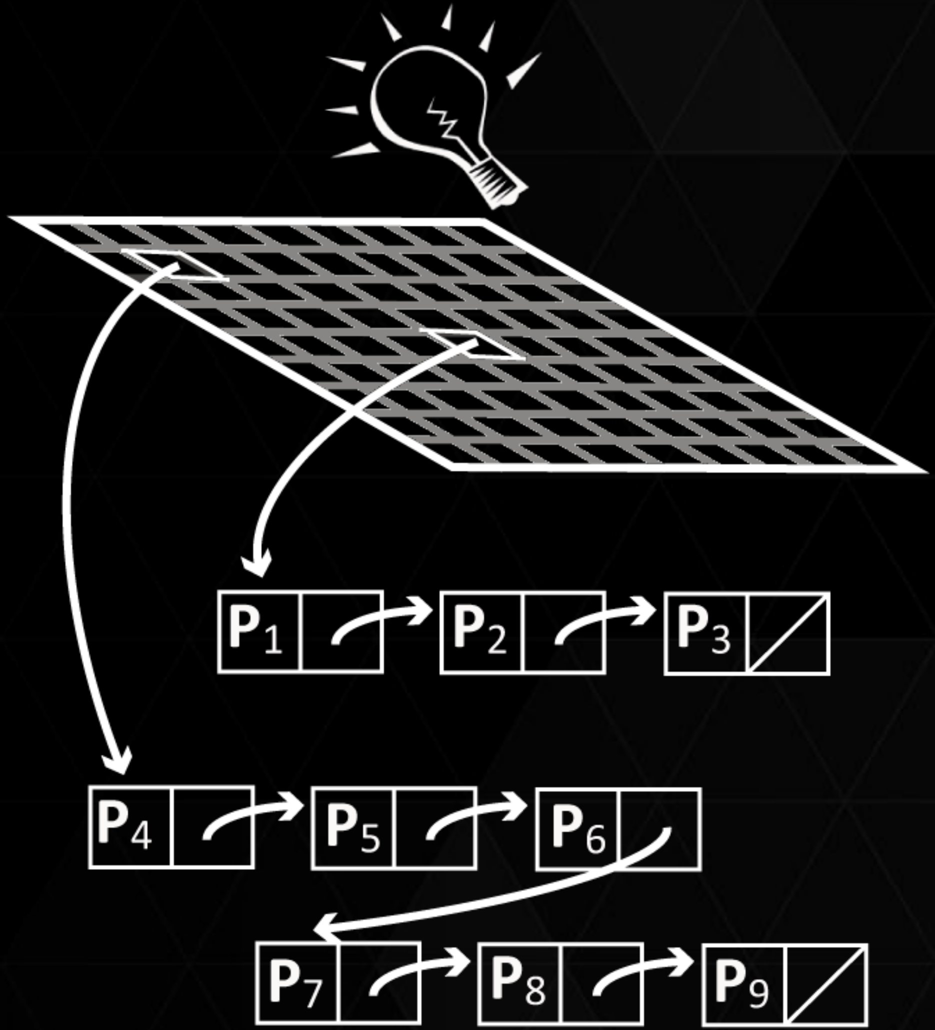
- Insert pixel samples (white dots) into light space grid at yellow samples



WHAT IS AN IRREGULAR Z-BUFFER?

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

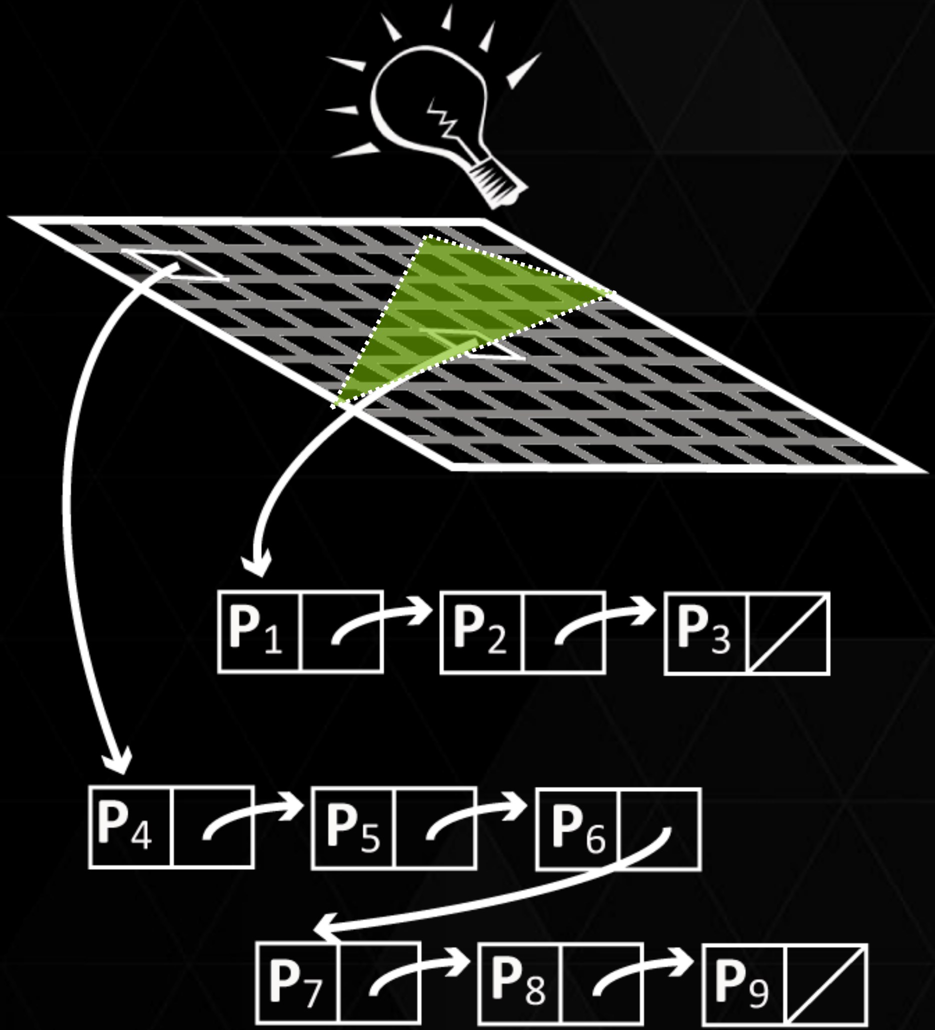




HOW DO YOU USE AN IZB?

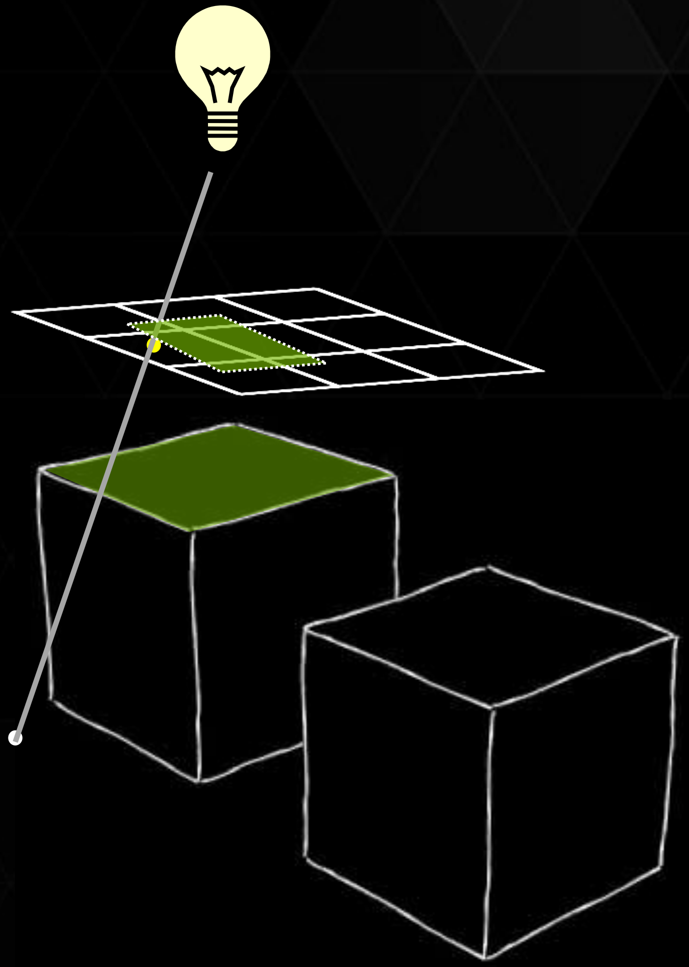
- Rasterize from light view
  - For each texel (partially) covered
    - Walk through list of eye-space pixels P_i
    - Test ray from P_i to the light
    - Update visibility at P_i

- We use eye-space buffer to store visibility for all pixels P_i



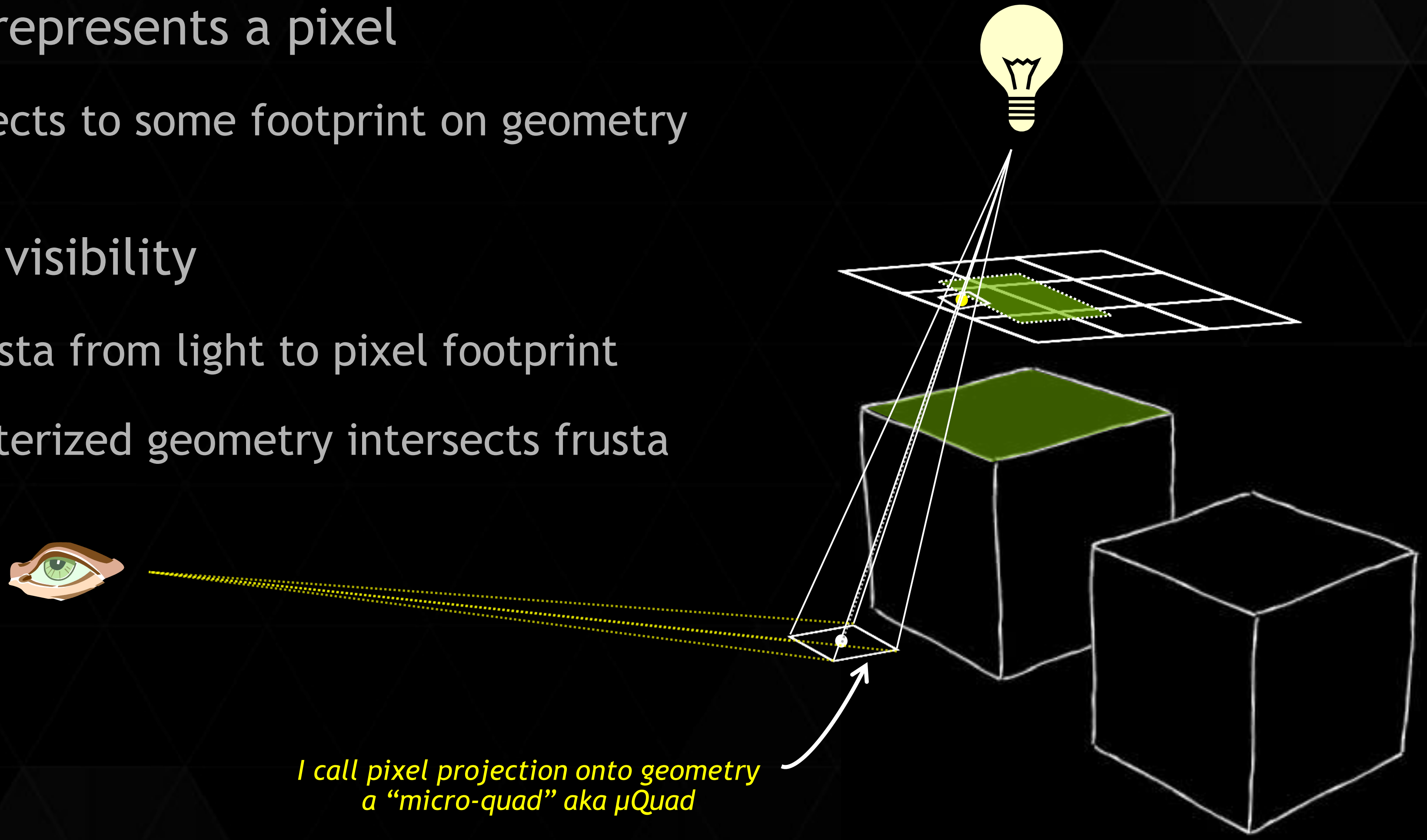
HOW DO YOU USE AN IZB?

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



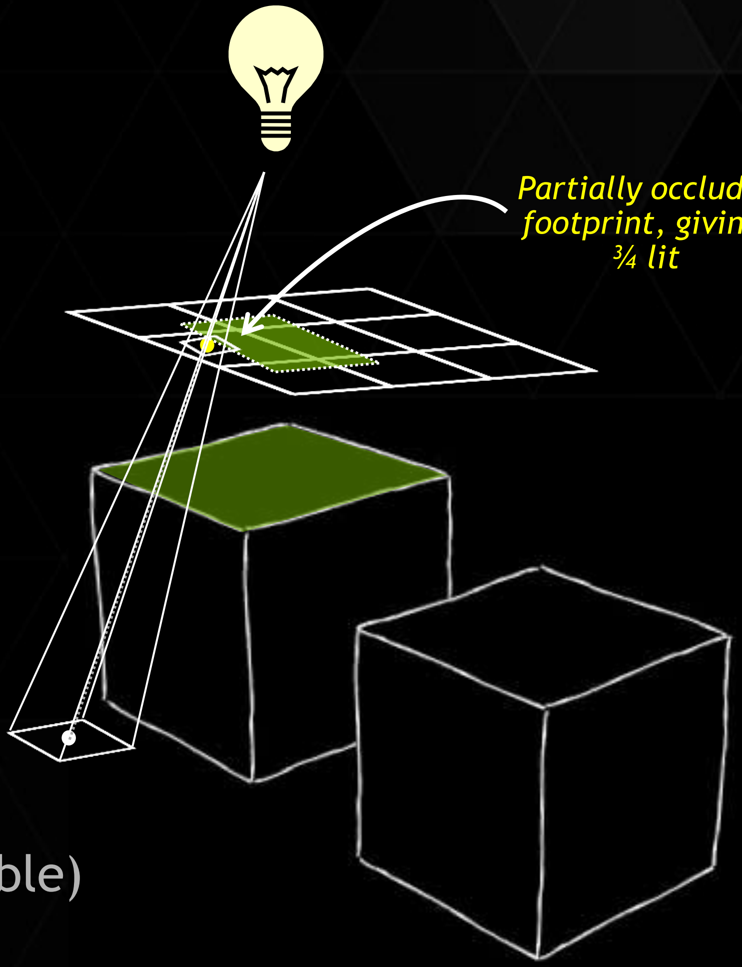
ADDING MULTIPLE SAMPLES PER PIXEL

- 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

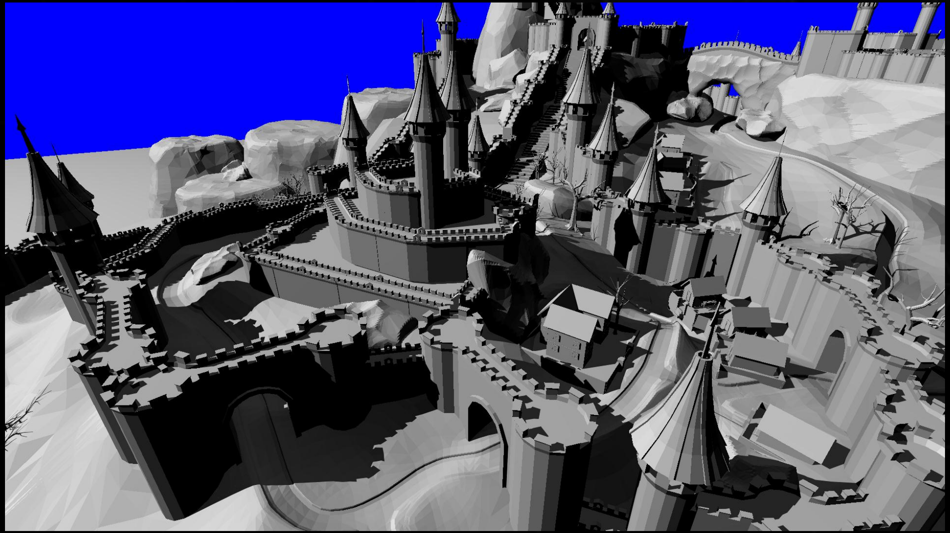
- Each sample represents a pixel
  - Pixel projects to some footprint on geometry
- When testing visibility
  - Create frusta from light to pixel footprint
  - Test if rasterized geometry intersects frusta
- Discretize visibility sampling on µQuad
  - We use pattern with 32 samples
  - Can be developer specified (currently a lookup table)
  - Each sample stores binary visibility

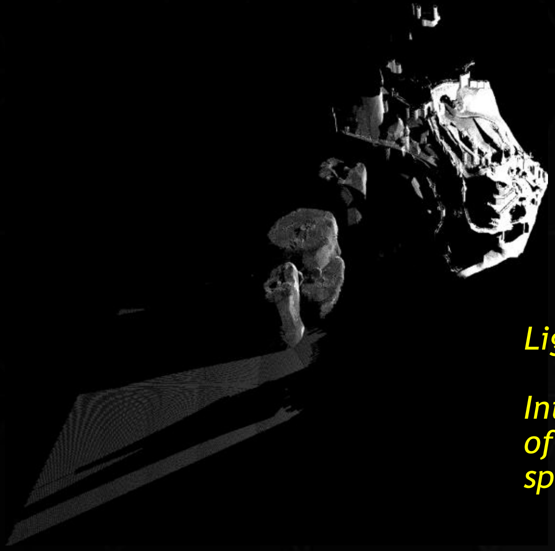


Problem with Irregular Z-Buffering

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 ARE WE DOING?

- 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 DO WE REDUCE COST?

- Either:
  - 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}.

REDUCING WORK

- How to reduce # fragments f_{ls}?
  - 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)

REDUCING WORK

- How to reduce # fragments f_{ls}?
  - 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)
  - 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 performance

REDUCING WORK

- How to reduce 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)

REDUCING WORK

- How to reduce 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 WORK

- Reducing *variance* in L? (i.e., cause $L_{max} \rightarrow L_{avg}$)
  - Match sampling rate between eye- & light-space (ideally 1:1)
    - Same goal as perspective, logarithm, adaptive, and cascaded shadow maps
  - The *key goal* for fast GPU implementation

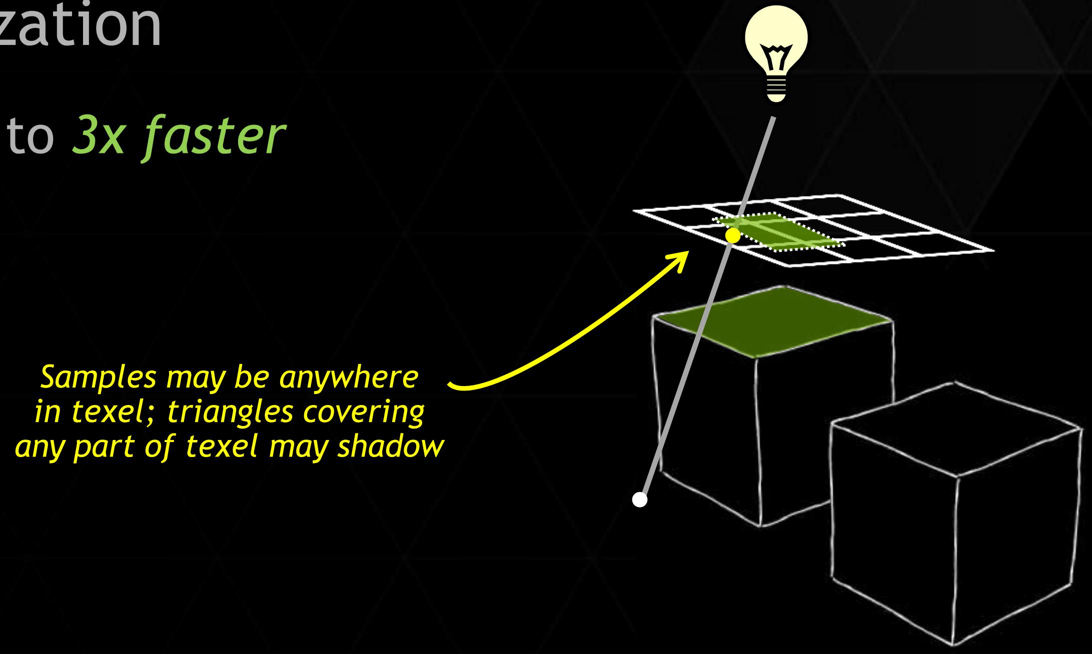
REDUCING WORK

- Reducing *variance* in L? (i.e., cause $L_{max} \rightarrow L_{avg}$)
  - Match sampling rate between eye- & light-space (ideally 1:1)
    - 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)

Miscellaneous Optimizations

GENERAL GPU OPTIMIZATIONS

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



GENERAL GPU OPTIMIZATIONS

- IZBs require conservative rasterization
  - 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

- List traversal induces long dependency chains
  - Hide latency via *software pipelining*
  - *Avoid long latency* operations (e.g., int divide, modulo)

GENERAL GPU OPTIMIZATIONS

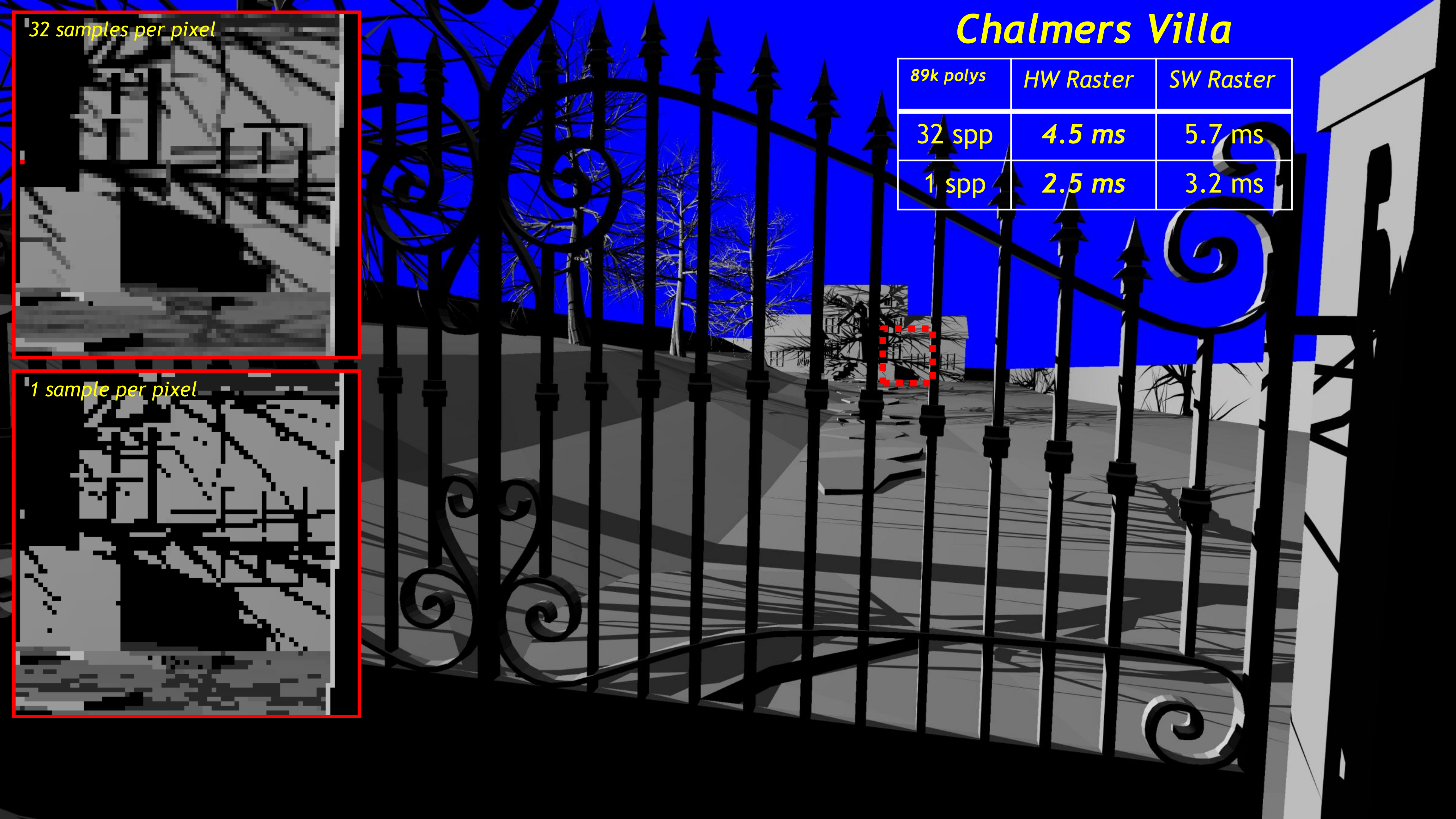
- List traversal induces long dependency chains
  - Hide latency via *software pipelining*
  - *Avoid long latency* operations (e.g., int divide, modulo)
- Reduce SIMD divergence
  - *Flatten control flow* as much as possible

Results

(All numbers at 1080p on a GeForce GTX 980)

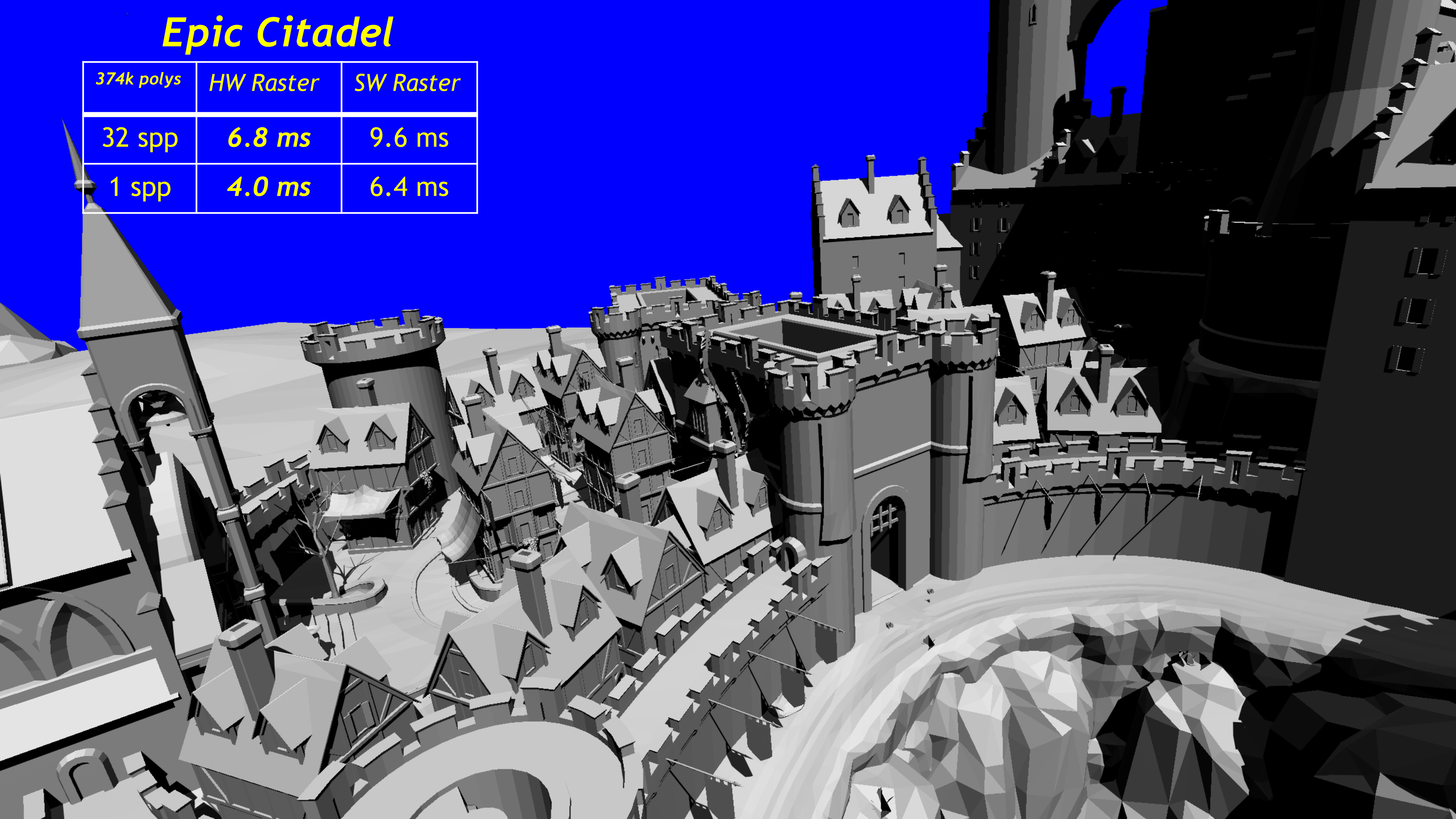
Chalmers Villa

89k polys	HW Raster	SW Raster
32 spp	***4.5 ms***	5.7 ms
1 spp	***2.5 ms***	3.2 ms



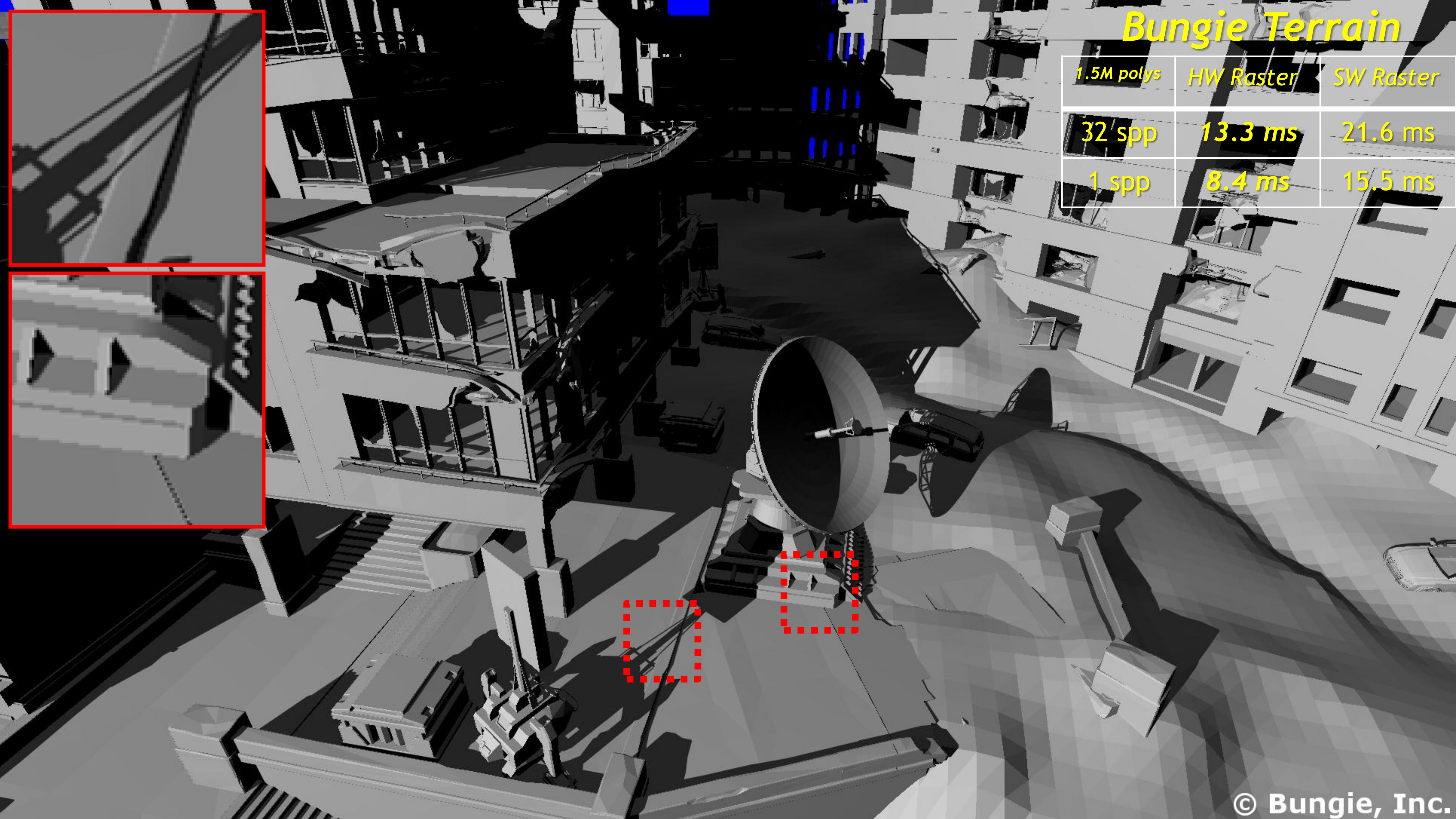
Epic Citadel

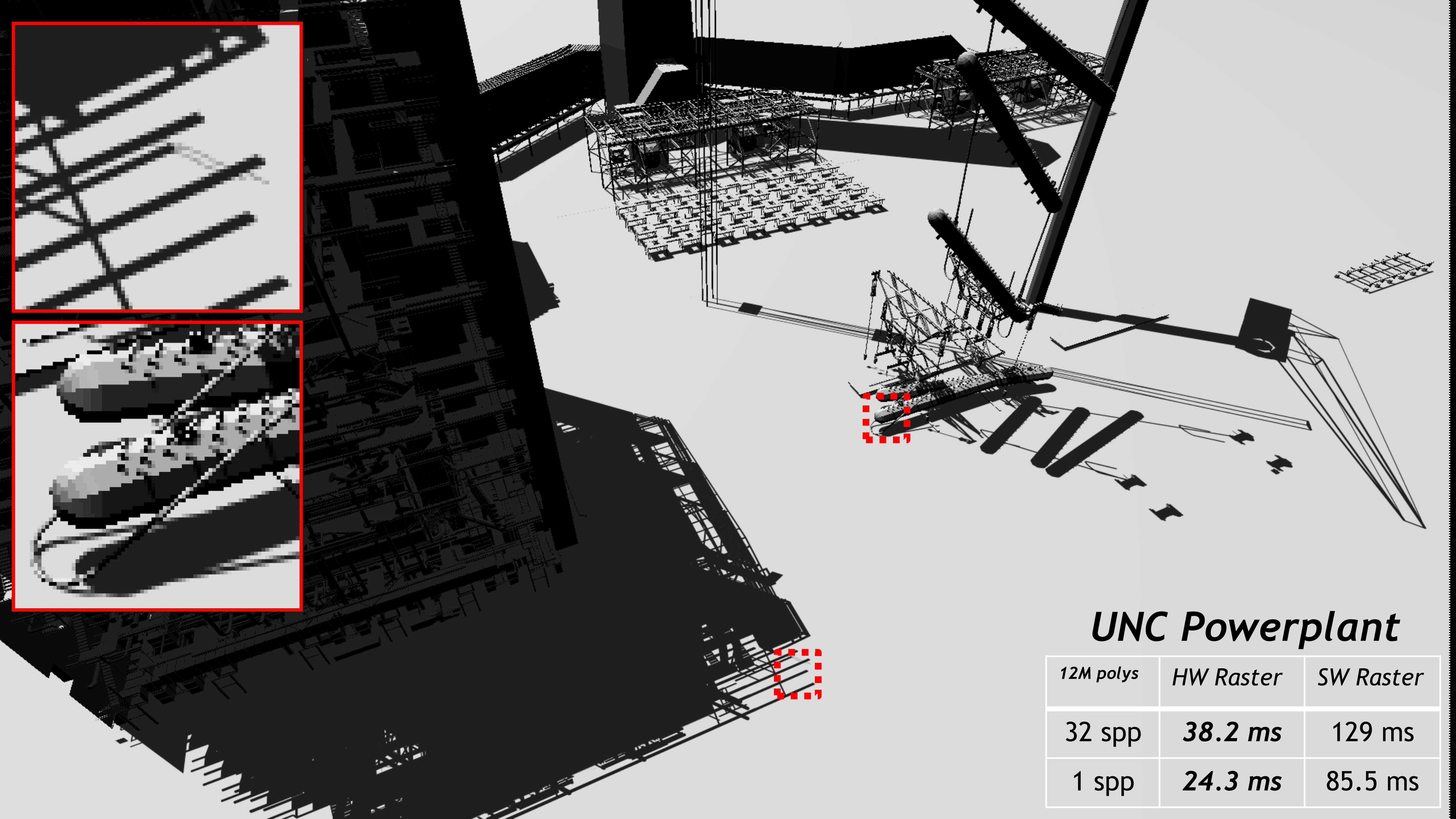
374k polys	HW Raster	SW Raster
32 spp	***6.8 ms***	9.6 ms
1 spp	***4.0 ms***	6.4 ms



Bungie Terrain

1.5M polys	HW Raster	SW Raster
32 spp	13.3 ms	21.6 ms
1 spp	8.4 ms	15.5 ms





UNC Powerplant

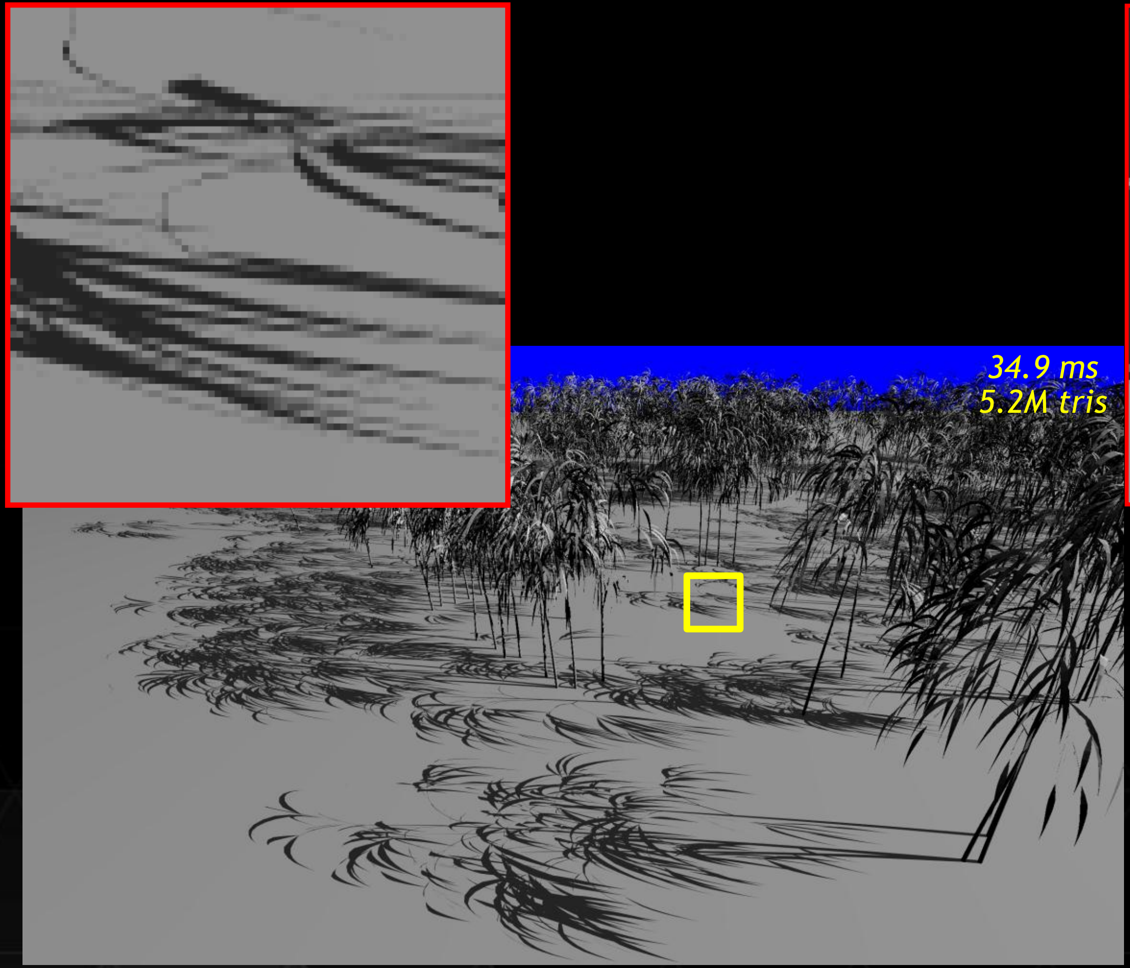
12M polys	*HW Raster*	*SW Raster*
32 spp	***38.2 ms***	129 ms
1 spp	***24.3 ms***	85.5 ms

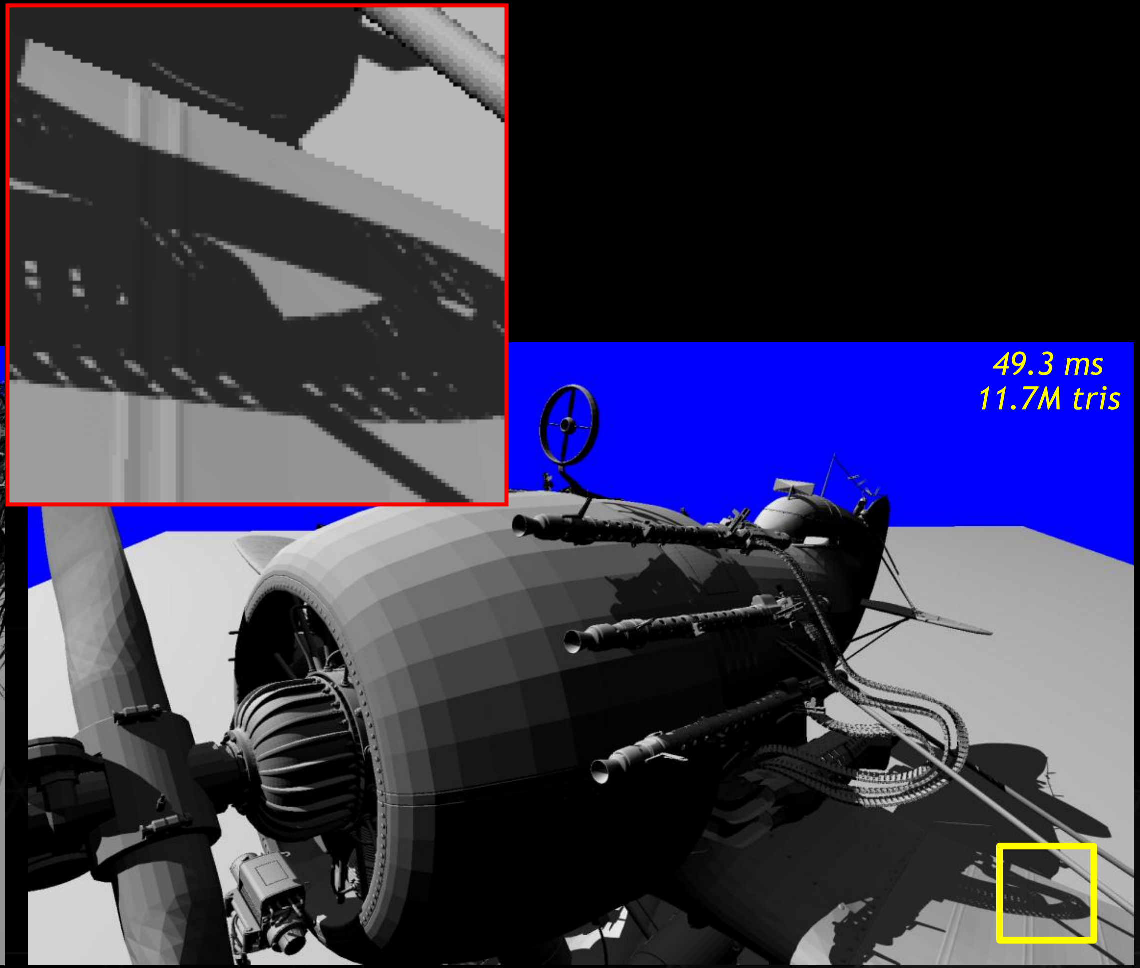
UNC Powerplant

12M polys	*HW Raster*	*SW Raster*
32 spp	***38.2 ms***	129 ms
1 spp	***24.3 ms***	85.5 ms

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
- Approximate 32 spp IZB insertion can break
  - Causes slight light leaking, esp. for finely tessellated models in distance
- Some sub-pixel robustness tricks needed for 32 spp
  - To avoid shadow leaks at interpenetrating triangle boundaries





QUESTIONS?

cwyman@nvidia.com http://chriswyman.org @_cwyman_

Demo? Find me during poster / demo session!