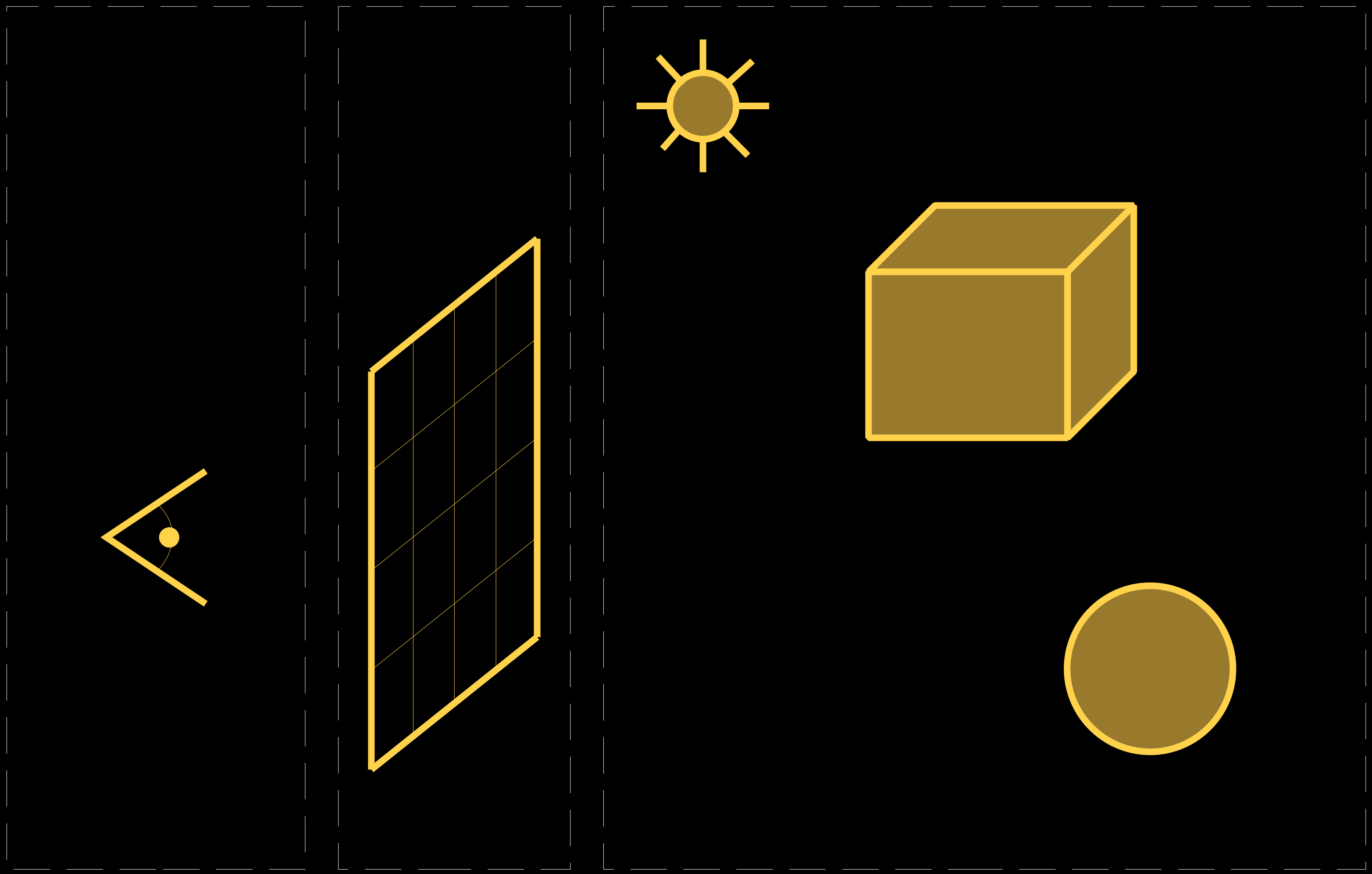
Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU

David Roger, Ulf Assarsson, Nicolas Holzschuch

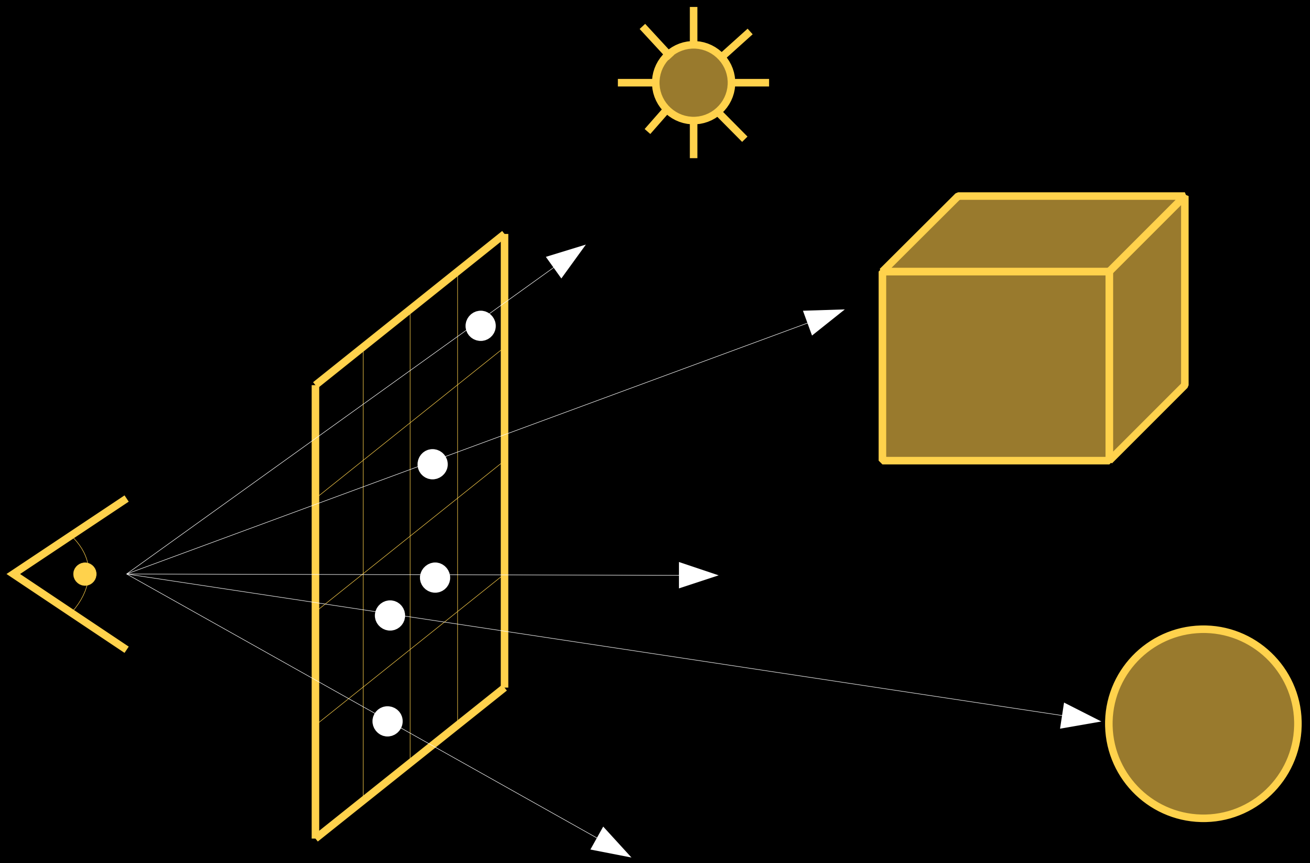
Grenoble University
Chalmers University of Technology

Ray Tracing

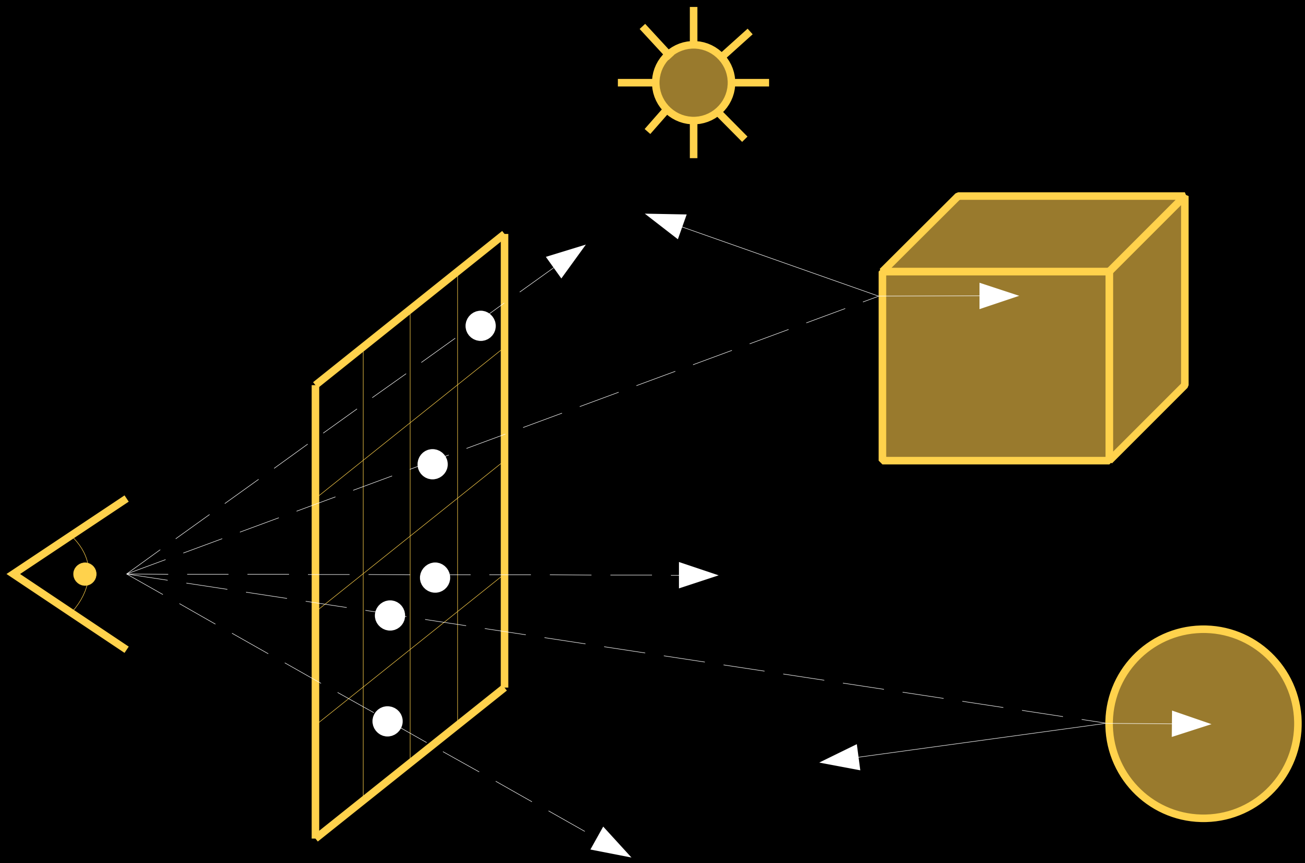


Camera Screen 3D Scene

Ray Tracing

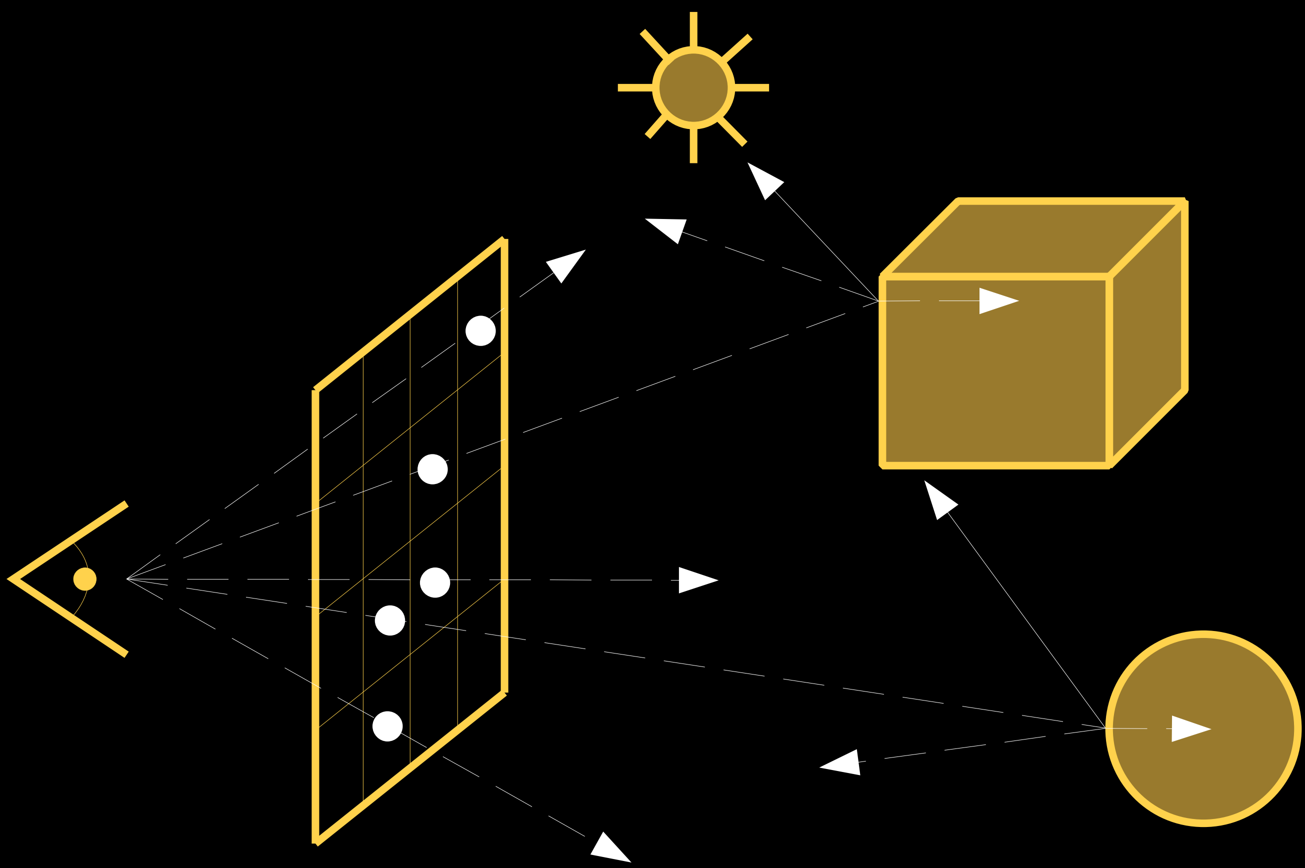


Whitted Ray Tracing



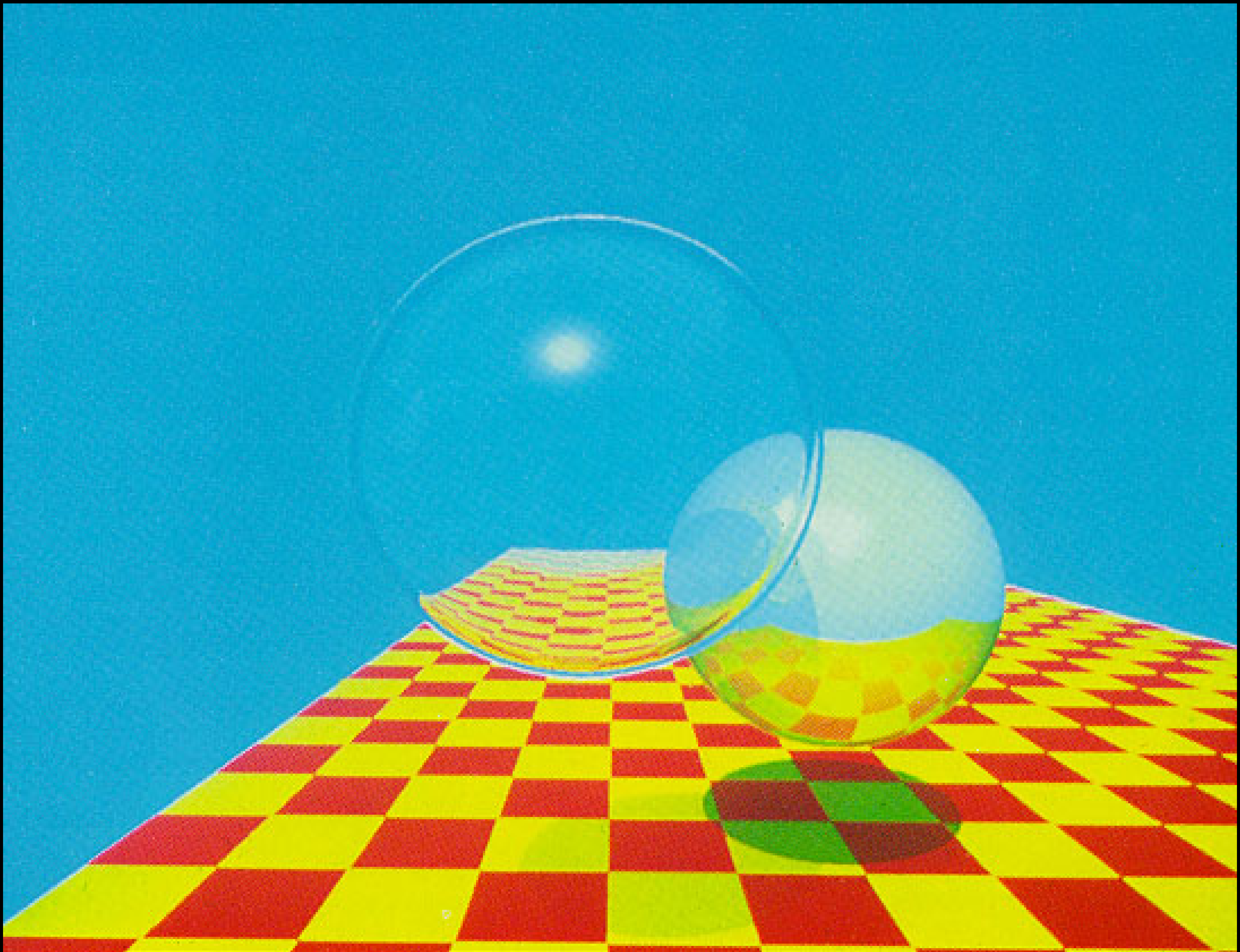
[Whitted80]
An Improved Illumination Model for Shaded Display

Whitted Ray Tracing



[Whitted80]
An Improved Illumination Model for Shaded Display

Whitted Ray Tracing



Interactive Rendering

- Ray tracing
  - Primary + specular + shadow rays
  - Scene Hierarchy
    - Not well suited for dynamic scenes
    - Log in #Triangles
  - Fast

Interactive Rendering

- Rasterization
  - Primary rays only
    + tricks (shadows …)
  - Dynamic scenes
  - Linear in #Triangles
  - Very fast

Splinter Cell: Conviction

Assassin's Creed

Side By Side

- Rasterization
  - Primary rays only + tricks (shadows …)
  - Dynamic scenes
  - Linear in #Triangles
  - Very fast
- Ray tracing
  - Primary + specular + shadow rays
  - Scene Hierarchy
    - Not well suited for dynamic scenes
    - Log in #Triangles
  - Fast

Side By Side

- Rasterization
  - Primary rays only
    + tricks (shadows …)
  - Dynamic scenes
  - Linear in #Triangles
  - Very fast

- Ray tracing
  - Primary + specular + shadow rays
  - Scene Hierarchy
    - Not well suited for dynamic scenes
    - Log in #Triangles
  - Fast

Take the best of both !
Primary : rasterization
Others : ray tracing

Our Approach

- Rasterization
  - Primary rays only
    + tricks (shadows ...)
  - Dynamic scenes
  - Linear in #Triangles
  - Very fast
- Our ray tracing
  - ~~Primary +~~ specular + shadow rays
  - ~~Scene~~ Ray Hierarchy
    - ~~Not~~ well suited for dynamic scenes
    - ~~Log~~ Linear in #Triangles
  - Fast

Take the best of both !
Primary : rasterization
Others : ray tracing

Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Conclusion

Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Conclusion

Previous Works: CPU Ray Tracing

- Static scenes, primary rays
  - [RSH05] Reshetov *et al.*, Multi Level Ray Tracing
- Dynamic scenes, primary rays
  - [WBS07] Wald *et al.*, Dynamic Bounding Volume Hierarchy
  - [WIK*06] Wald *et al.*, Coherent Grid
  - [LYTM06] Lauterbach *et al.*, Bounding Volume Hierarchy
- Dynamic scenes, secondary rays
  - [WK06] Wächter and Keller, Bounding Interval Hierarchy

Previous Works: GPU Ray Tracing

- Static scenes, kd-tree
  - [FS05] Foley and Sugerman
  - [HSHH07] Horn *et al.*
- Static scenes, Bounding Volume Hierarchy
  - [TS05] Thrane and Simonsen
- Dynamic scenes, primary rays
  - [CHCH06] Carr *et al.*, Geometry Images

Previous Works: Ray Hierarchy

- CPU, not interactive
  - [Ama84,HH84,AK87,GH98]
- GPU
  - [Szé06] Szécsi
    - hierarchy with 2 levels
    - refraction only

Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Conclusion

Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Conclusion

Main Features

- Rasterization for primary rays
- Ray tracing for secondary rays
- Ray hierarchy: rebuilt at each frame
- Runs completely on GPU

Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

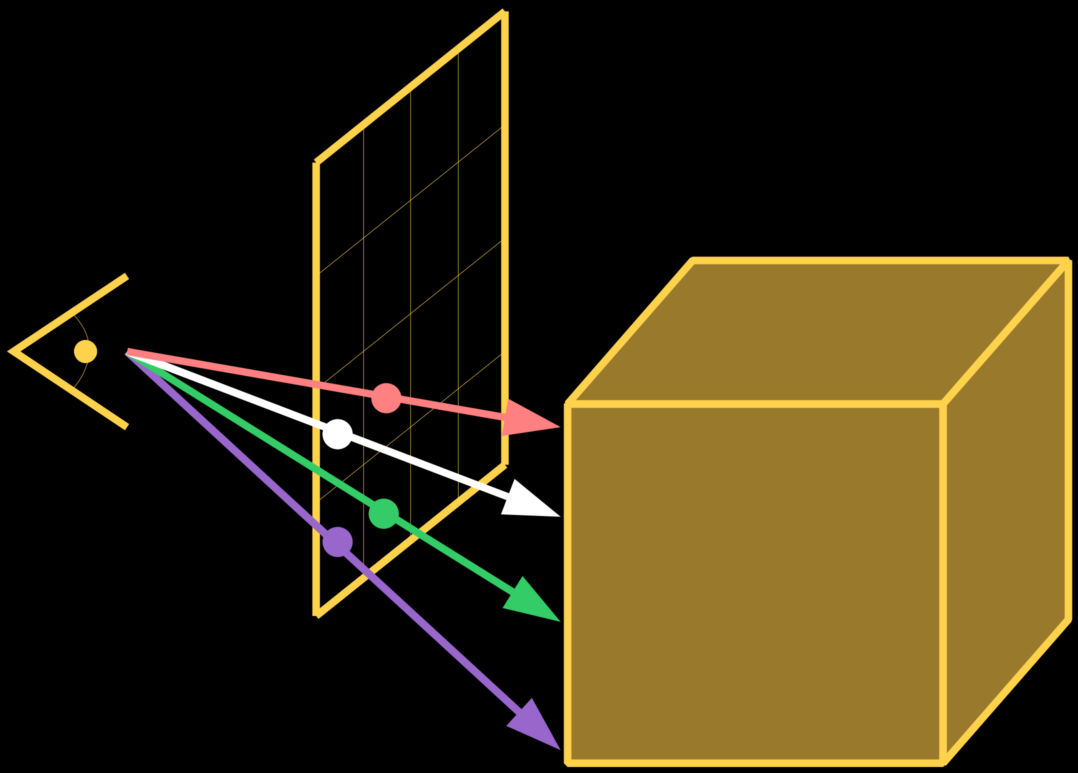
Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

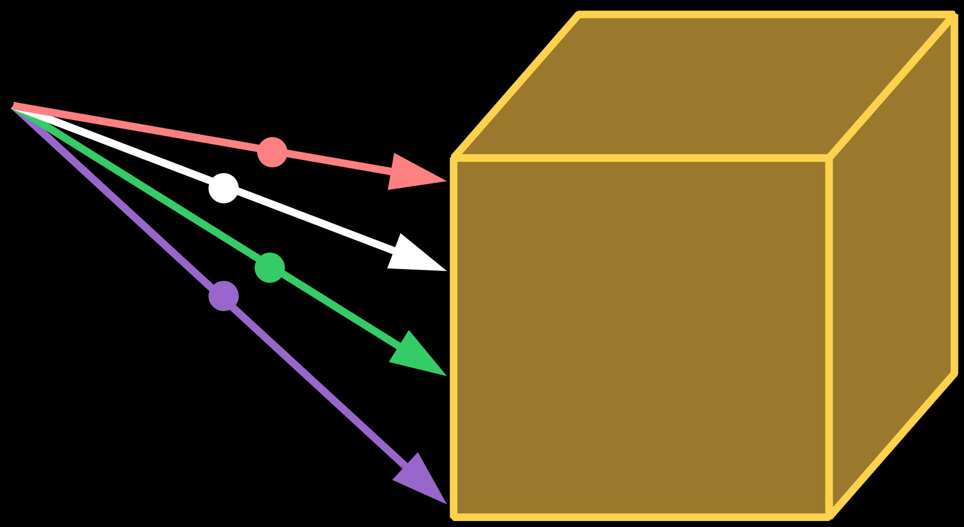
Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

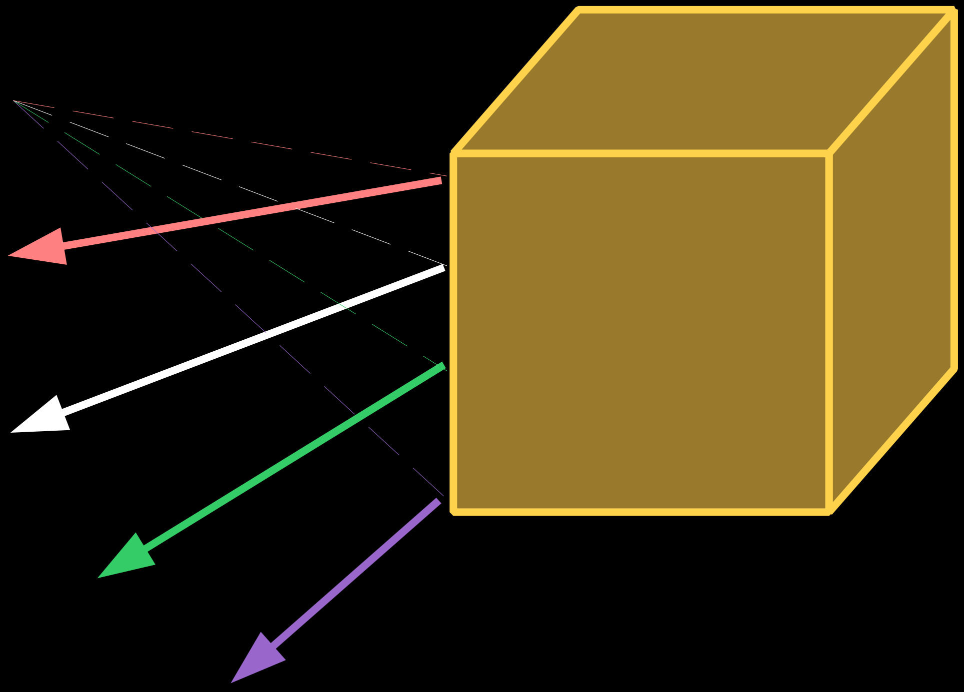
Primary Rays



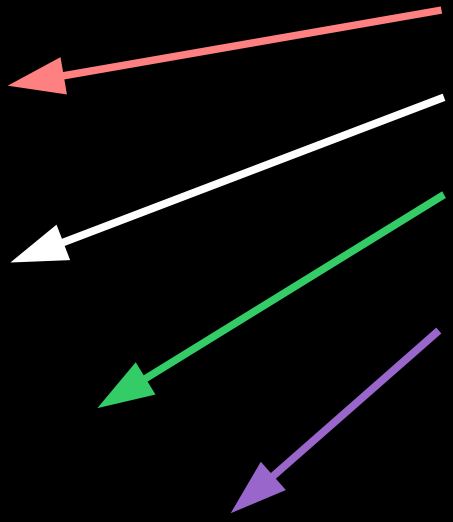
Primary Rays



Secondary Rays

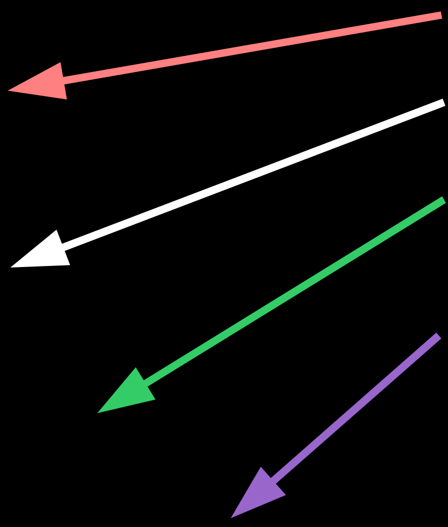


Secondary Rays

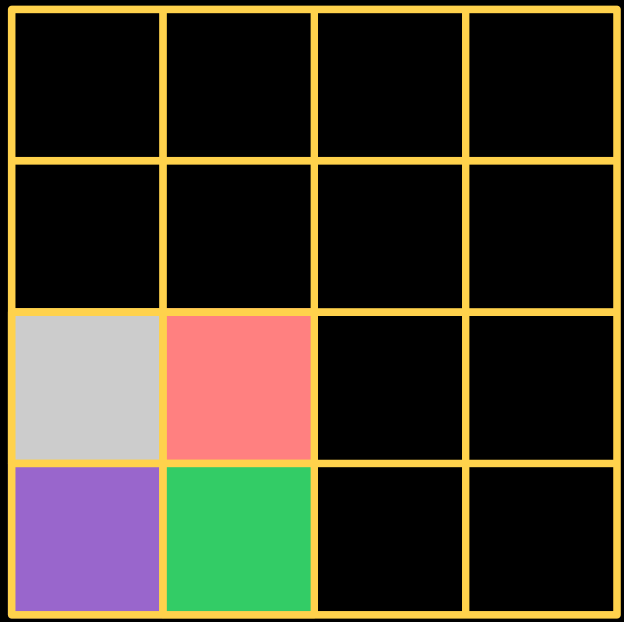


Secondary Rays

Leaves of the hierarchy



Corresponding pixels



Stored in 2 textures (position + direction)
Same size as the screen

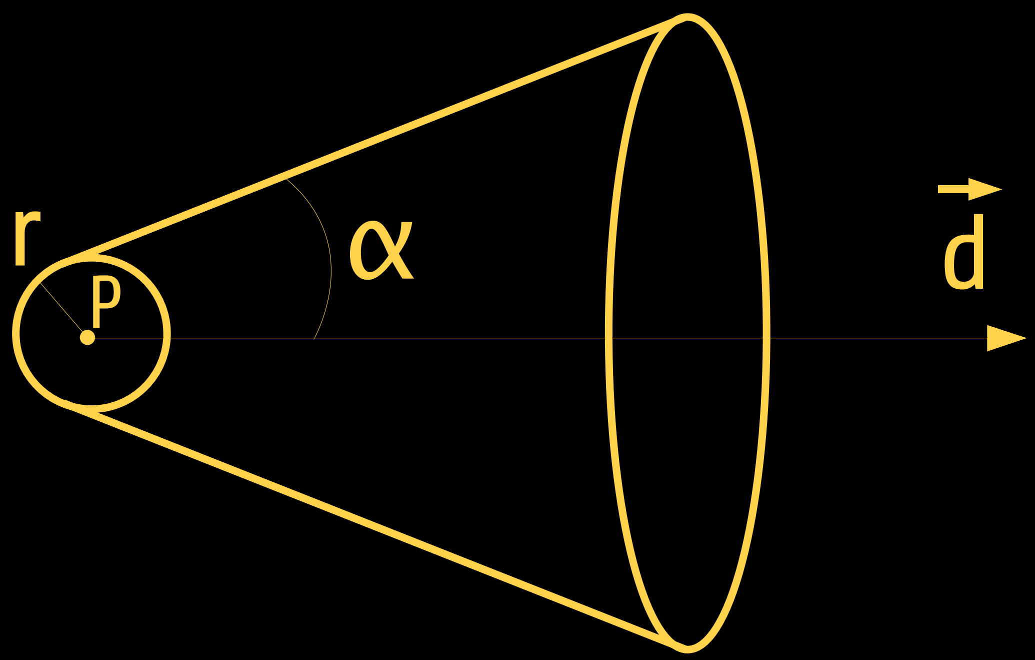
Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

Algorithm Overview

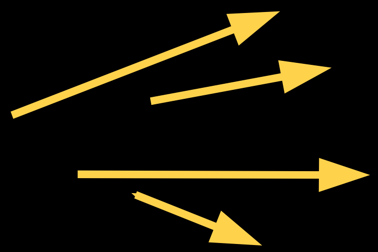
1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

Hierarchy Node



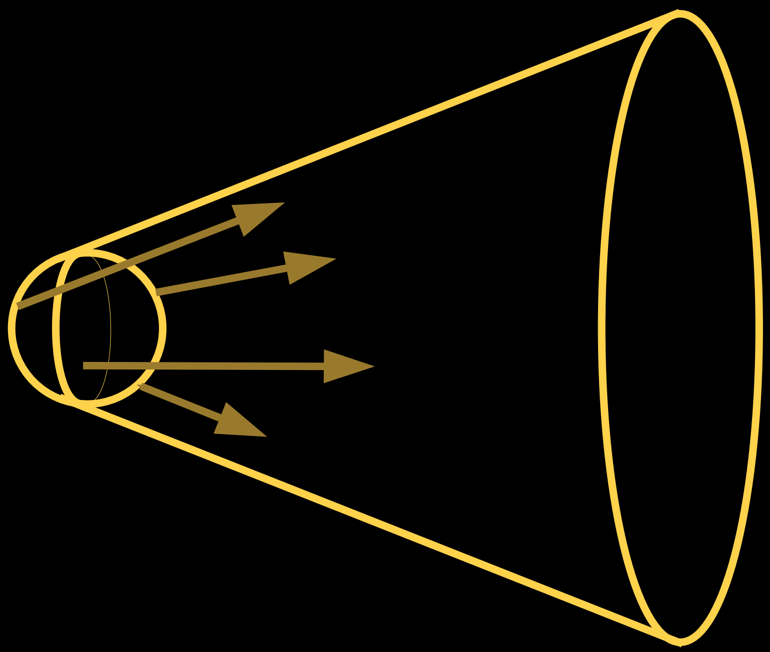
Hierarchy Construction

- Bottom-up



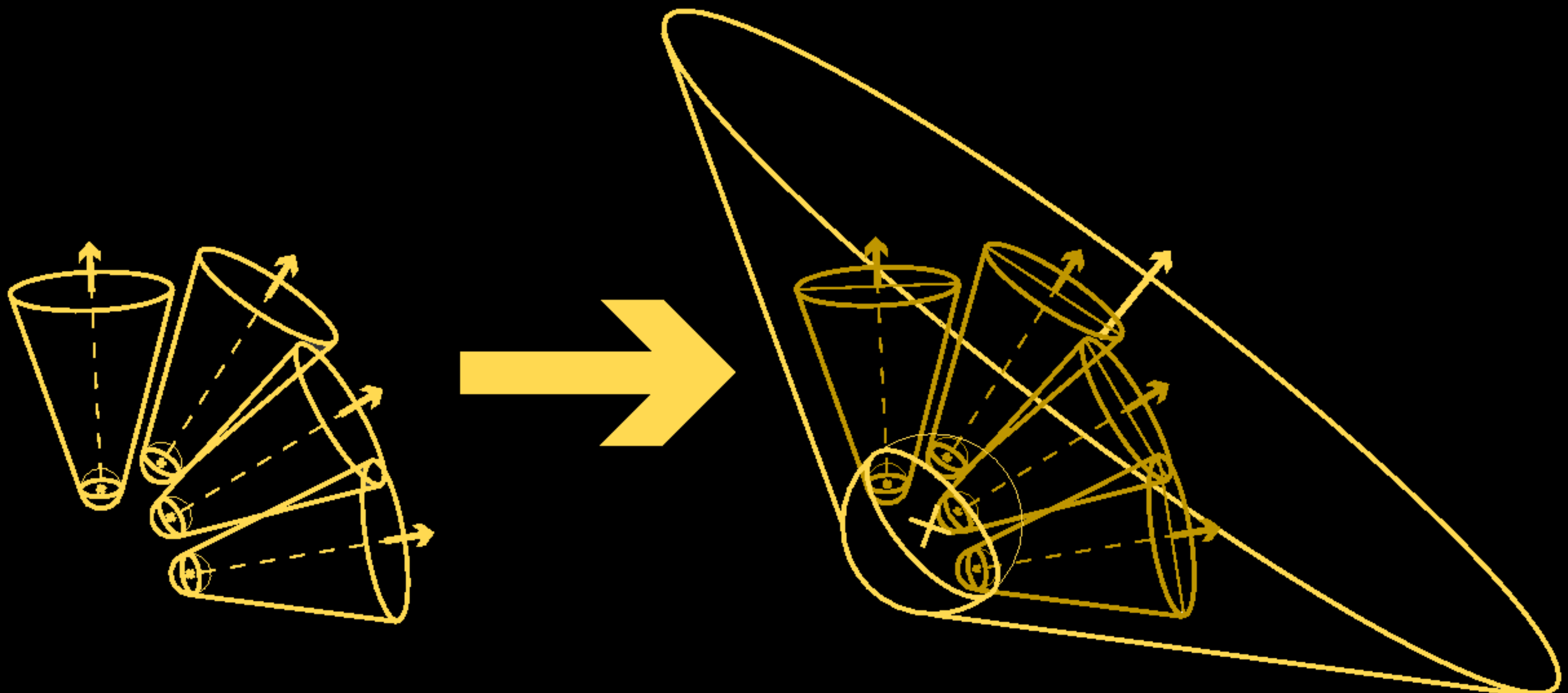
Hierarchy Construction

- Bottom-up



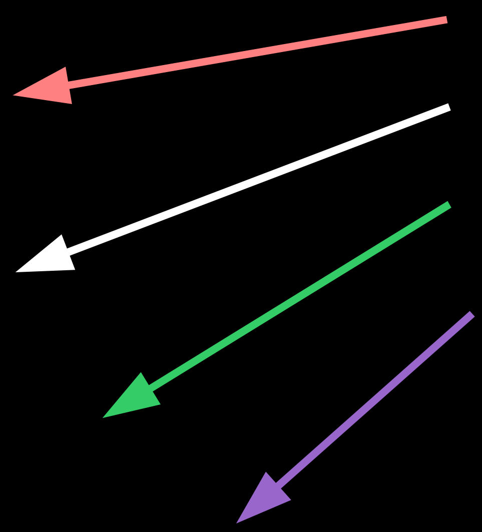
Hierarchy Construction

- Bottom-up

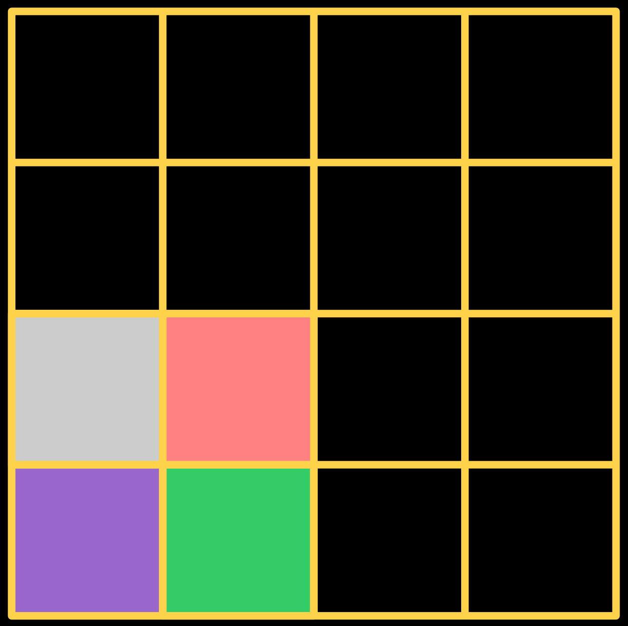


Hierarchy Construction

Secondary rays: leaves

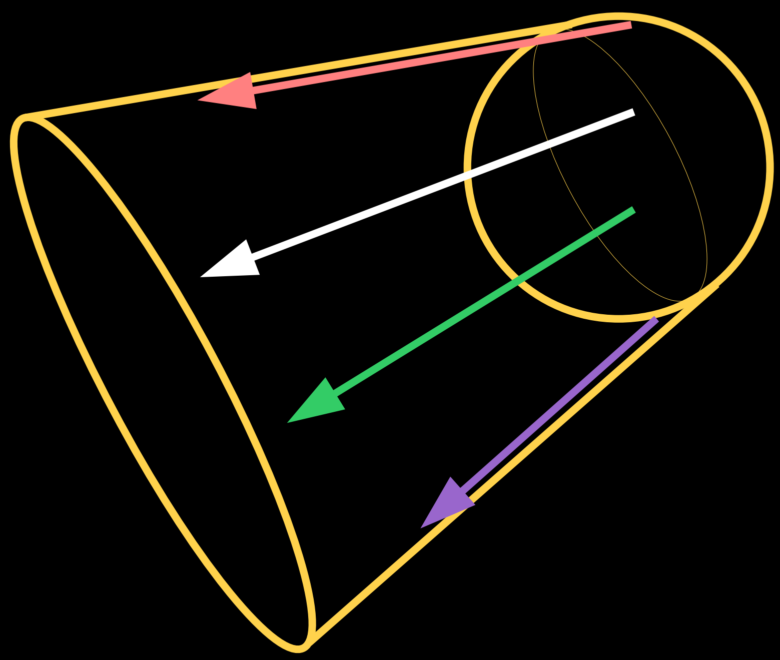


Corresponding pixels

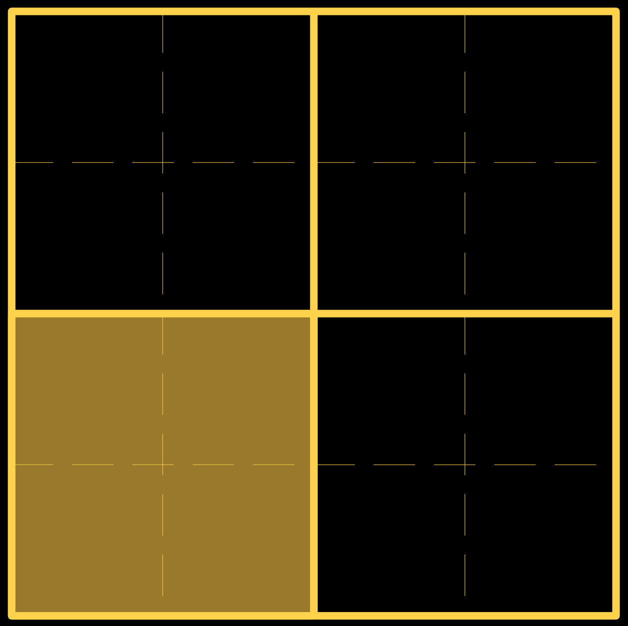


Hierarchy Construction

Level 1



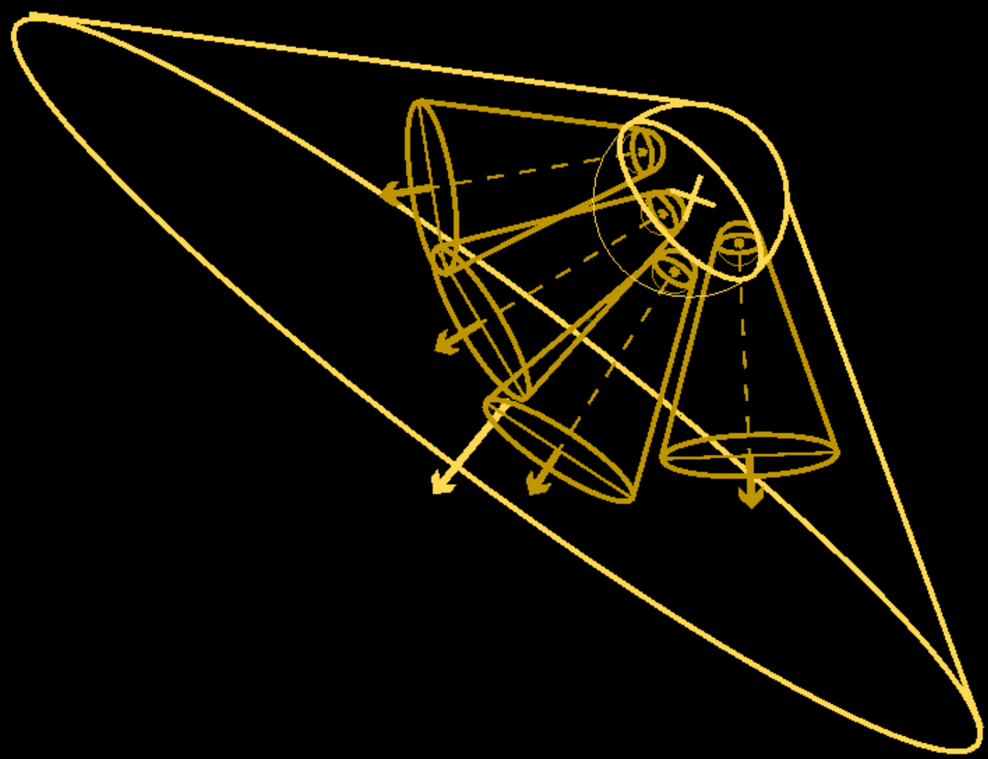
Corresponding pixels



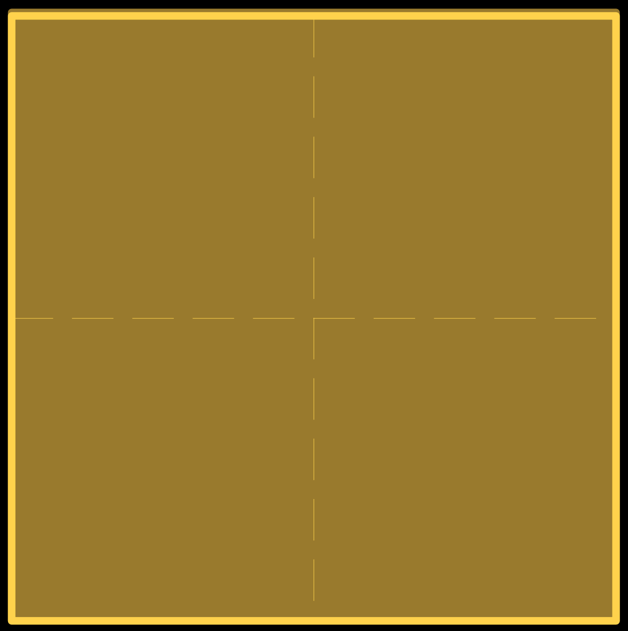
Similar to mimap generation

Hierarchy Construction

Root



Corresponding pixels



Similar to mimap generation

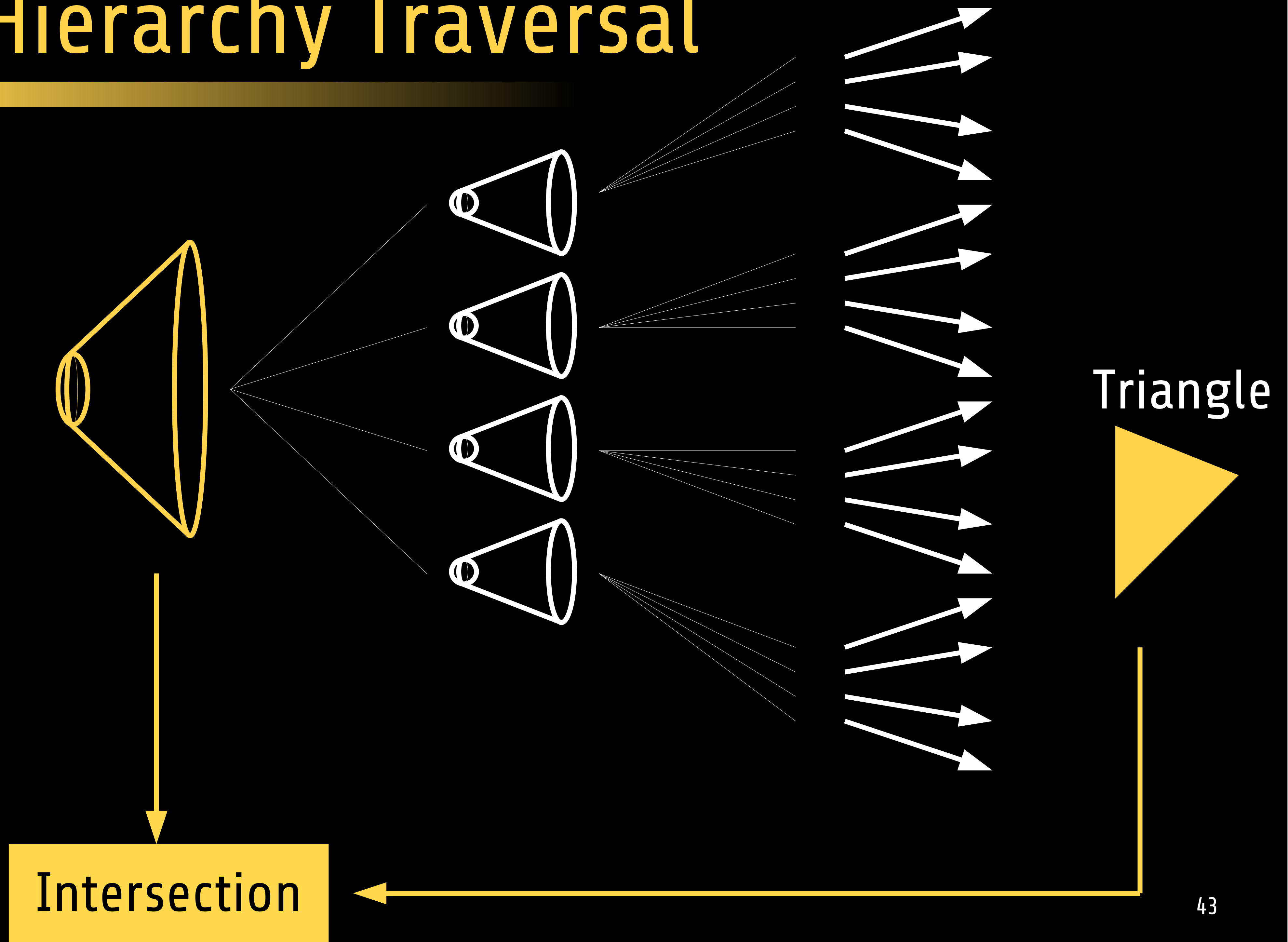
Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
   - 1 Million pixels: 10 levels
   - Similar to mip-map generation
   - Very fast: 1M pixels on GPU < 2ms
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

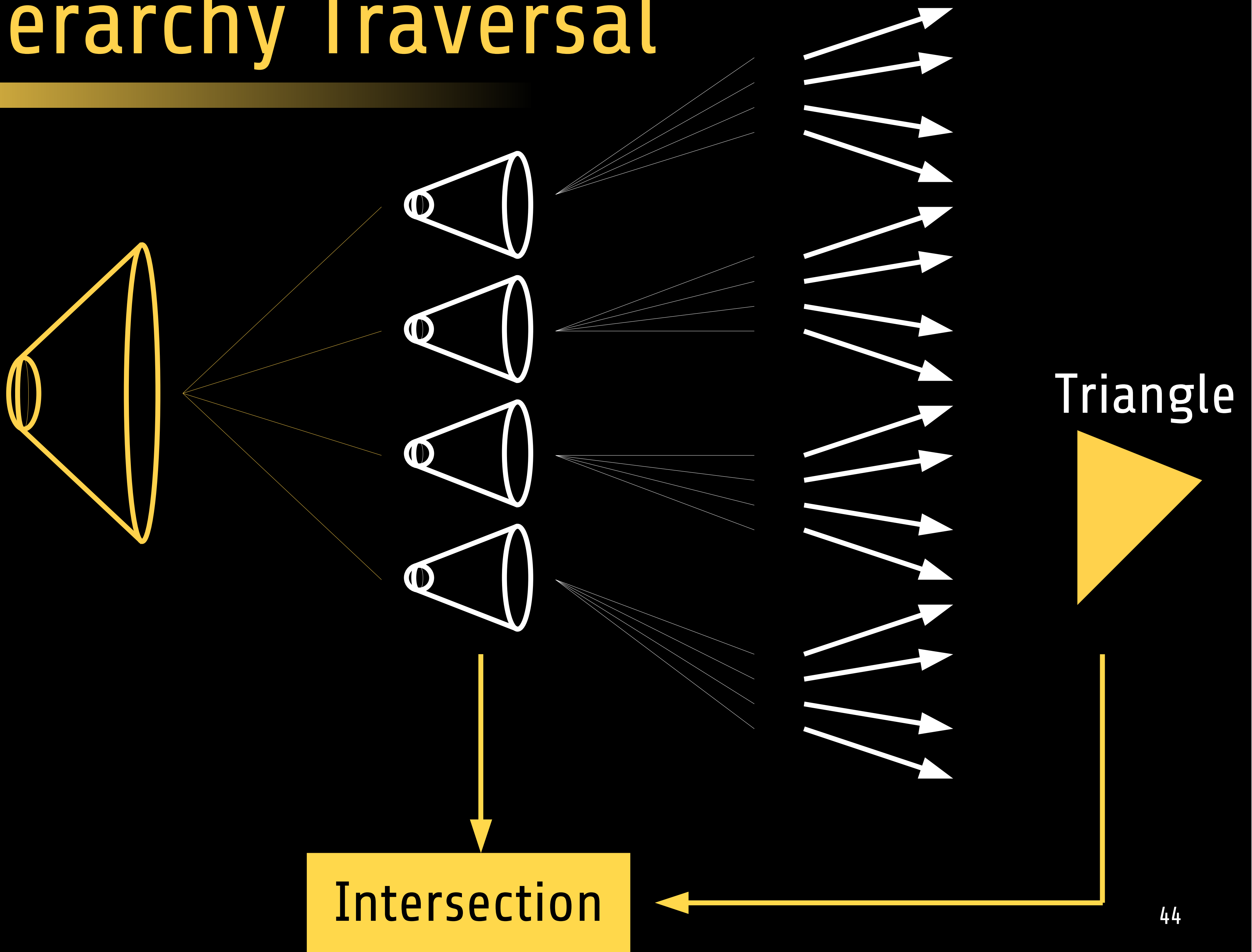
Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

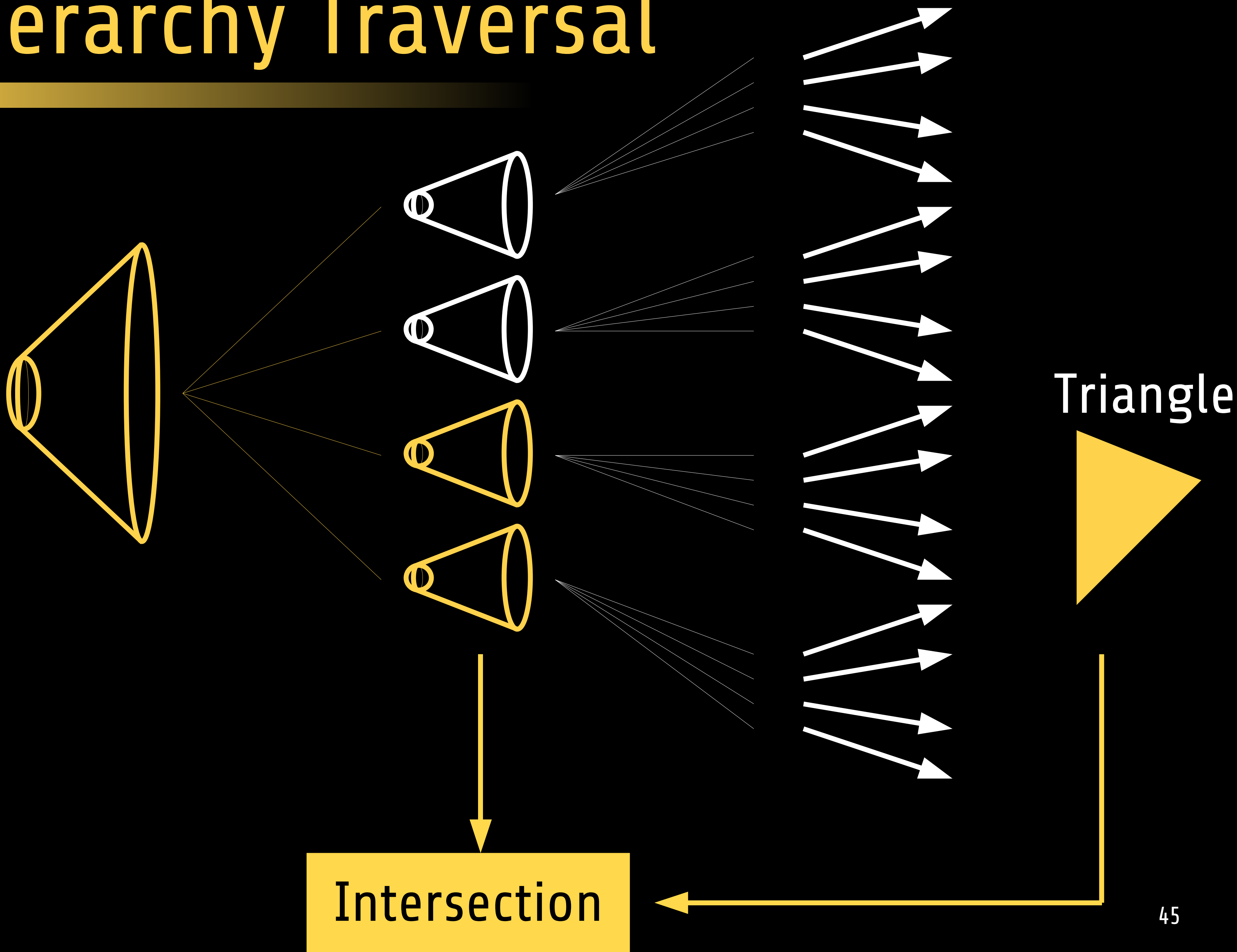
Hierarchy Traversal



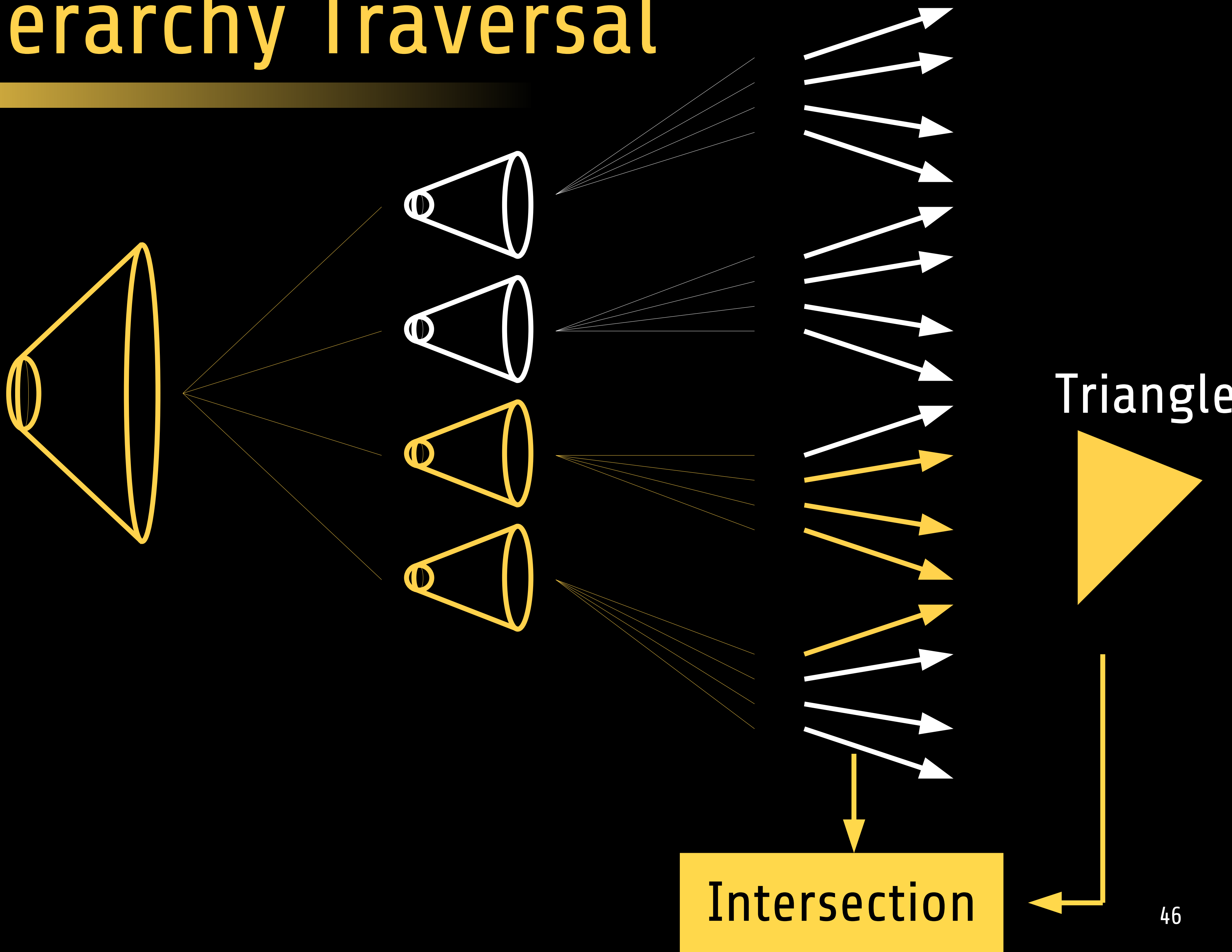
Hierarchy Traversal



Hierarchy Traversal



Hierarchy Traversal



Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
   - Process triangles in parallel
   - Same execution path length, minimal branching
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

Talk Structure

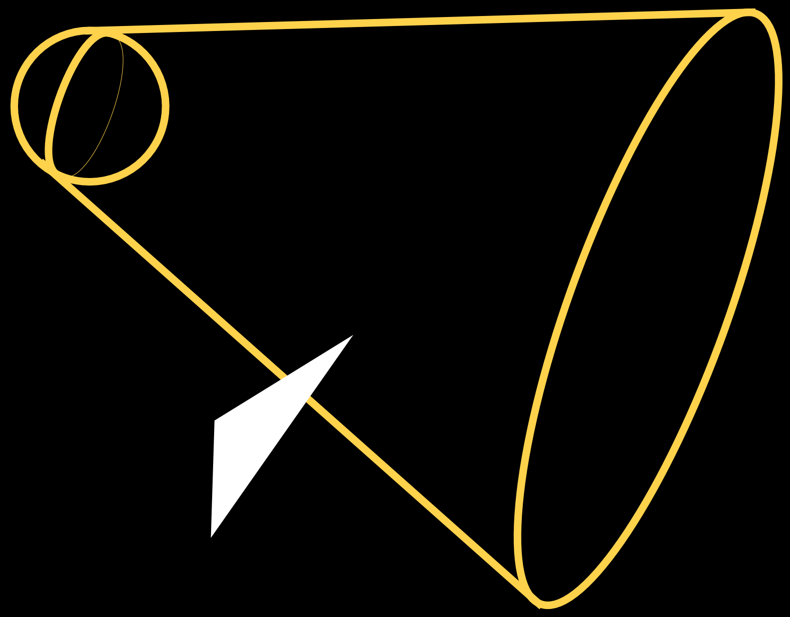
- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Conclusion

Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Conclusion

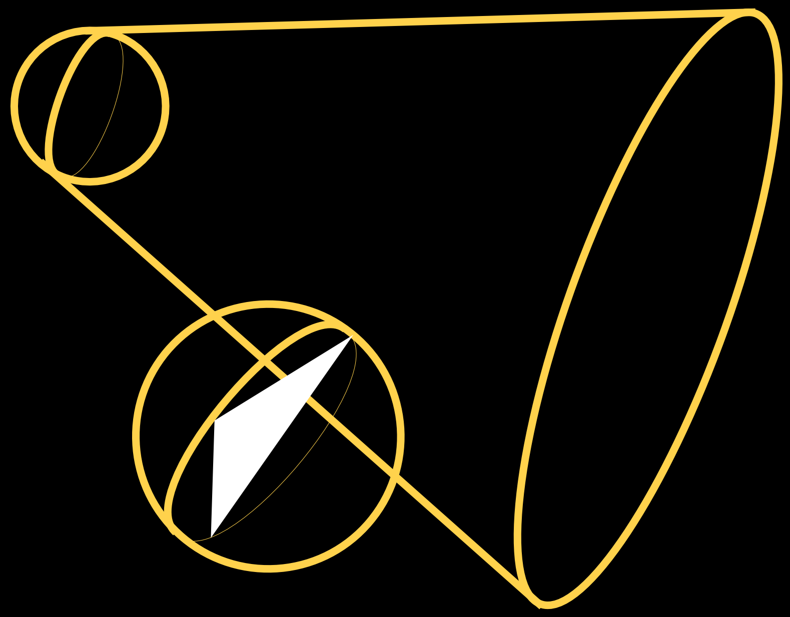
Triangle-Node Intersection

- Cone-triangle test is expensive
- Use the bounding sphere of the triangle
  - Approximation, but faster overall



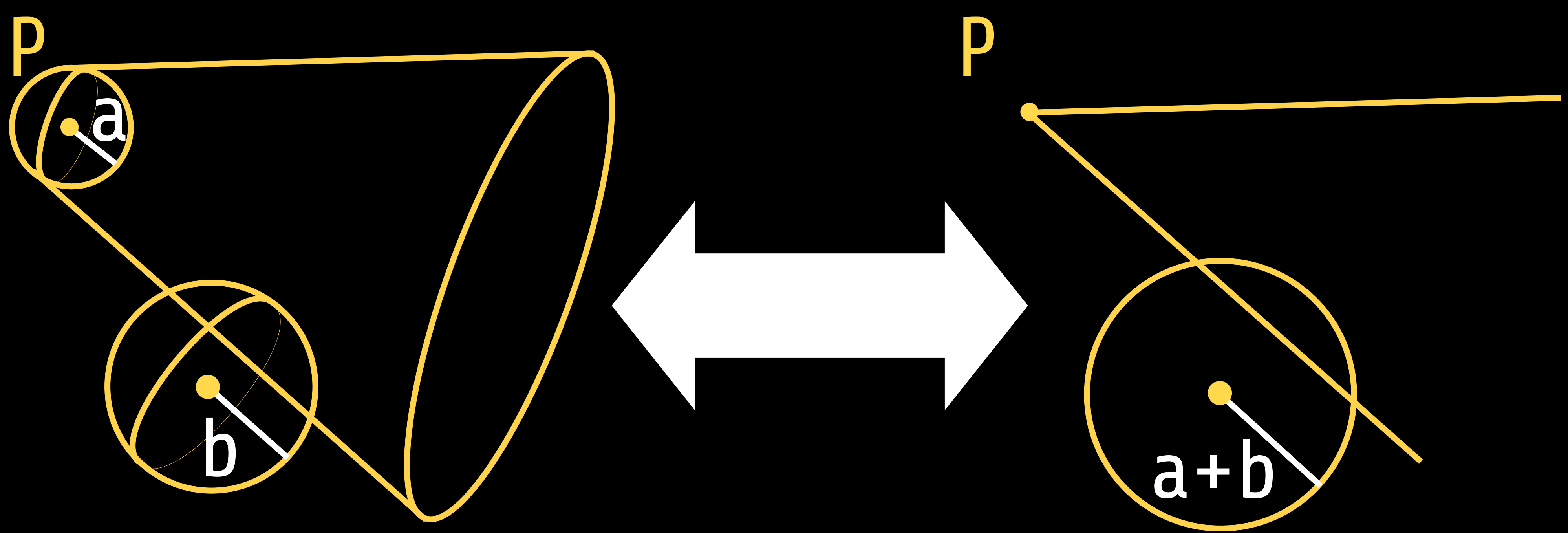
Triangle-Node Intersection

- Cone-triangle test is expensive
- Use the bounding sphere of the triangle
  - Approximation, but faster overall

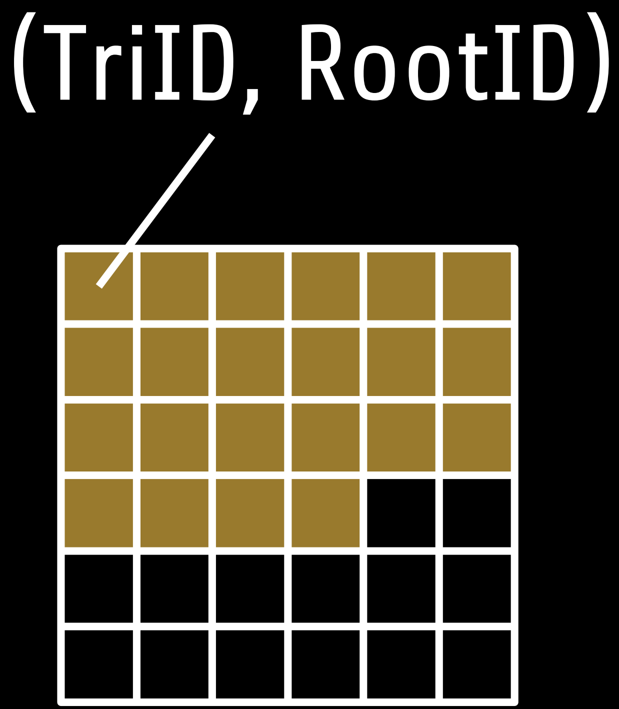


Triangle-Node Intersection

- Cone-triangle test is expensive
- Use the bounding sphere of the triangle
  - Approximation, but faster overall

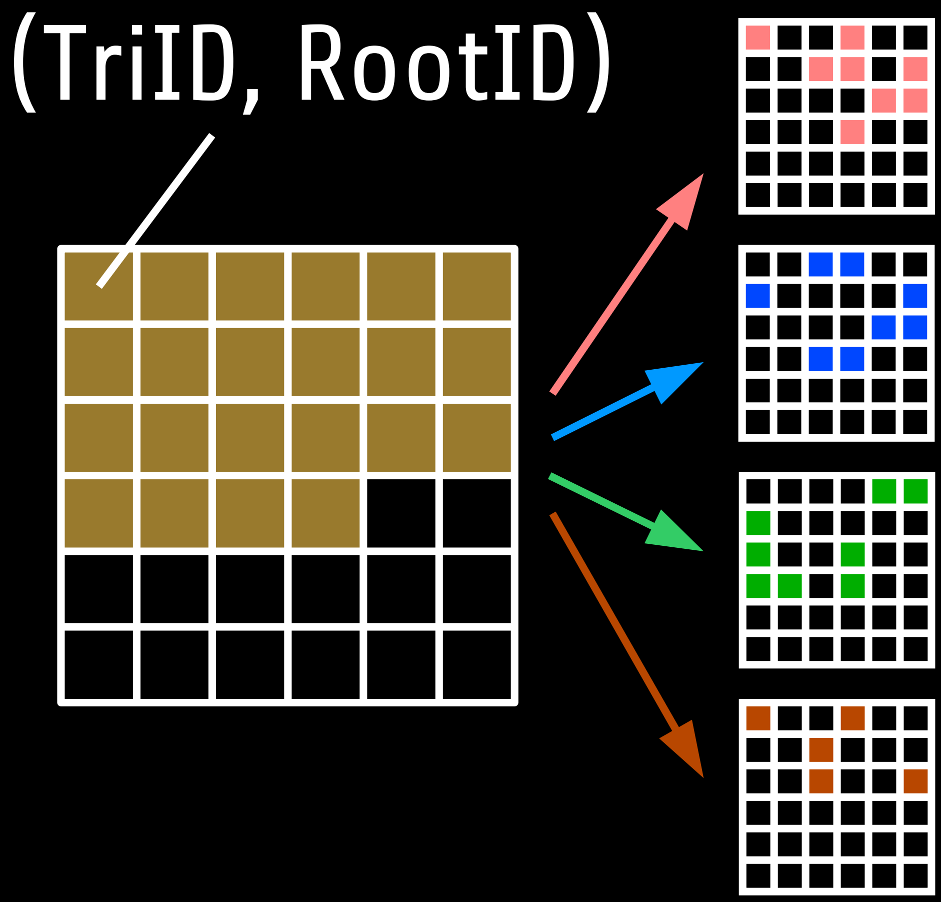


GPU Traversal Implementation



- Top-down
- All triangles processed in parallel
- (Triangle,Cone) pairs
  - Triangle has to be tested against Cone's children

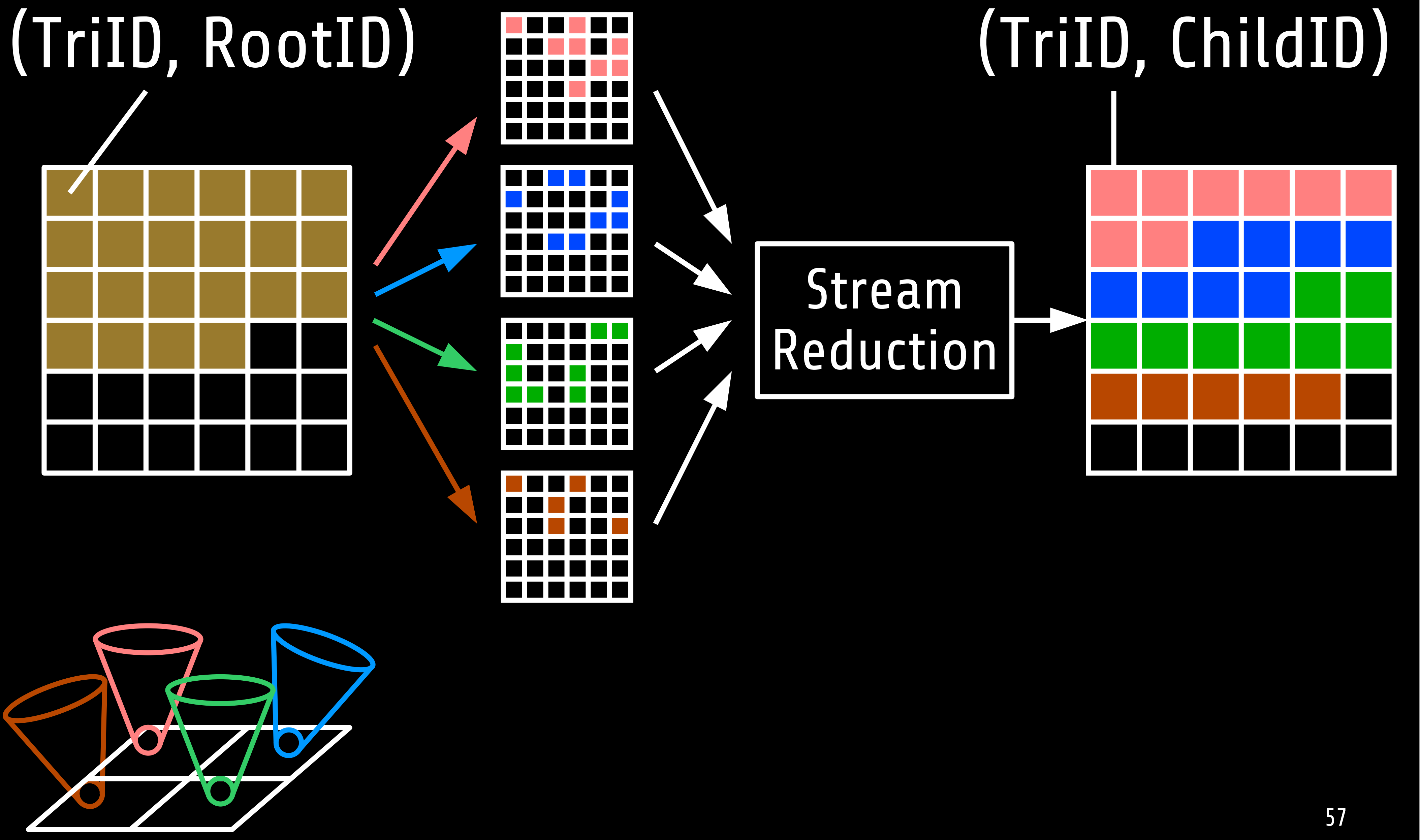
GPU Traversal Implementation



- 4 intersection tests
  - 1 rendering pass
  - 4 render targets
- If intersection
  - Then (TriID, ChildID)
  - Else NULL

First hierarchy level

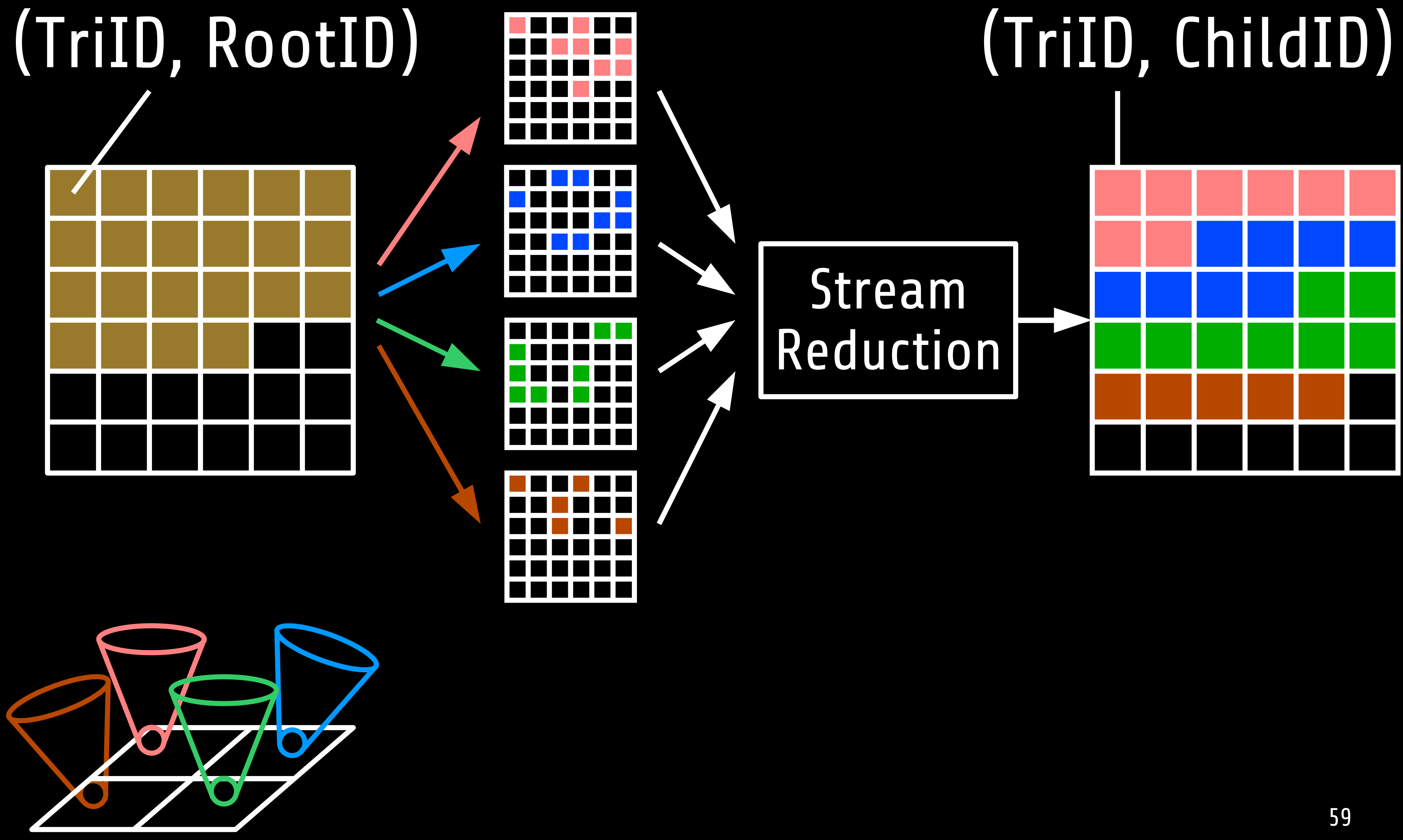
GPU Traversal Implementation



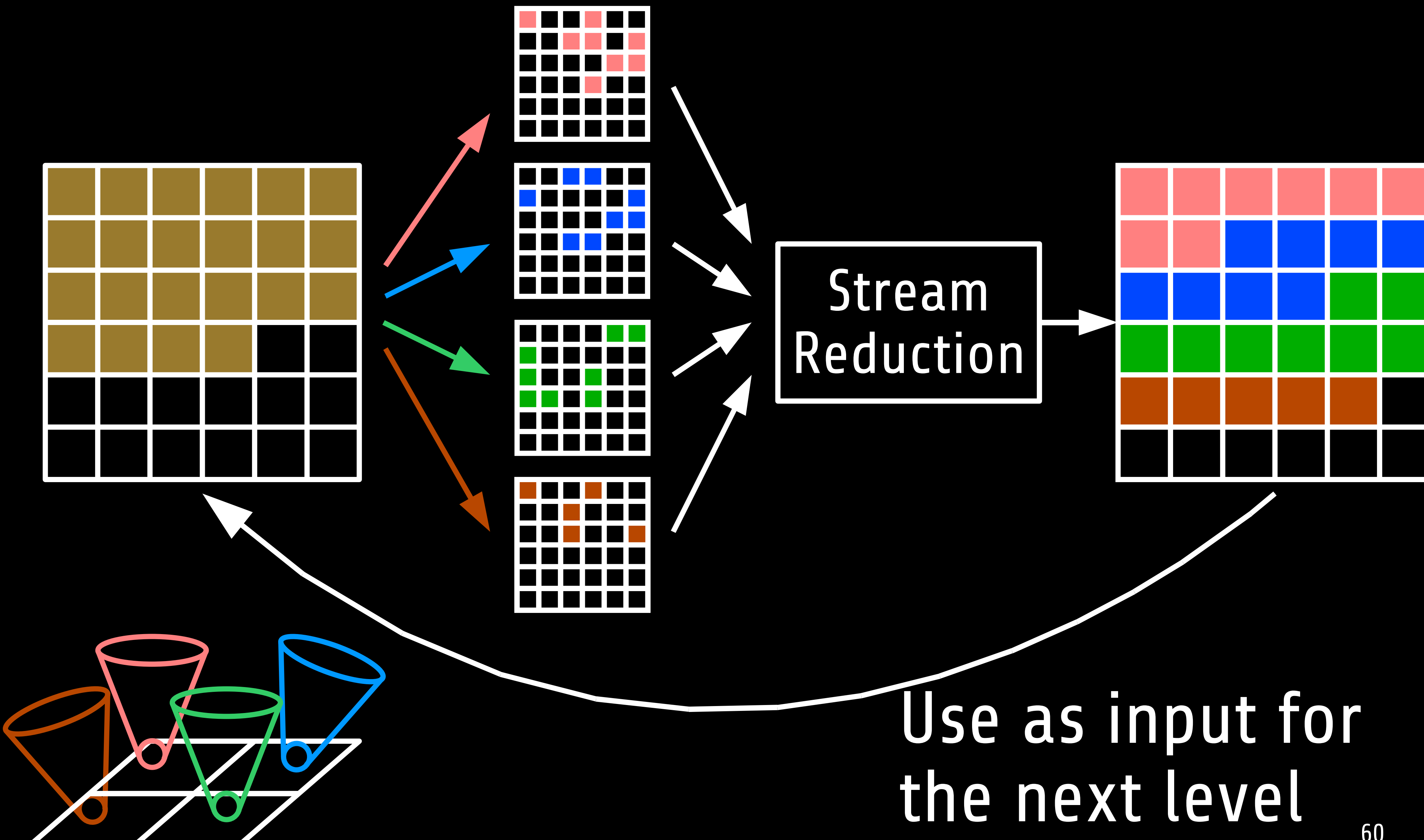
Stream Reduction

- Contribution of our work
- Faster
- Other applications
- See paper for details

GPU Traversal Implementation

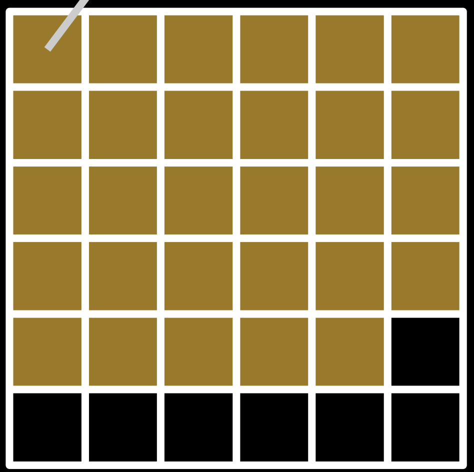


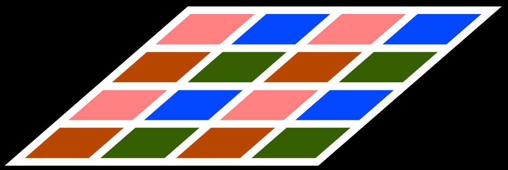
GPU Traversal Implementation



GPU Traversal Implementation

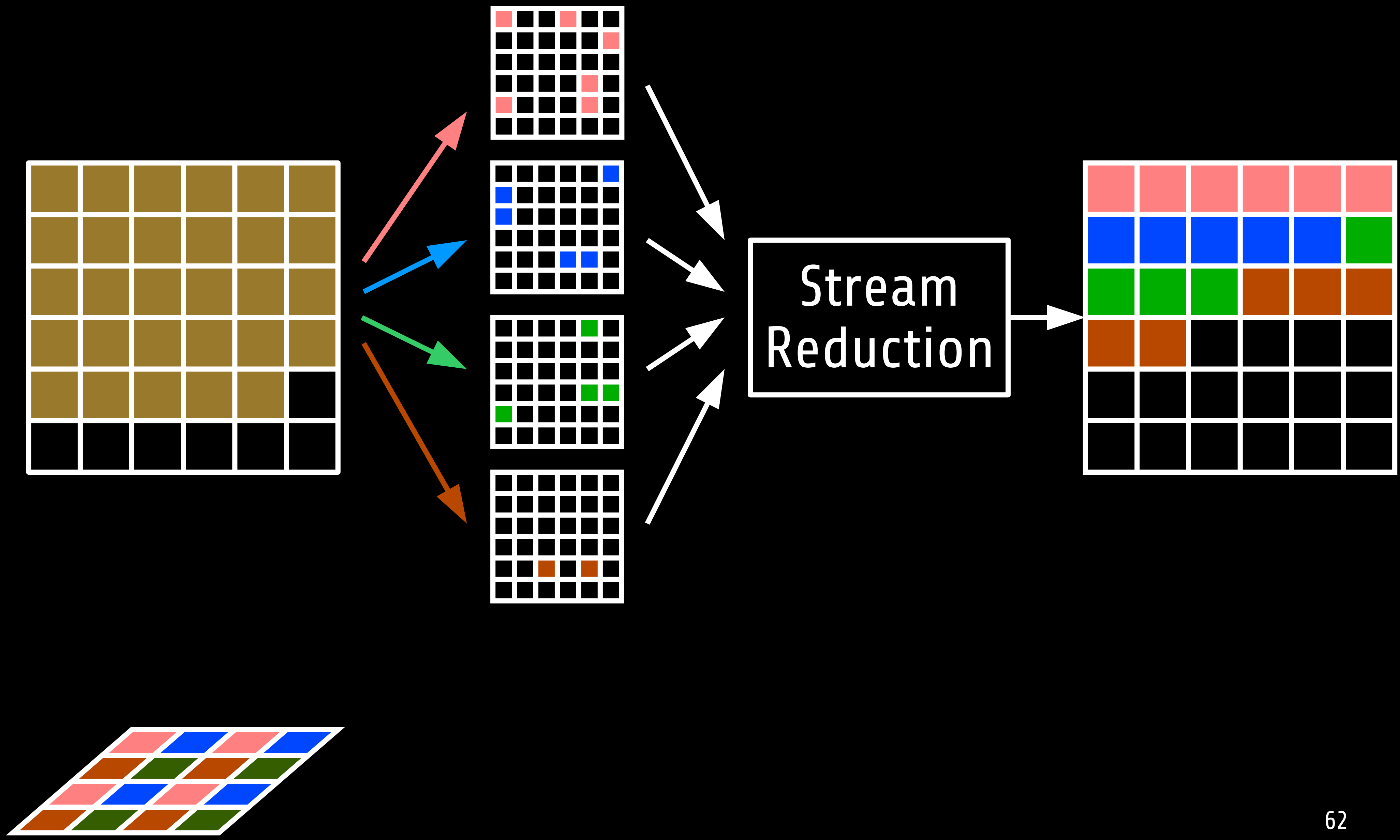
(TriID, NodeID)



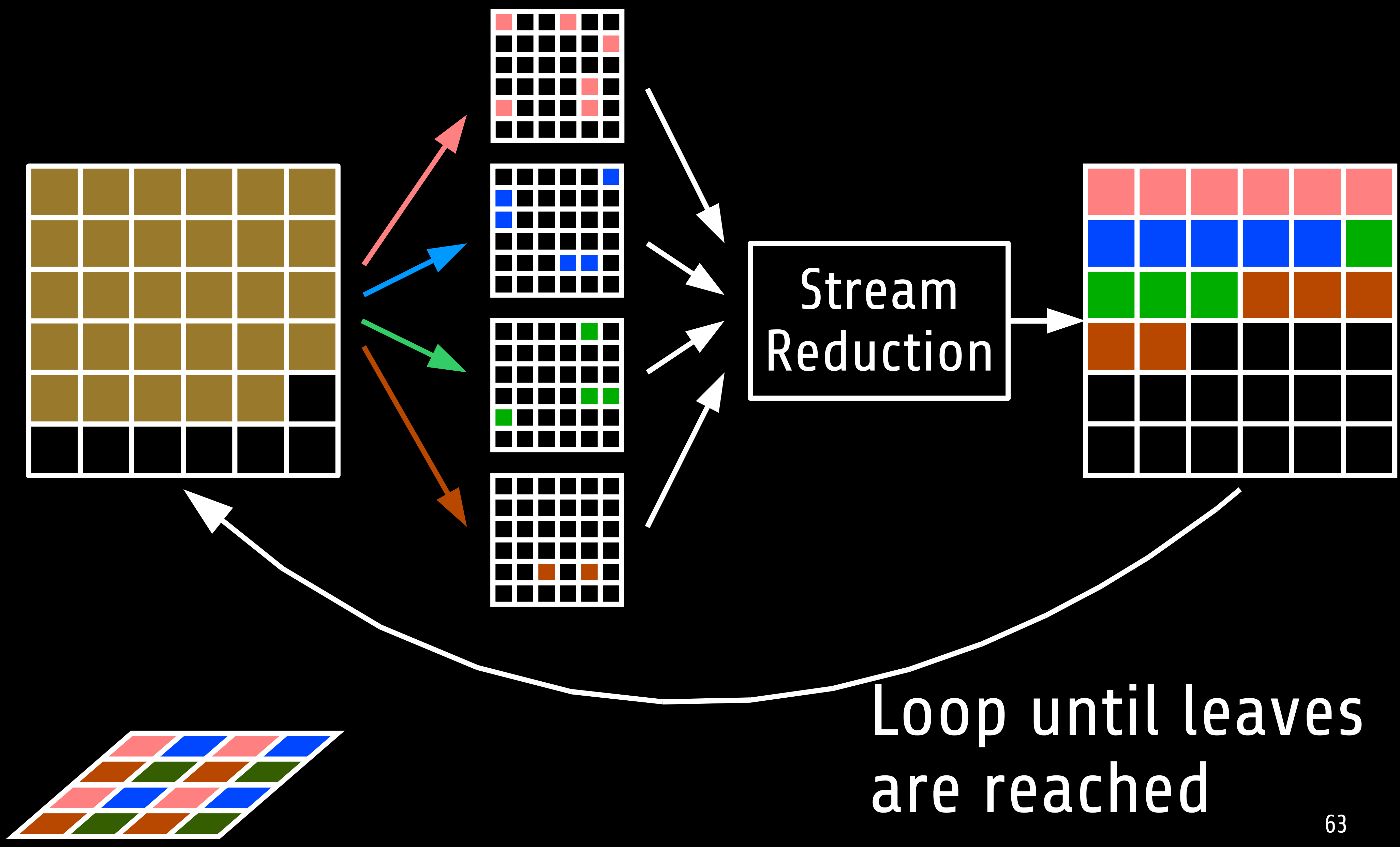


Next hierarchy level

GPU Traversal Implementation

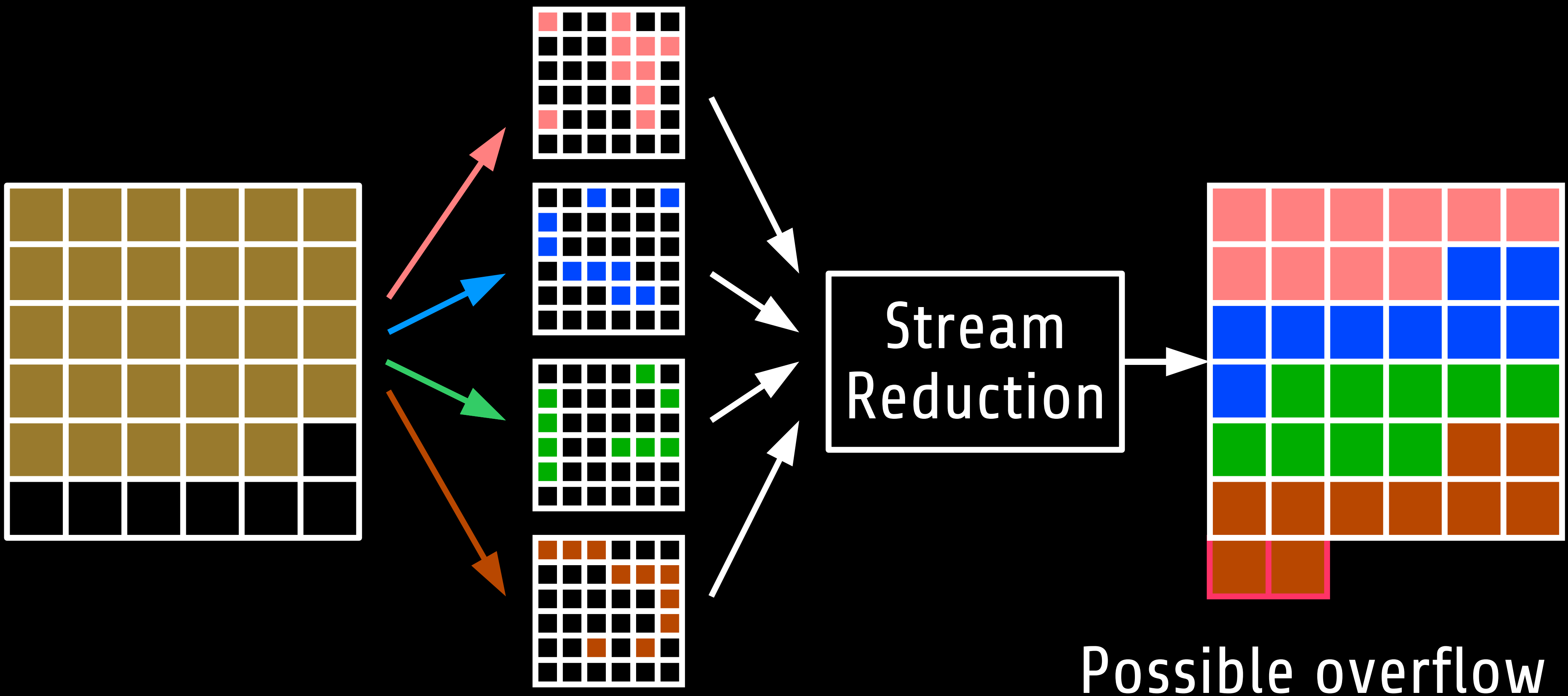


GPU Traversal Implementation



Loop until leaves are reached

Memory Considerations



Memory Considerations

- Simple workaround:
  - Subdivide scene in batches
  - Process the batches one after the other
  - Combine the results
- Constant overhead per batch (≈ 30ms)
- Allows large scene rendering too !
  - Even if it does not fit into GPU memory

Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Conclusion

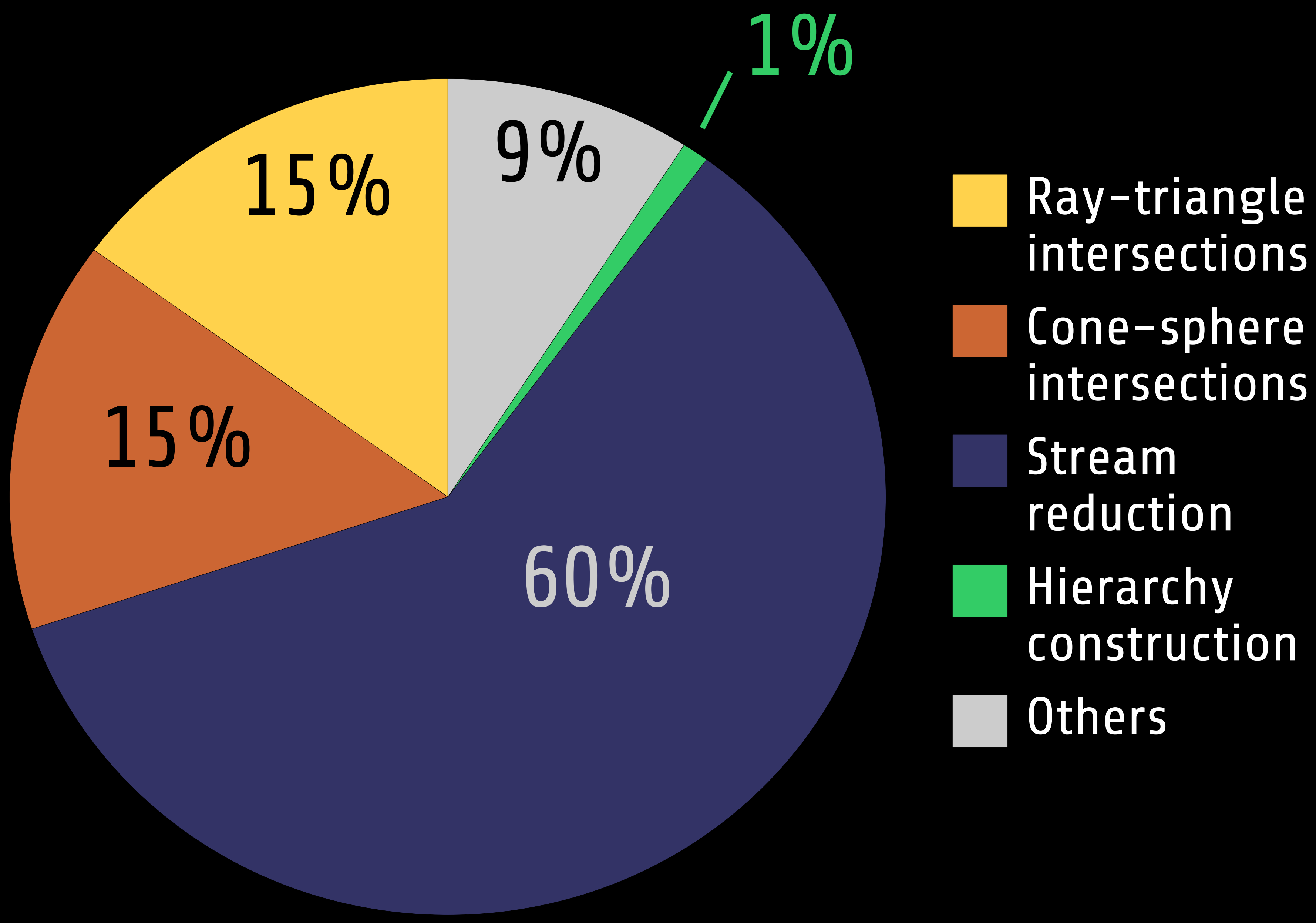
Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Conclusion

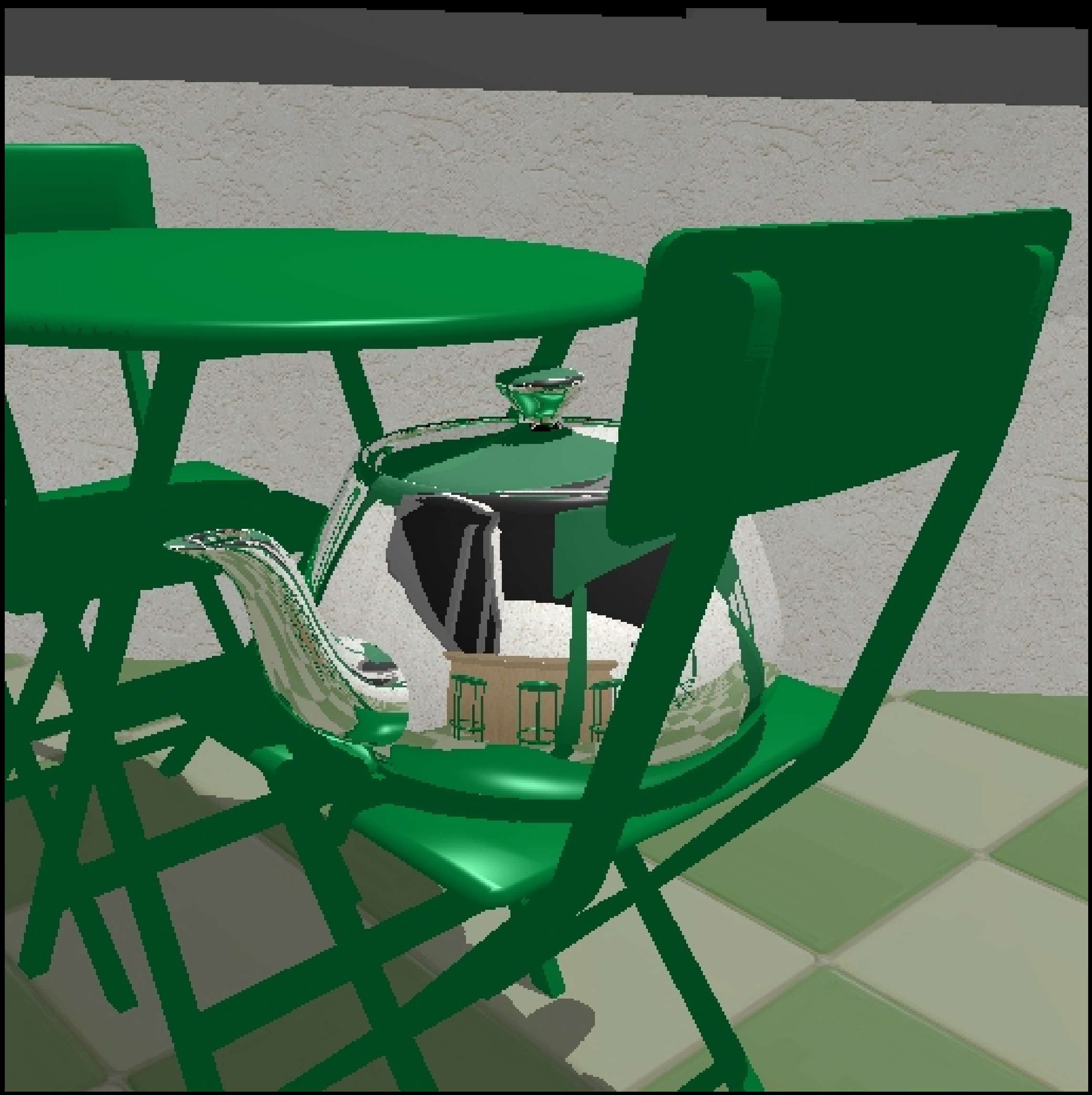
Hardware

- GPU: GeForce 8800 GTS
  - 640 Mo RAM
- CPU: Intel Pentium 4
  - 3 GHz
  - 2 Go RAM

Time Repartition

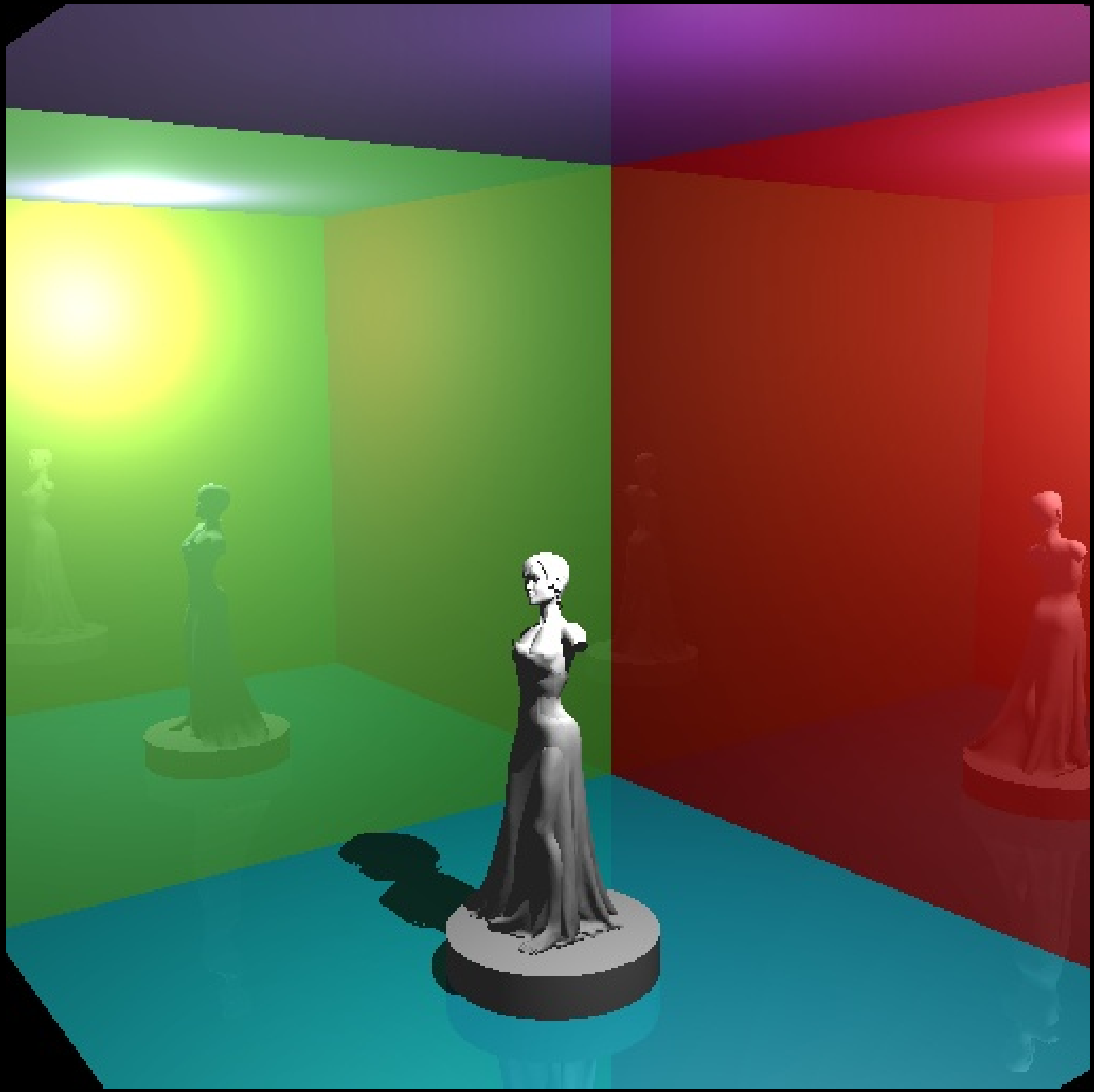


Results



79 ms
20 K Triangles
512 x 512

Results



Large spatial hierarchy

136 ms for 1 reflection
391 ms for 2 reflections
30 K Triangles
512 x 512

Results

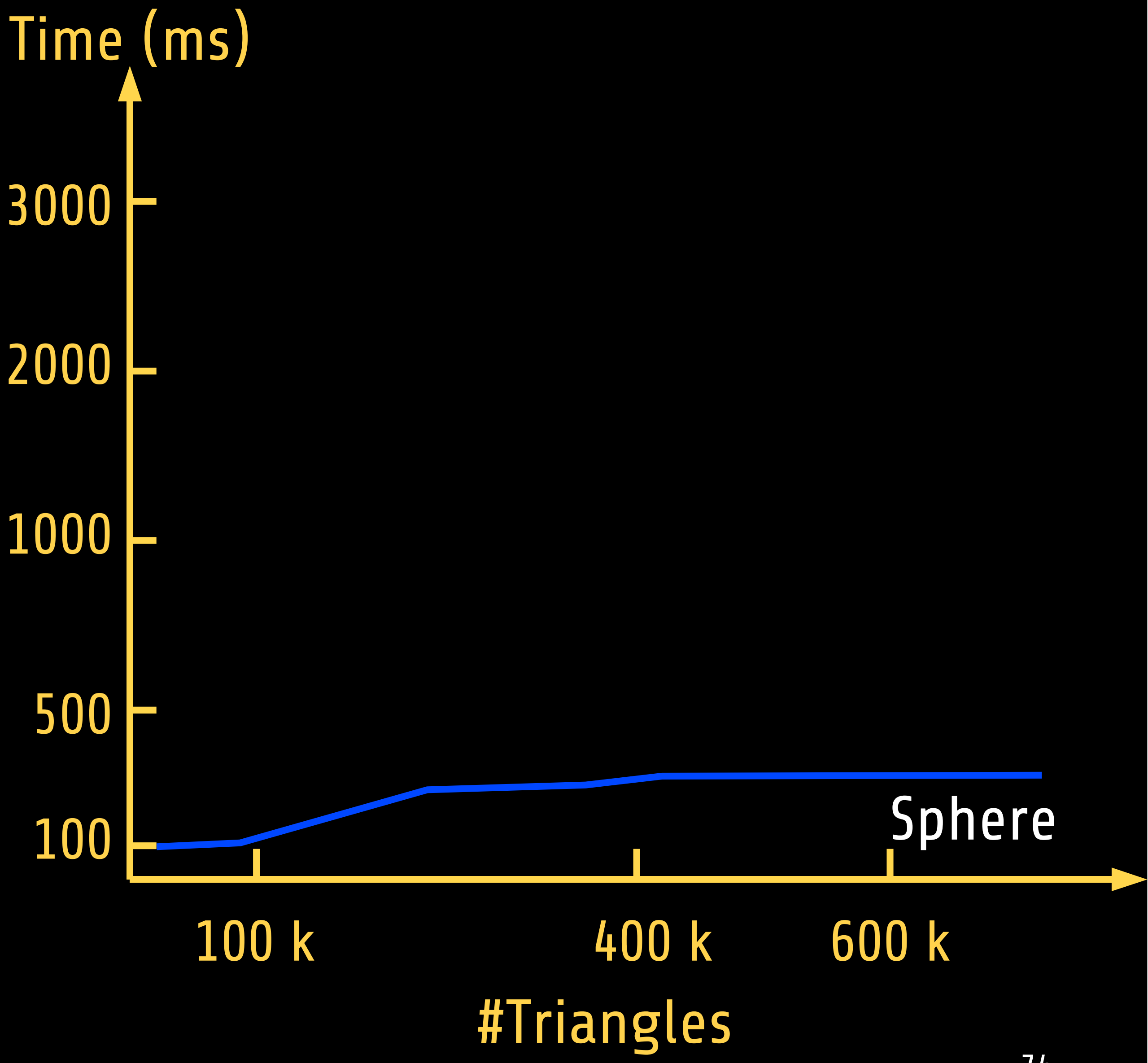
Several reflectors:
Discontinuities in the top levels of the hierarchy

302 ms
83 K Triangles
512 x 512

Results

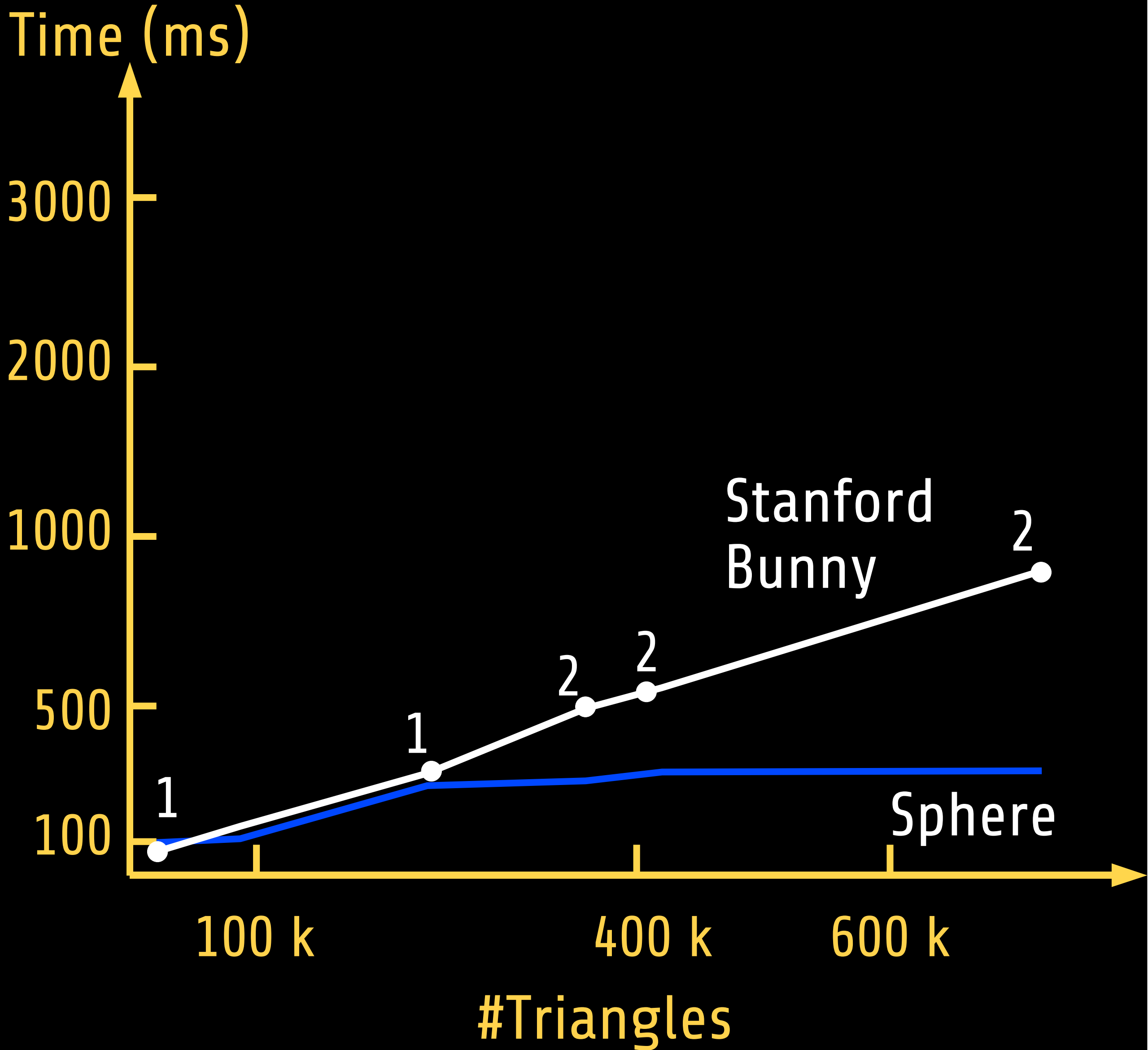
Video: Dynamic Scene

#Triangles

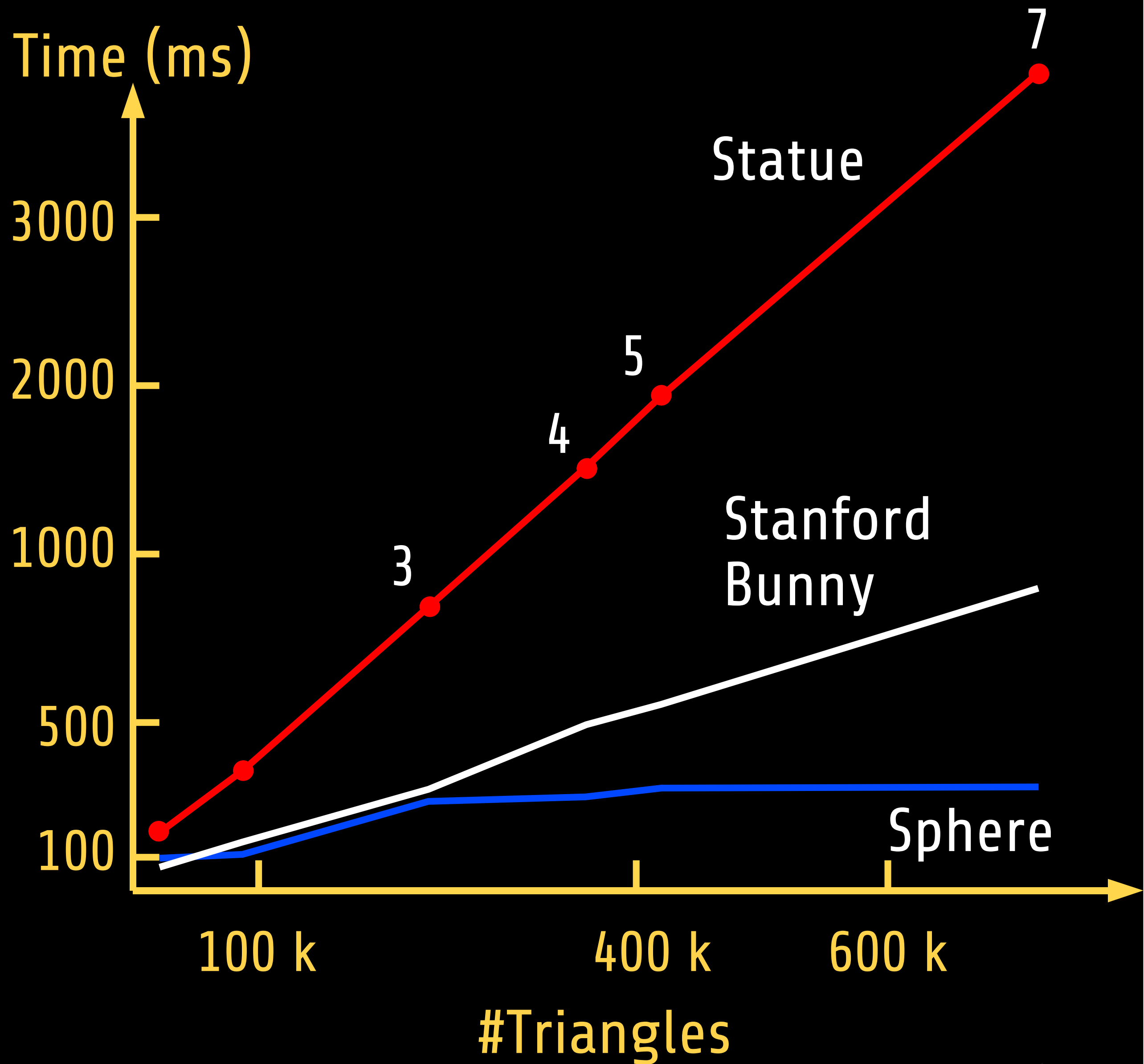


Time (ms)
3000
2000
1000
500
100
Sphere
100 k
400 k
600 k
#Triangles

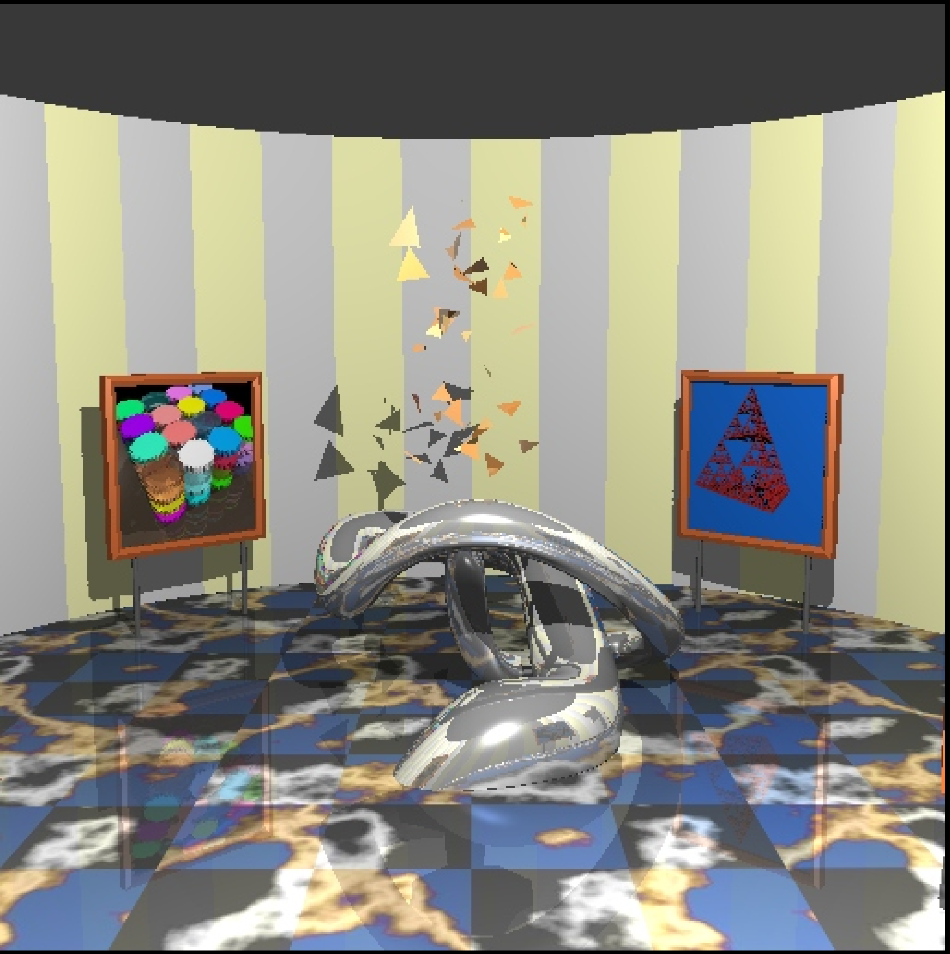
#Triangles


Time (ms)
3000
2000
1000
500
100
Stanford Bunny
Sphere
1
1
2
2
2
100 k
400 k
600 k
#Triangles

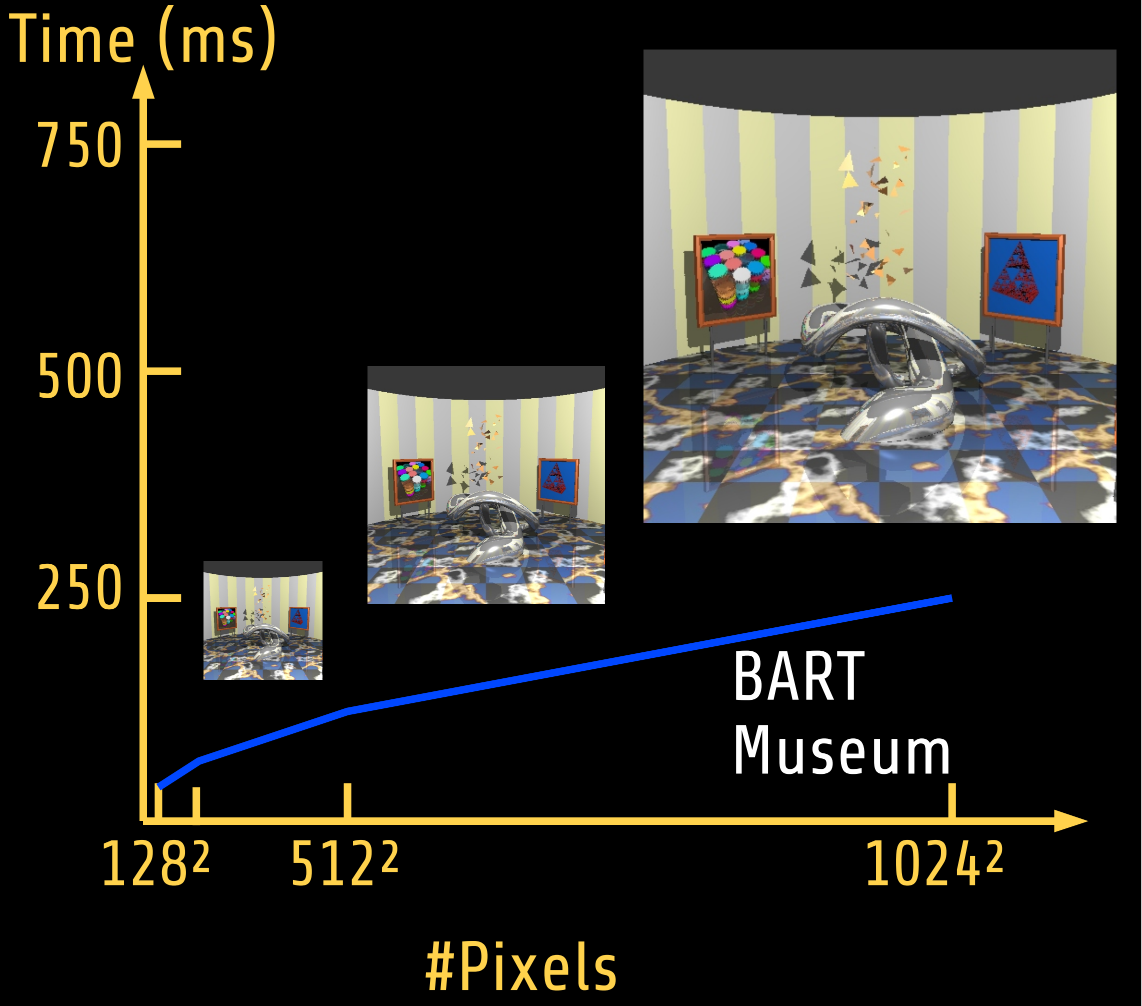
#Triangles


Time (ms)
3000
2000
1000
500
100
100 k
400 k
600 k
#Triangles
Statue
Stanford Bunny
Sphere
3
4
5
7

#Pixels

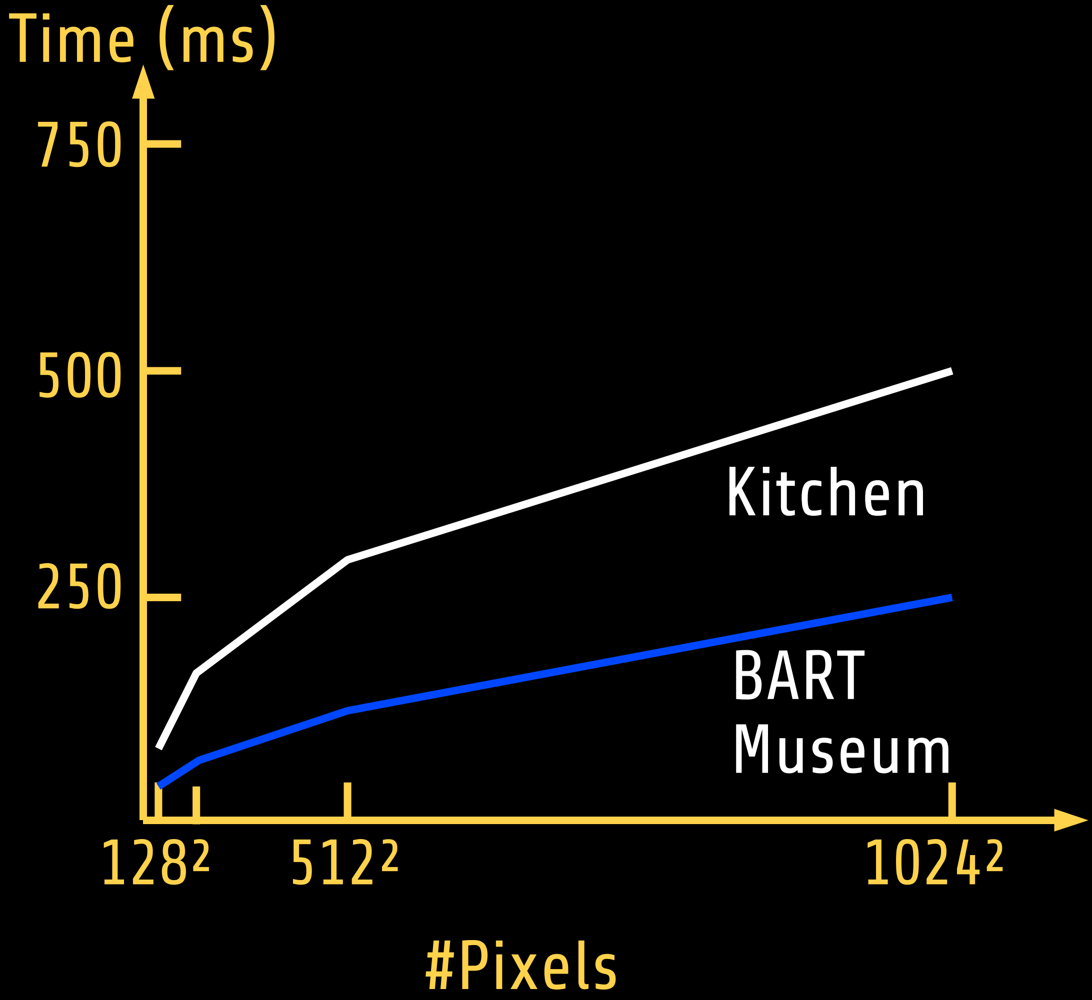


BART Museum
10 k Triangles



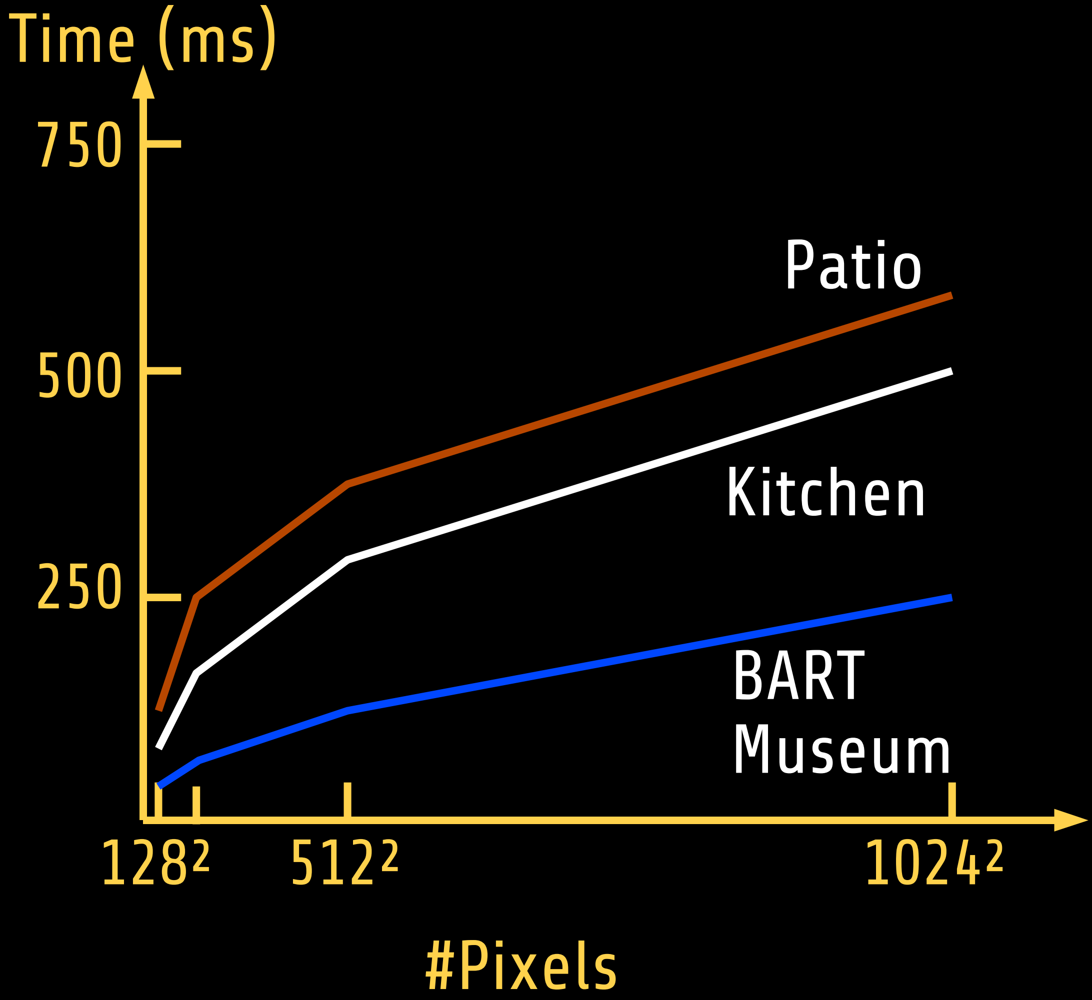
#Pixels

Kitchen
83 k Triangles



#Pixels

Patio
87 k Triangles



Large Scene

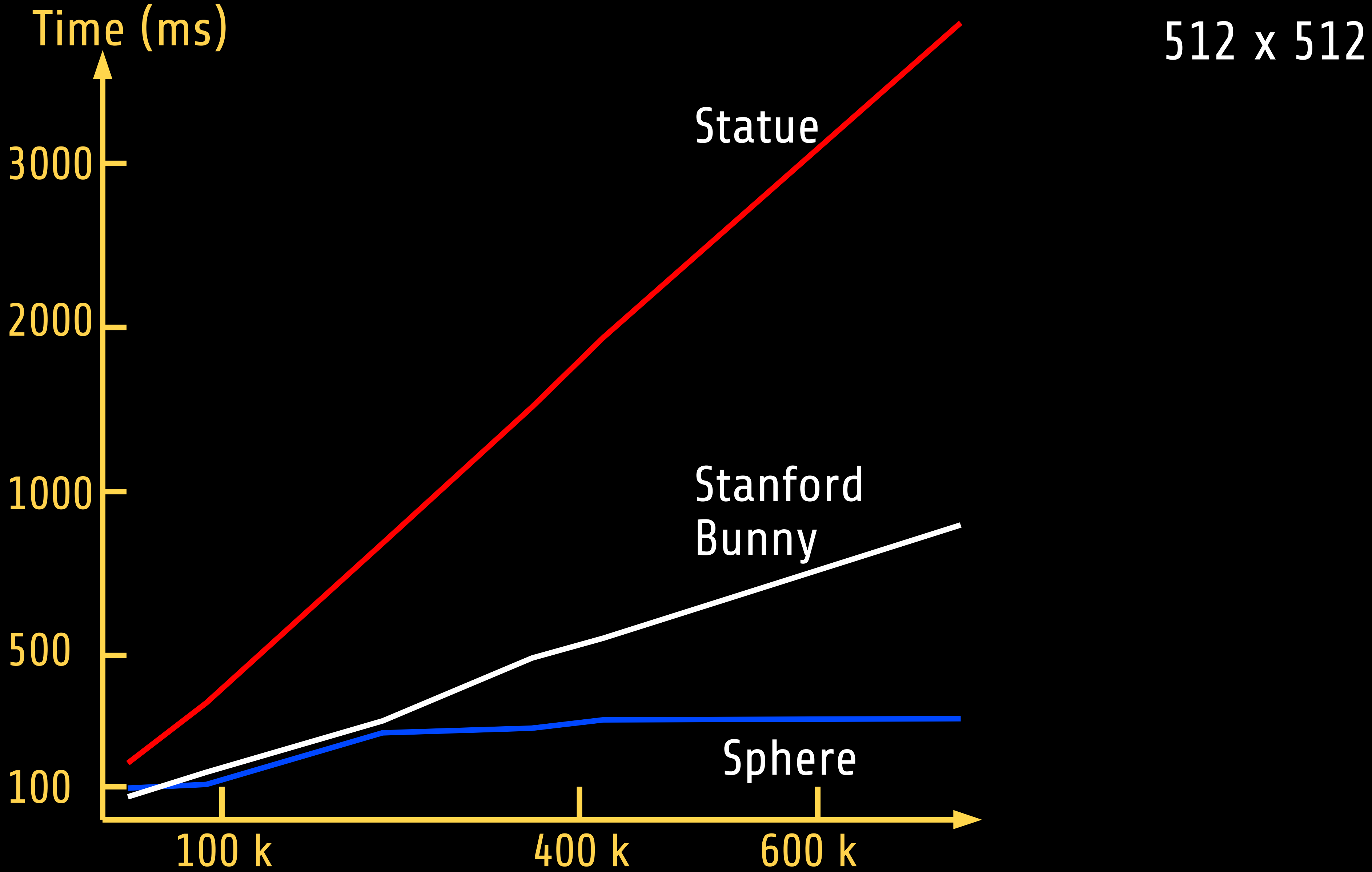
50 batches

18.5 s
2.3 M Triangles
512 x 512

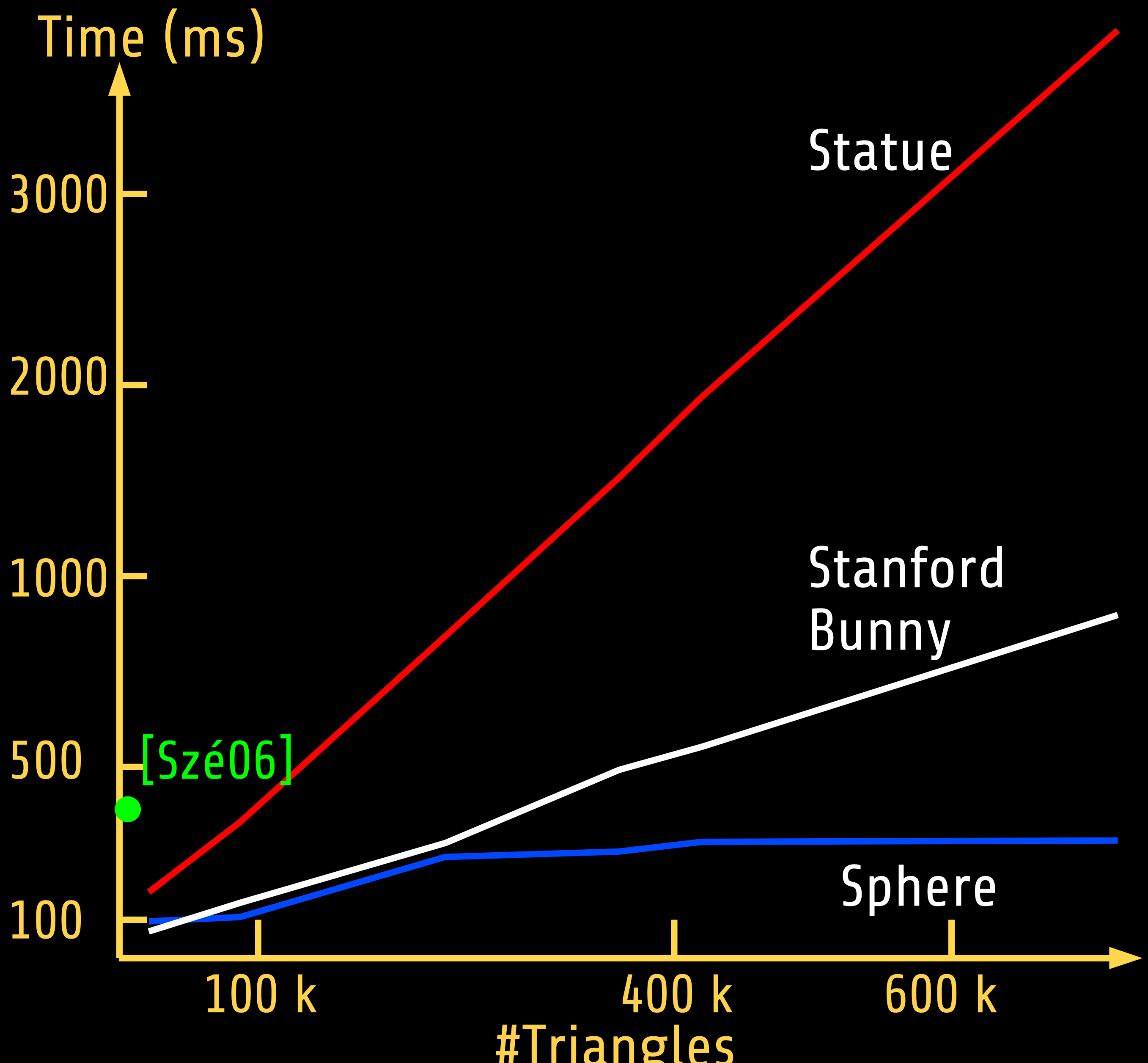
Comparison with Previous Works

- Very few papers report figures for both:
  - Dynamic scenes
  - Secondary rays
- We compared to papers that do at least one
- Hard to compare

Comparison with Previous Works



Comparison with Previous Works

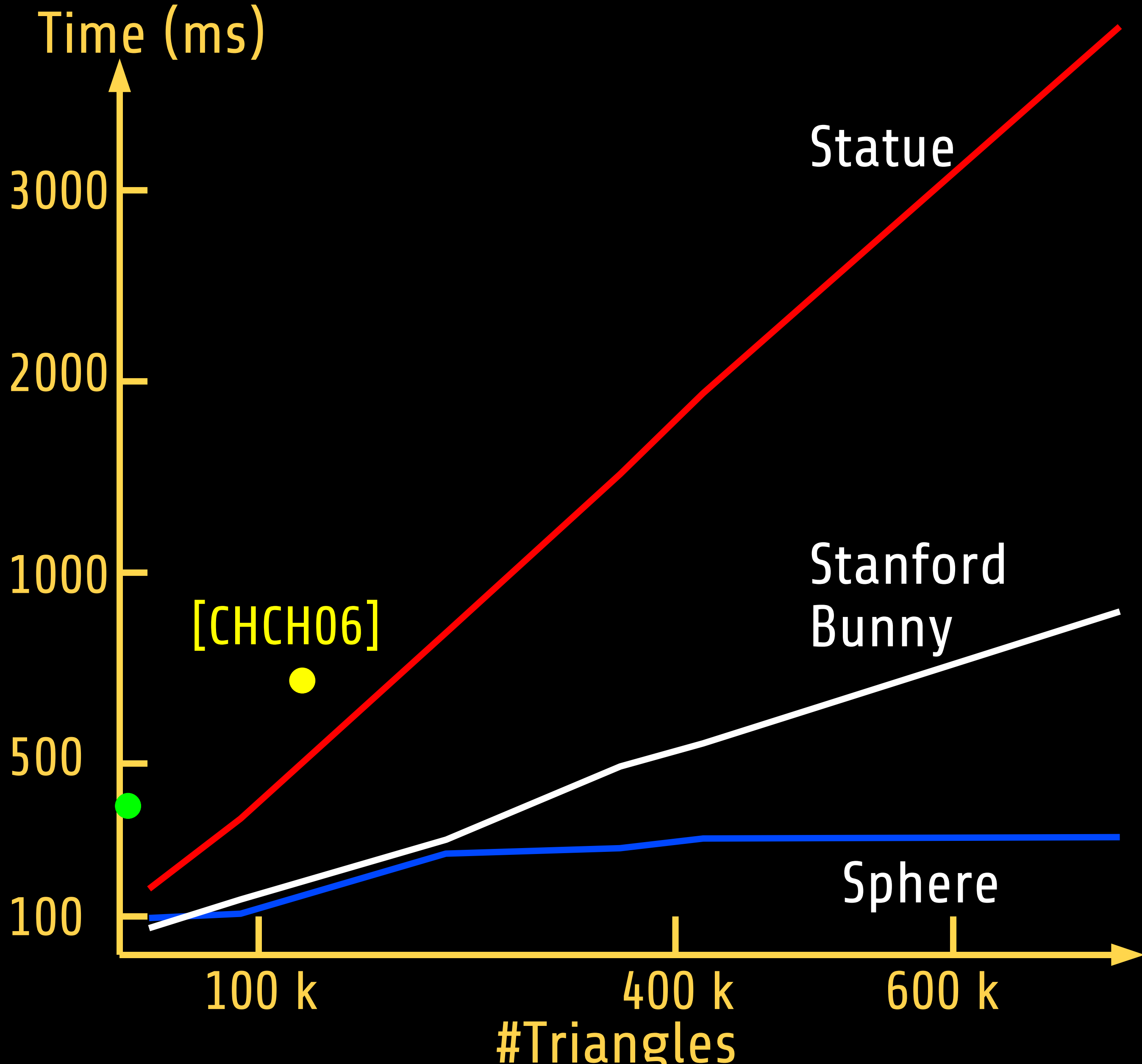


512 x 512

[Szé06] Szécsi

- 2 levels ray hierarchy
- GPU (GeForce 6800 GT)

Comparison with Previous Works

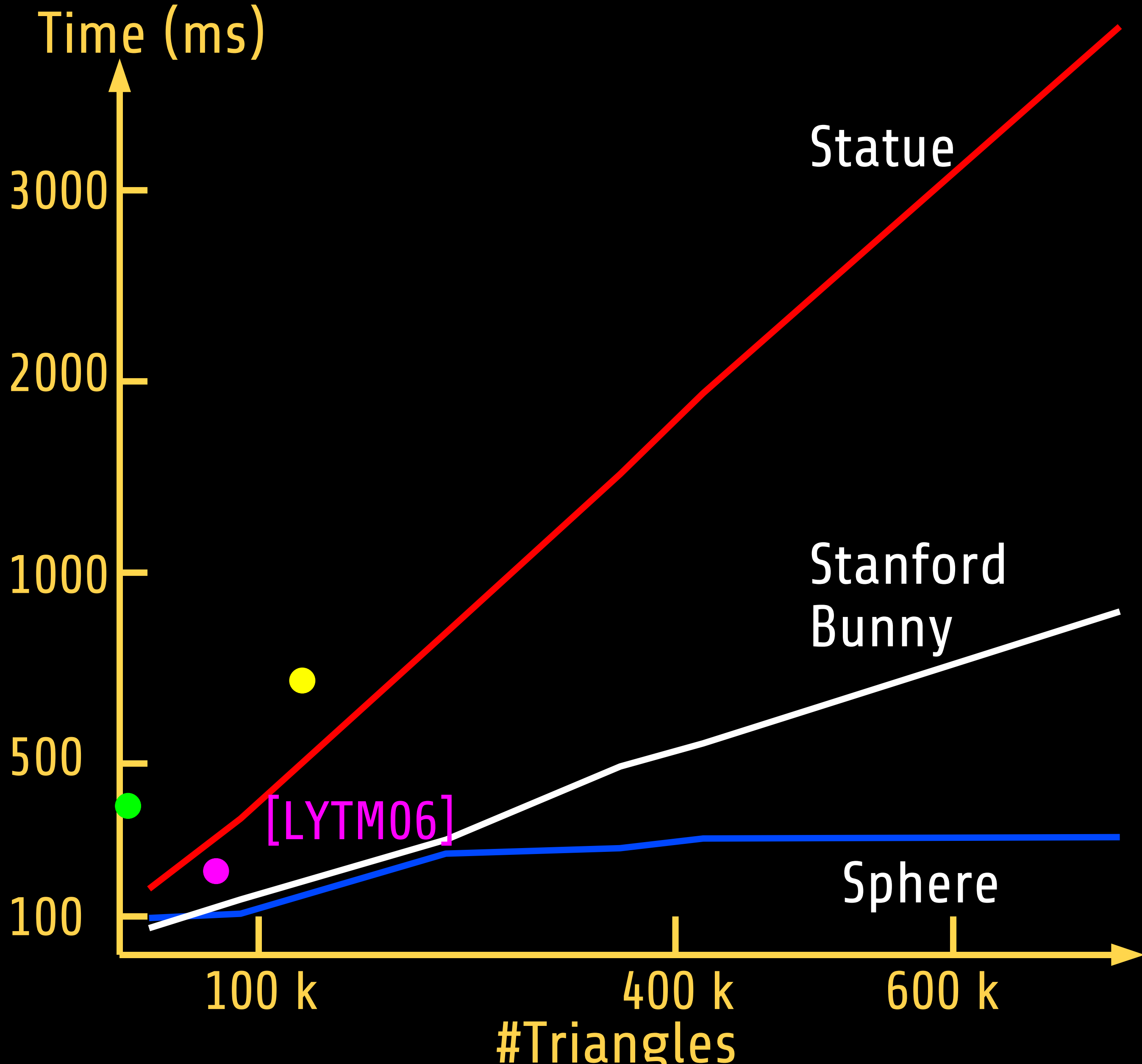


512 x 512

[CHCH06] Carr *et al.*

- Dynamic scene
- Primary rays
- Geometry images
- GPU (GeForce 6800 Ultra)

Comparison with Previous Works

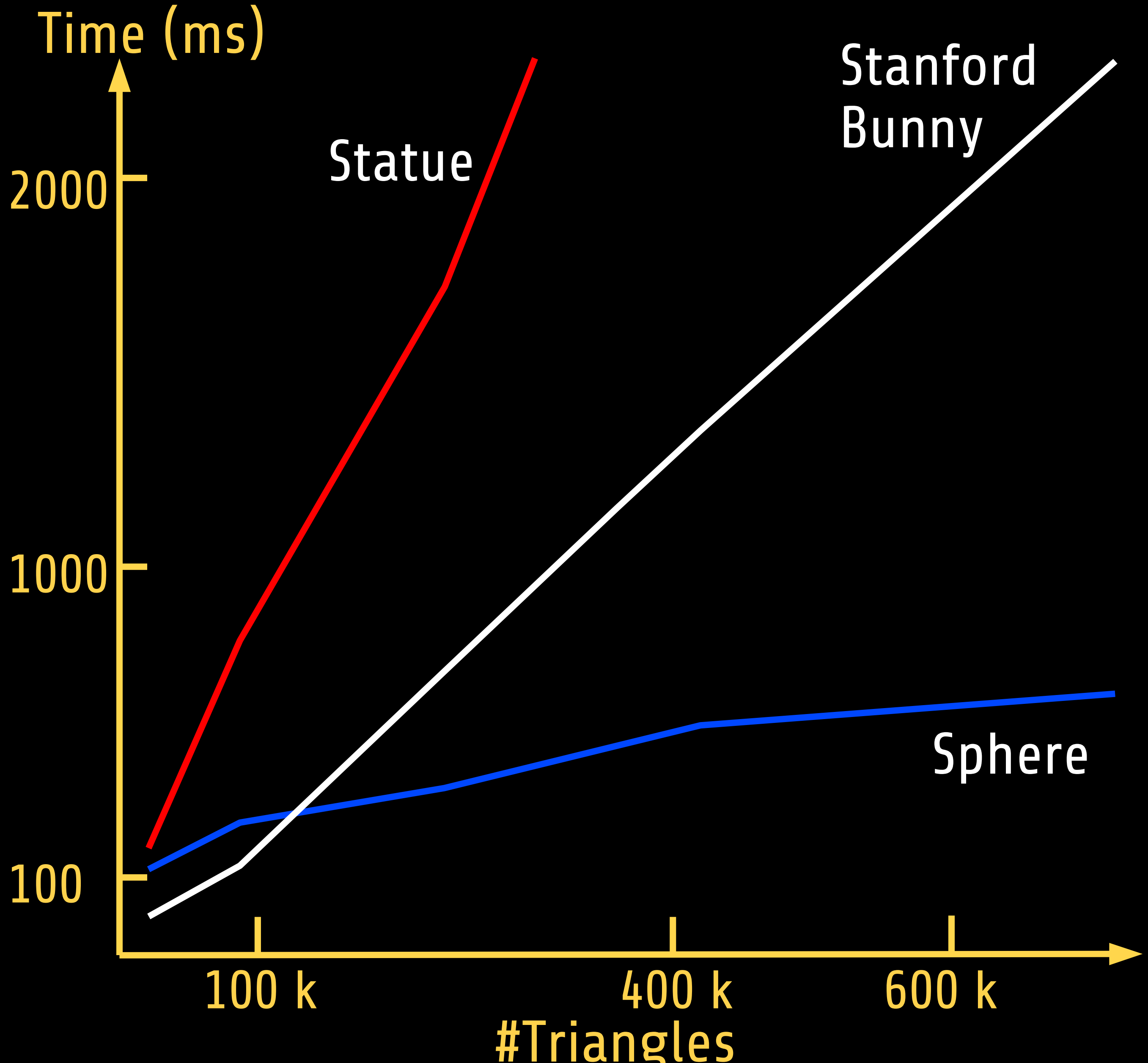


512 x 512

[LYTM06] Lauterbach *et al.*

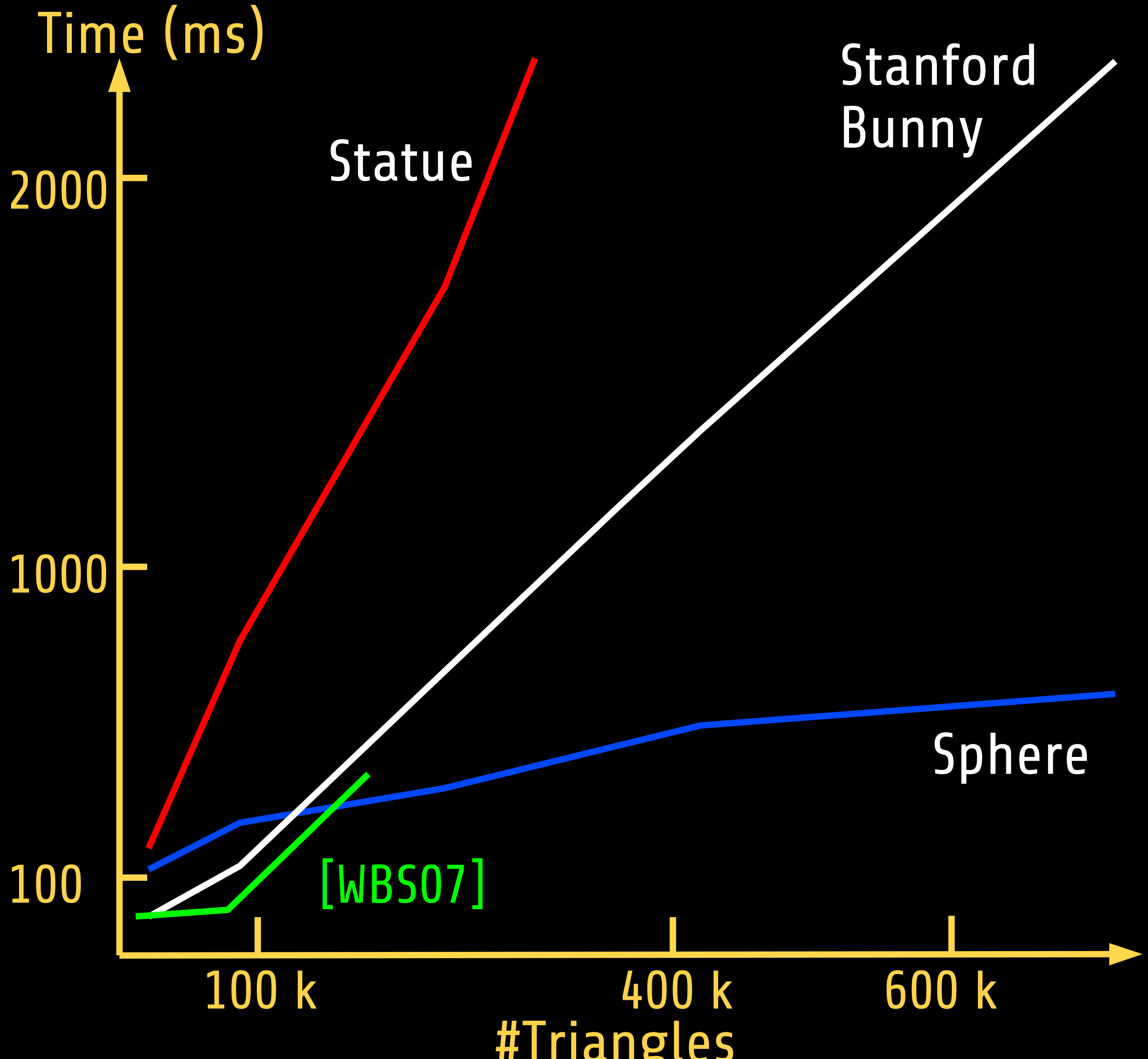
- Dynamic scene
- 1 Reflection
- Bounding Volume Hierarchy
- CPU

Comparison with Previous Works



1024 x 1024

Comparison with Previous Works

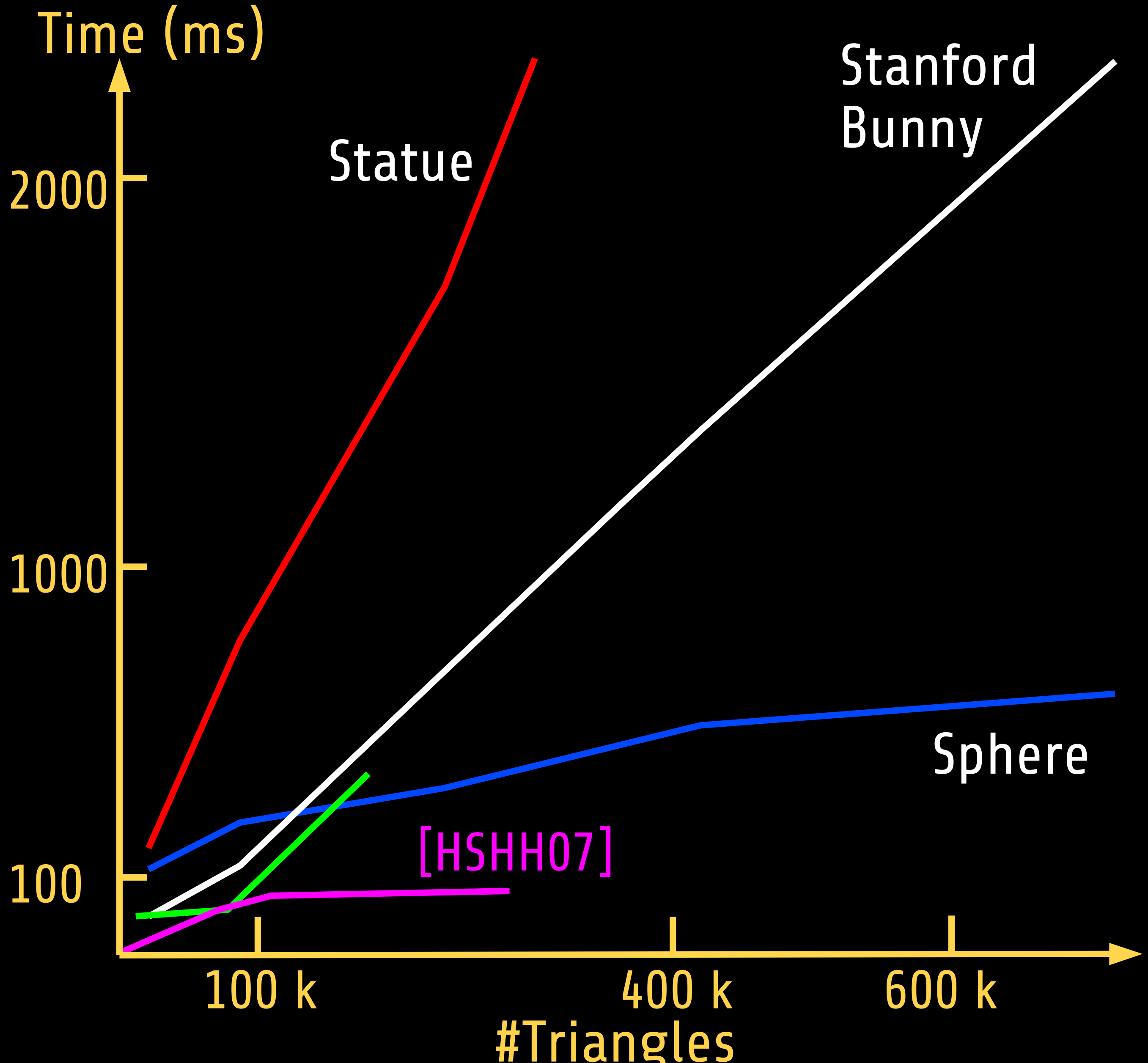


1024 x 1024

[WBS07] Wald *et al.*

- Dynamic scene
- Primary rays
- Bounding Volume Hierarchy
- CPU

Comparison with Previous Works



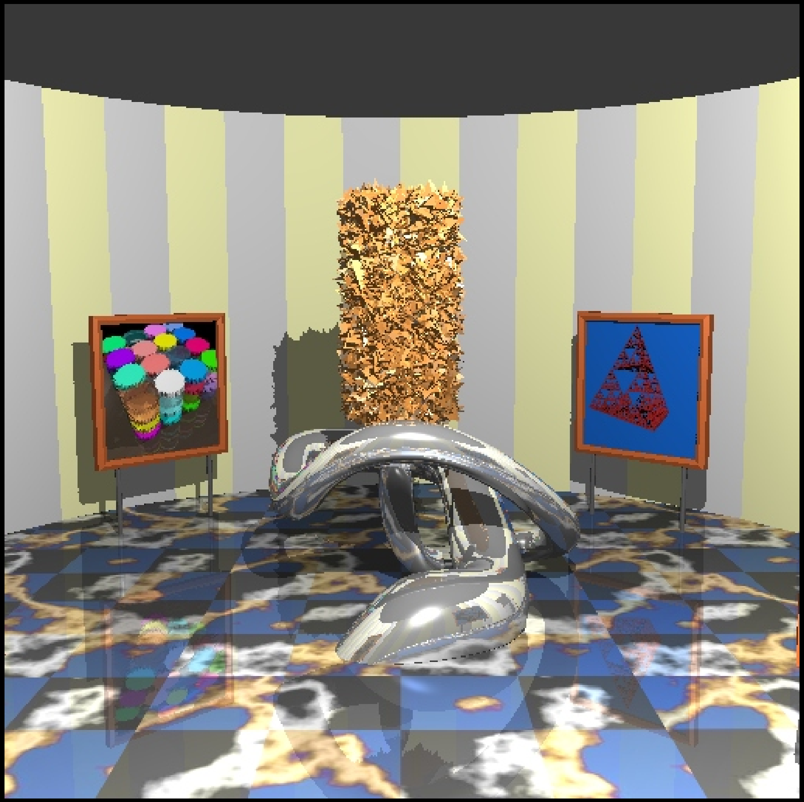
1024 x 1024

[HSHH07] Horn *et al.*

- Static scene
- 1 Reflection
- Kd tree
- GPU (ATI X1900 XTX)

Comparison with Previous Works

- [WK06] Wächter and Keller, Bounding Interval Hierarchy, CPU
  - Dynamic scene + secondary rays



Reflection + shadow

	Us	[WK06]
Museum3 10 k Tris	289 ms	1282 ms
Museum8 76 k Tris	3330 ms	2040 ms

Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Conclusion

Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Conclusion

Pros and Cons

- Pros
  - Dynamic scenes
  - No precomputation
  - Scales well with resolution
  - Large scenes support
- Cons
  - No early ray termination
  - Linear in #Triangles
  - Ray hierarchy looser than scene hierarchy (ray coherency)

Contributions

- Interactive ray tracing algorithm
  - Secondary rays
  - Dynamic scenes
  - No precomputation
  - Scales well with resolution
  - Large scenes support
- Faster stream reduction

Future Works

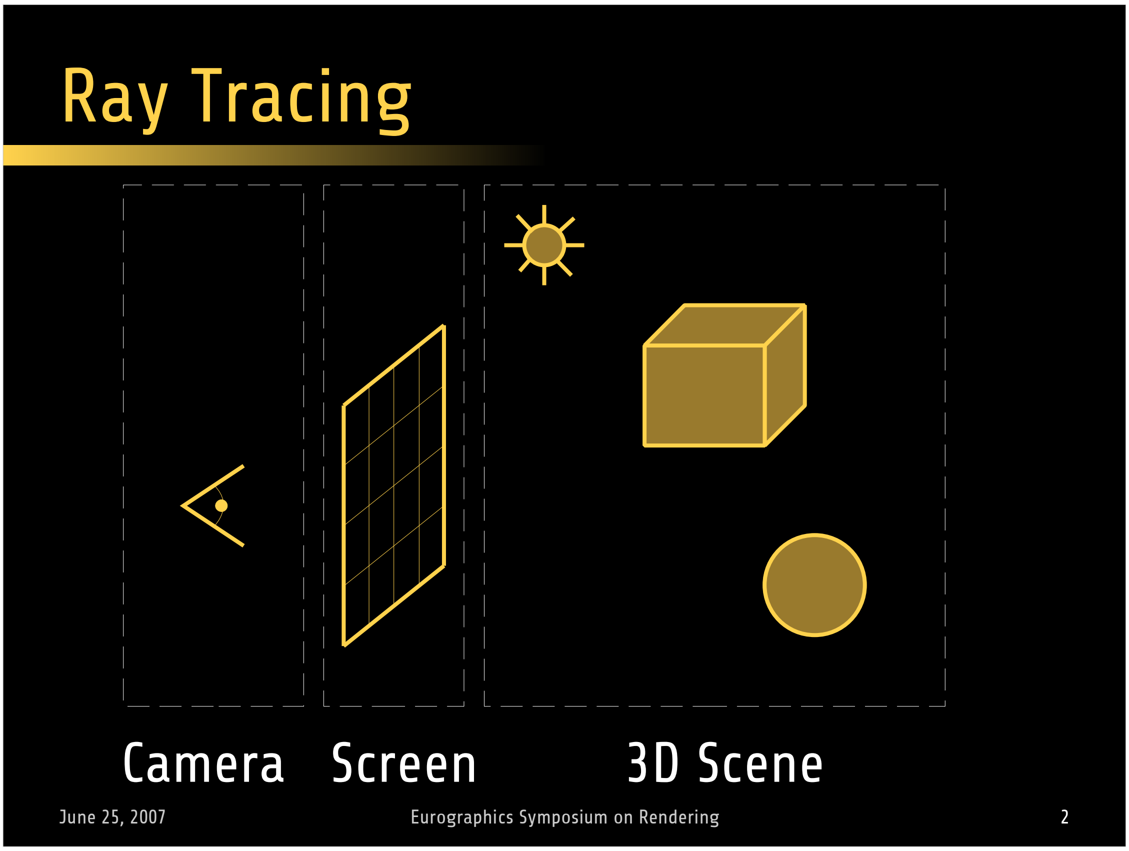
- Cone tracing:
  - anti aliasing
  - soft shadows
  - glossy reflections
- Scene Structure
  - Regular grid, bounding volumes
  - More complex structure ? Hierarchy ?

Thank You

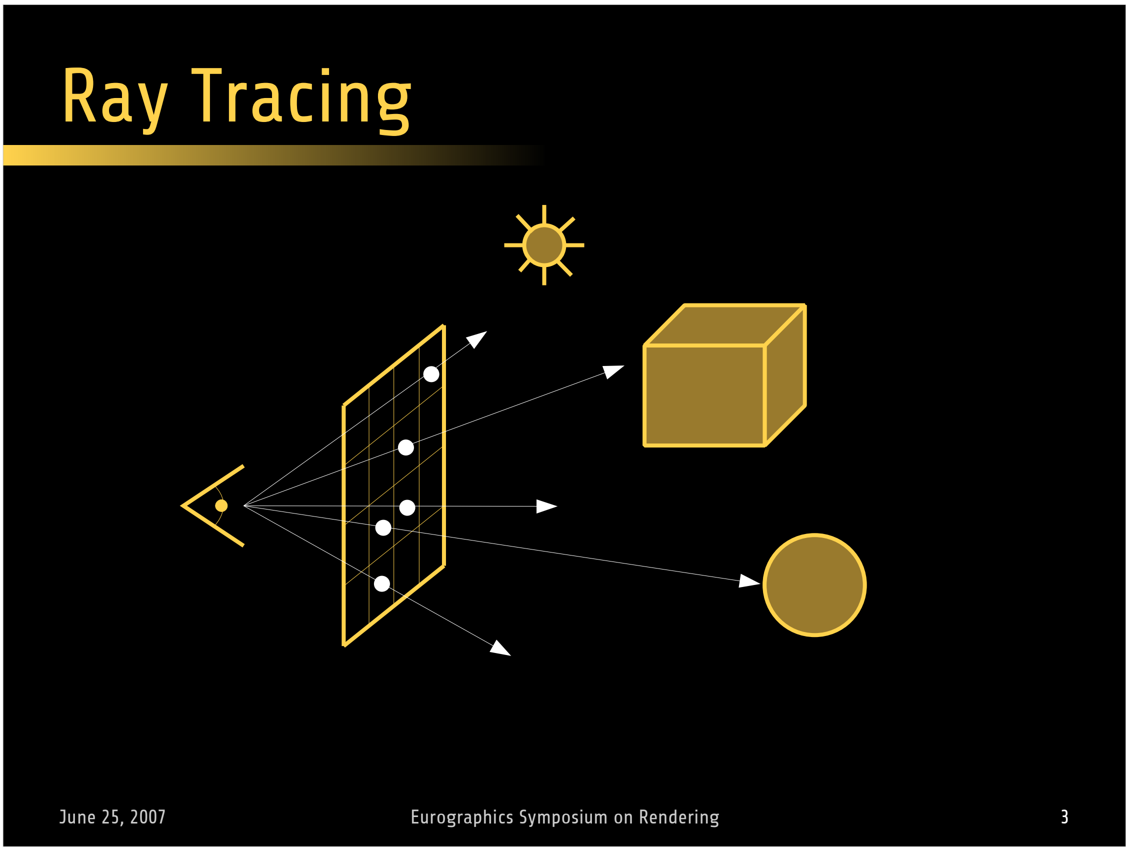
Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU

David Roger, Ulf Assarsson, Nicolas Holzschuch

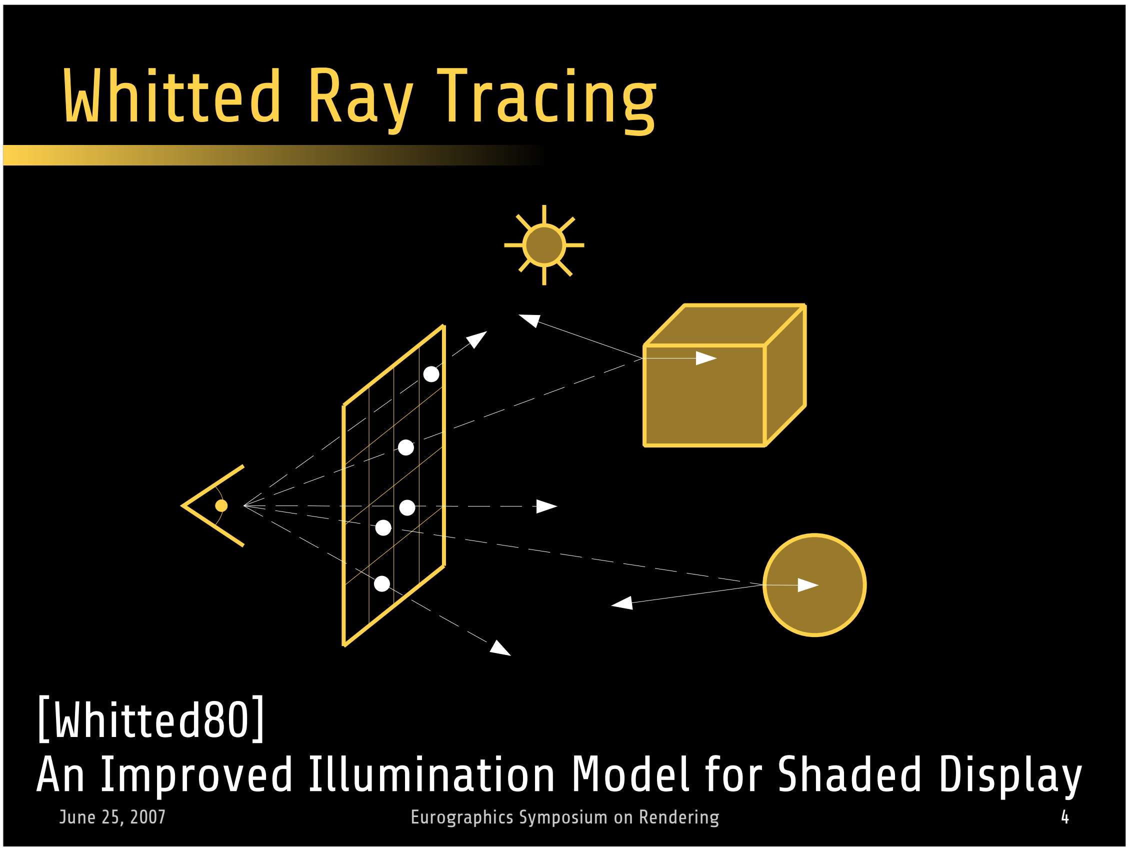
Grenoble University
Chalmers University of Technology



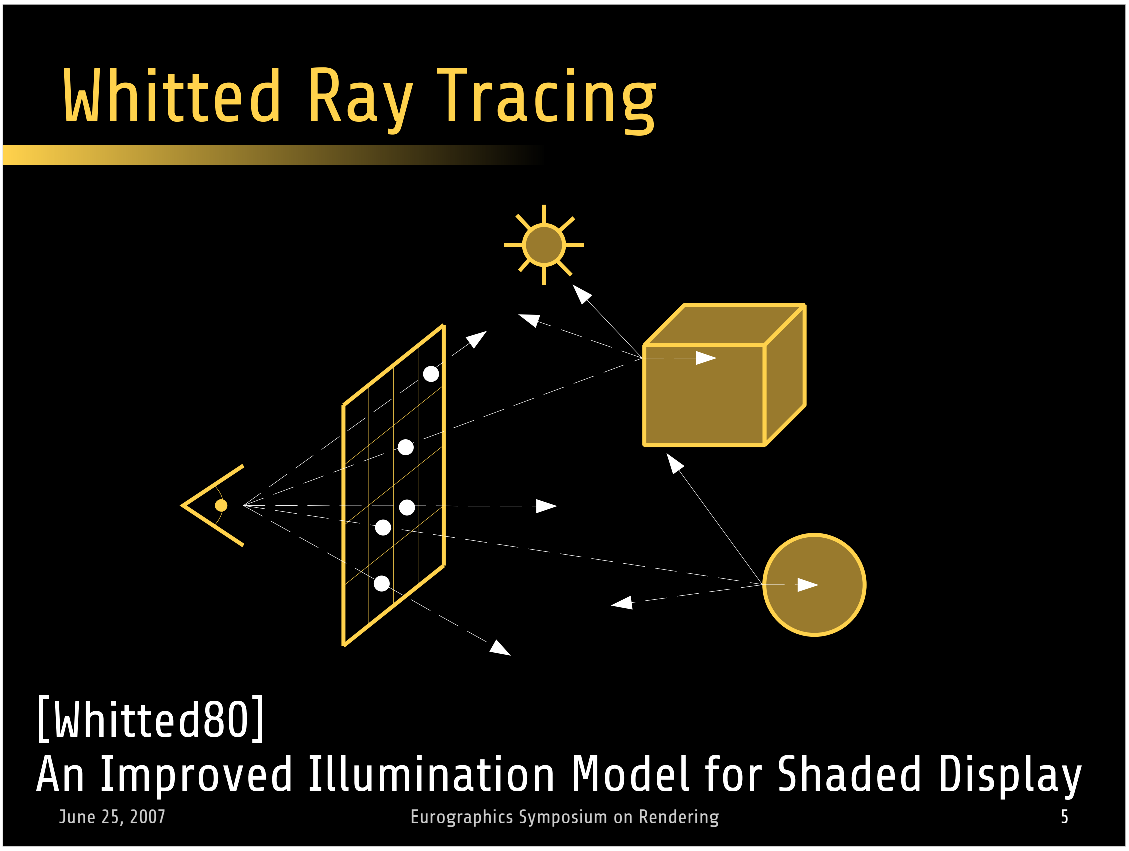
Ray tracing is a rendering technique



Consisiting in shooting one or several rays through each pixels of the screen and intersecting them with the scene

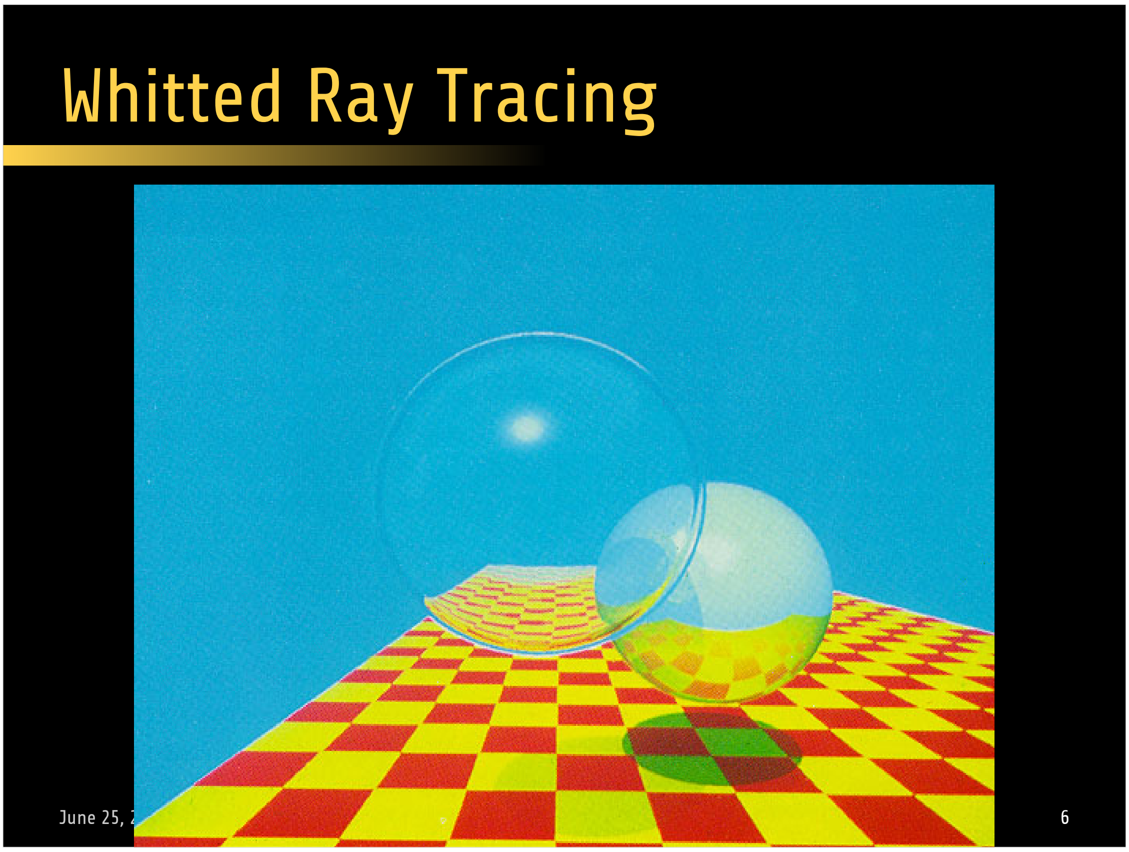


Turner Whitted designed a shading model relying on raytracing.
It renders specular effects (reflections and refractions) by shooting additional rays

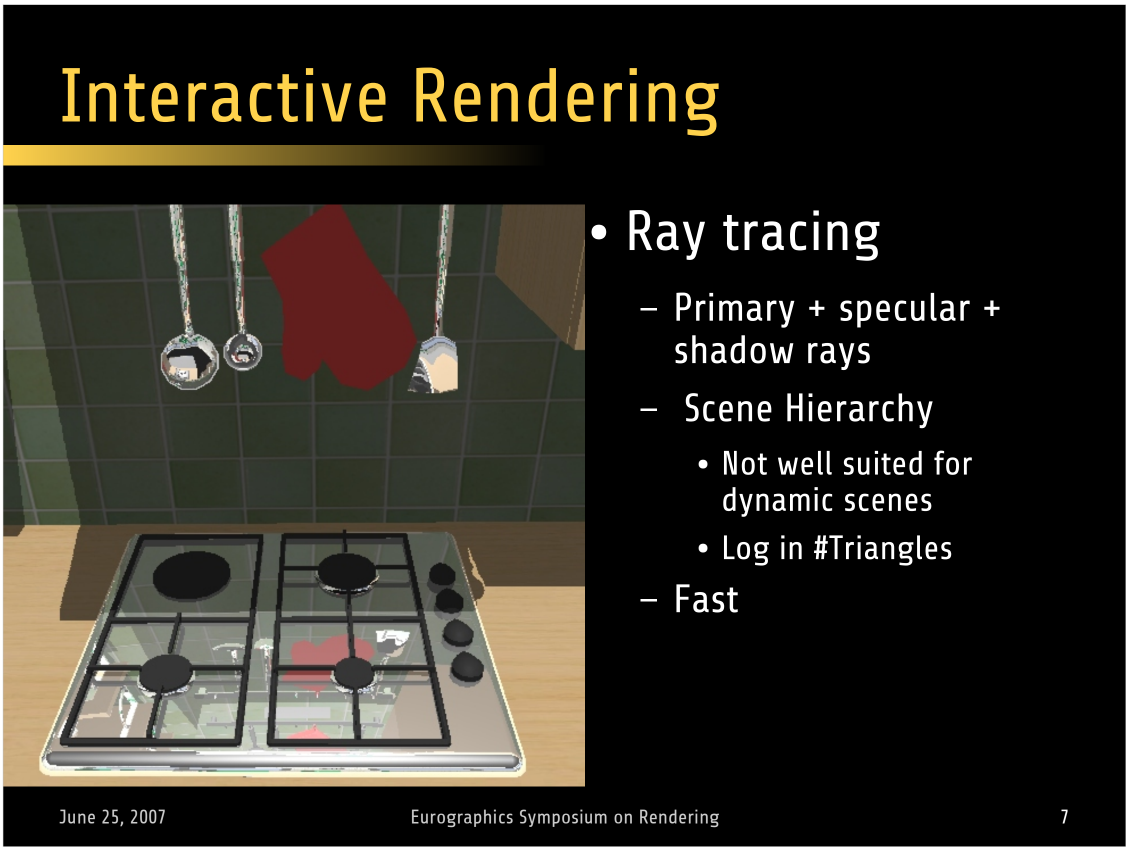


As well as shadows by shooting rays from the objects toward the light source.

All those rays originating from the objects are called secondary rays.



This is an example of Whitted ray tracing with reflections, refractions and shadows.

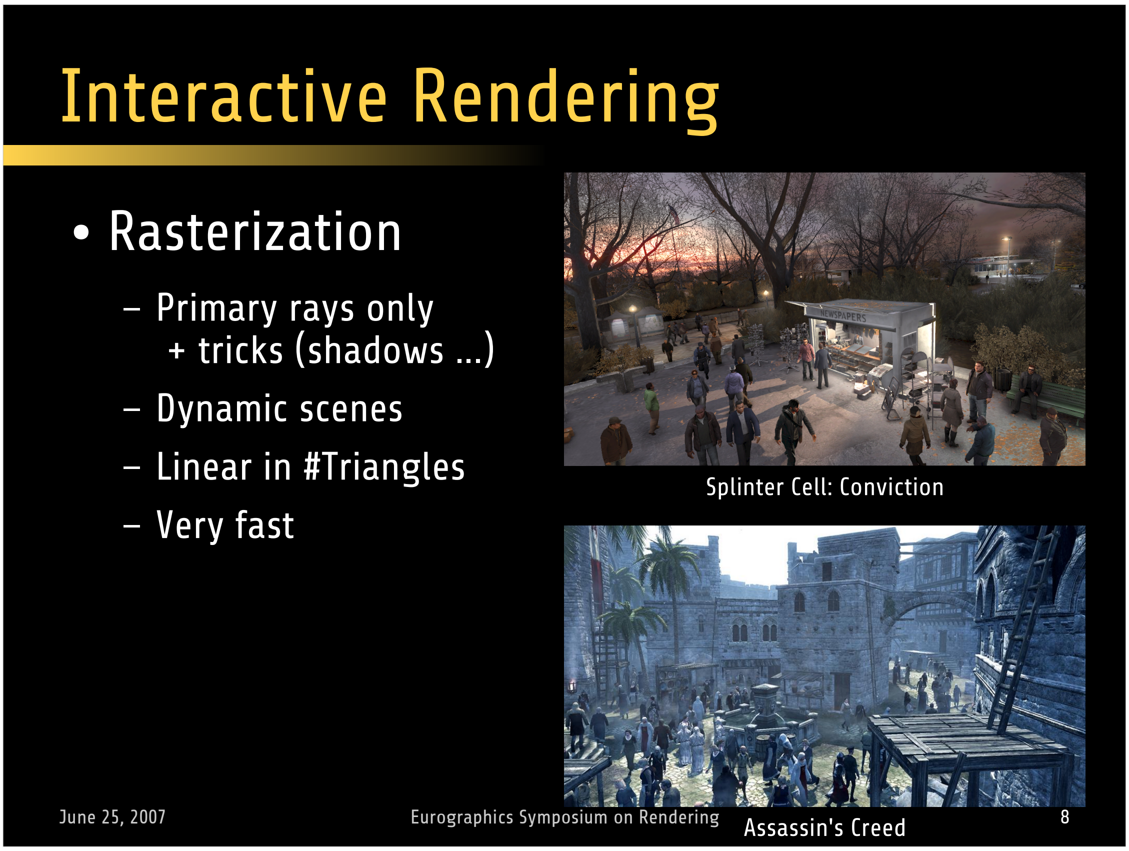


Nowadays, ray tracing can definitely be used for interactive rendering.

It leads to high quality pictures, because it is able to render specular effects like refraction, or reflection as you can see on the left.

Most interactive ray tracers use structures on the scene to speed up their computations, particularly hierarchies.

They manage to get logarithmic complexity, but they have troubles with dynamic scenes, because those structures have to be updated or recomputed as the scene changes.



The most commonly used technique for interactive rendering is rasterization, for example in current games, as you can see on the right.

It is designed for primary rays, although some tricks or approximations can be used for other effects like shadows.

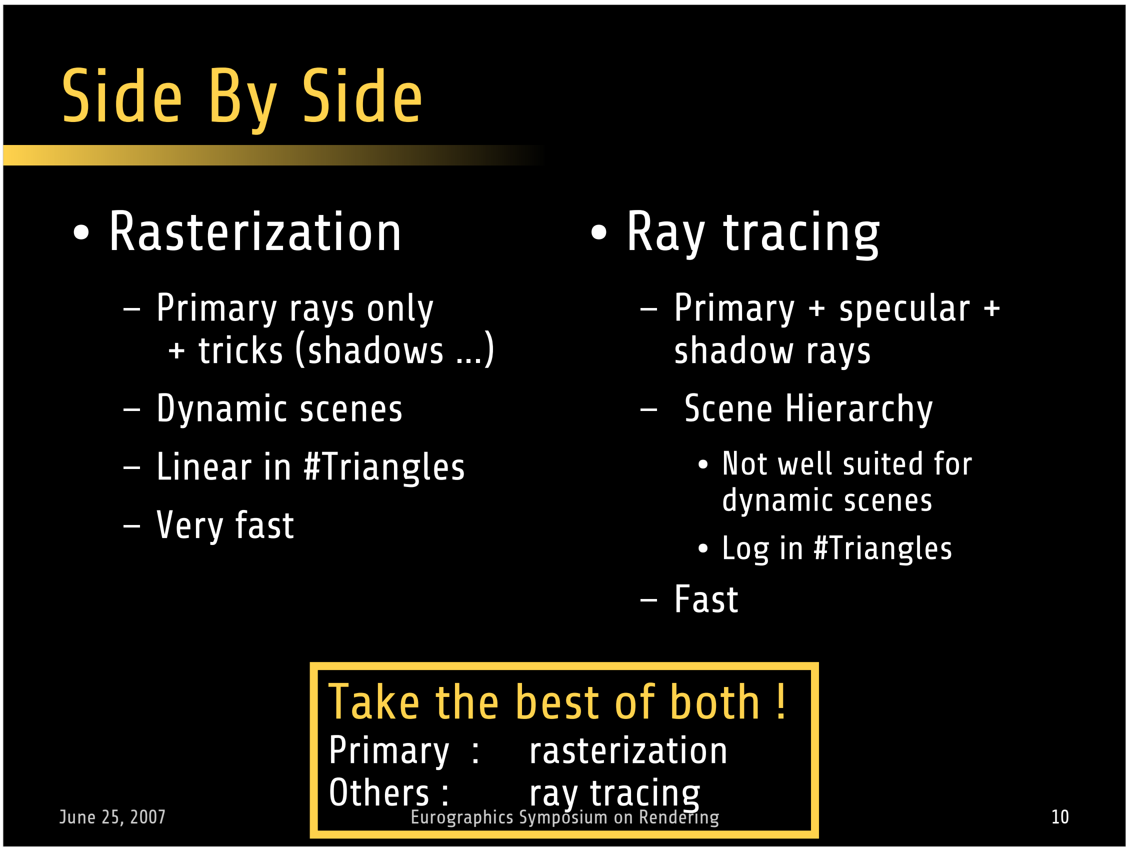
It is able to render very detailed and dynamic scenes extremely fast, and that is why it is so popular.

Side By Side

- Rasterization
  - Primary rays only + tricks (shadows ...)
  - Dynamic scenes
  - Linear in #Triangles
  - Very fast
- Ray tracing
  - Primary + specular + shadow rays
  - Scene Hierarchy
    - Not well suited for dynamic scenes
    - Log in #Triangles
  - Fast

June 25, 2007 Eurographics Symposium on Rendering 9

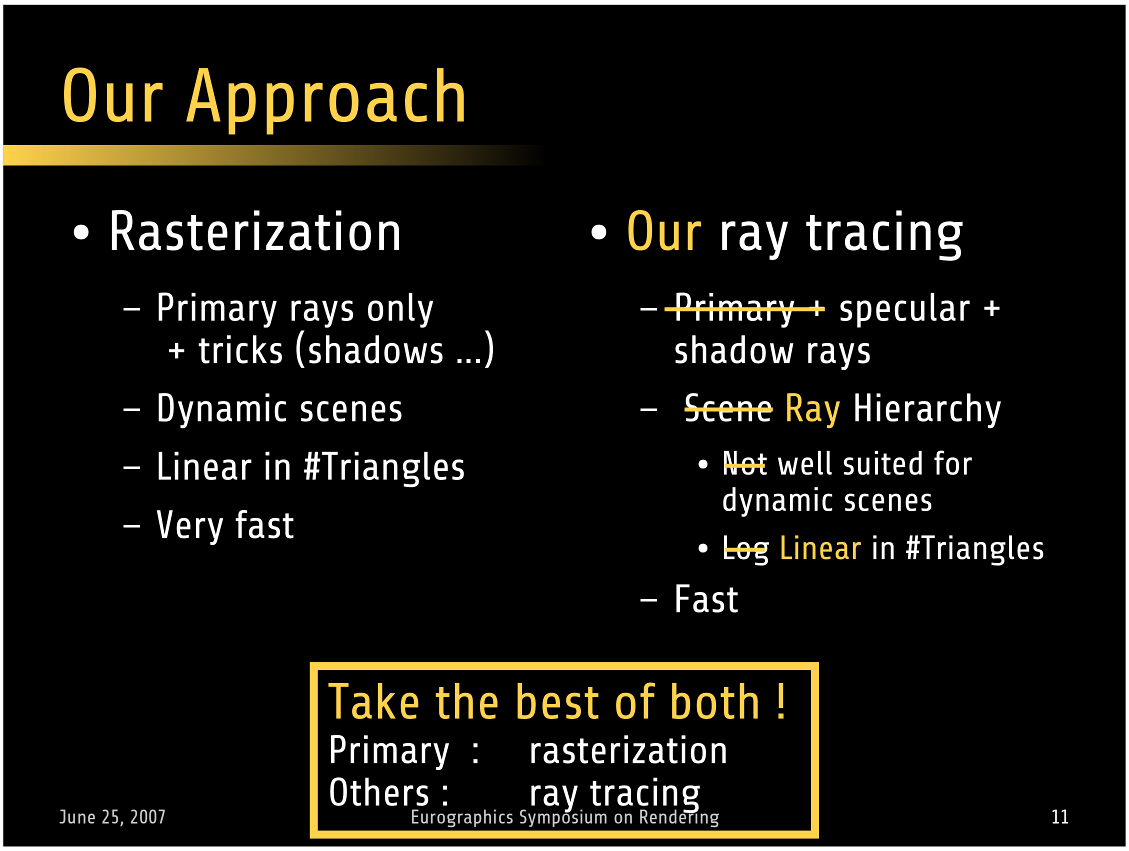
Let's put the two techniques side by side.



As rasterization is faster, it seems to be a good idea to use it when possible (that is for primary rays) and use ray tracing for the other rays.

But doing this would also suffer from the drawbacks of the two methods.

That is why we propose a different approach for ray tracing.



We focus on the secondary rays only.

And we replace the scene hierarchy by a ray hierarchy.

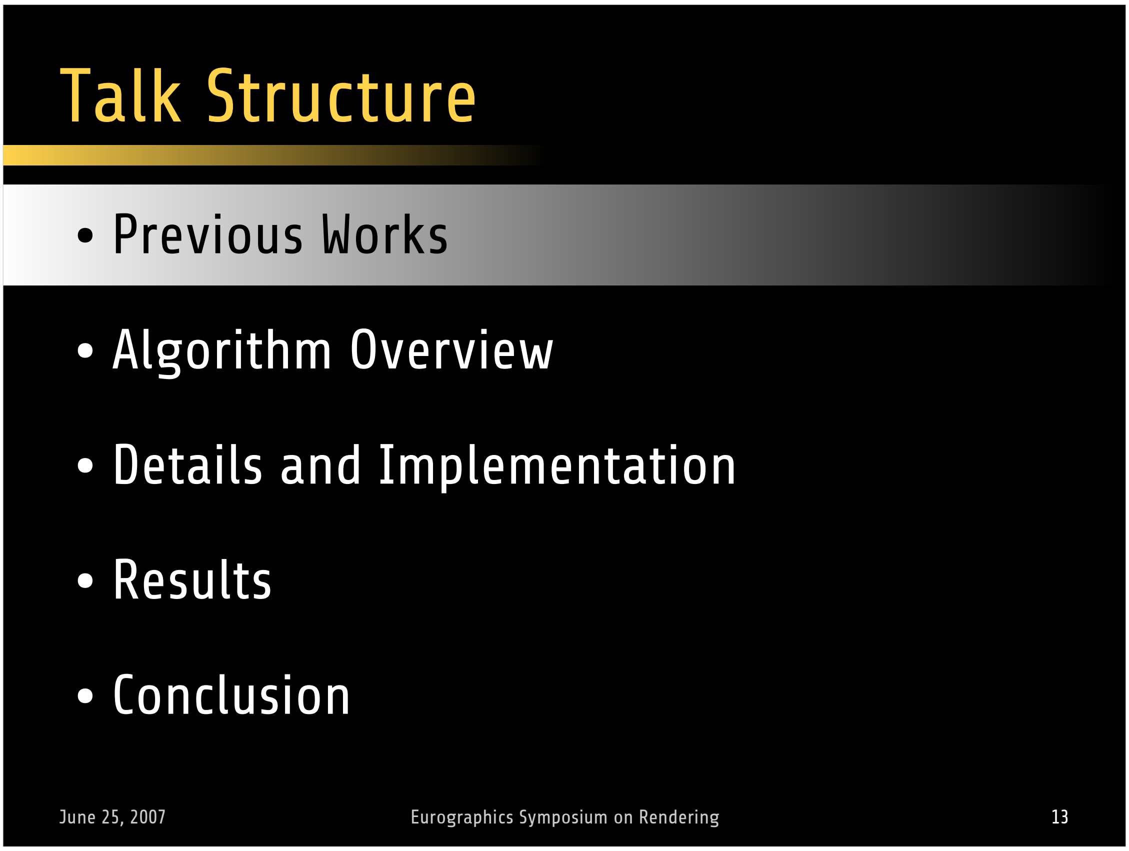
Thus, dynamic scenes are not anymore a problem for the algorithm.

We lose the logarithmic complexity, and have only a linear complexity. But rasterization is also linear anyways ...

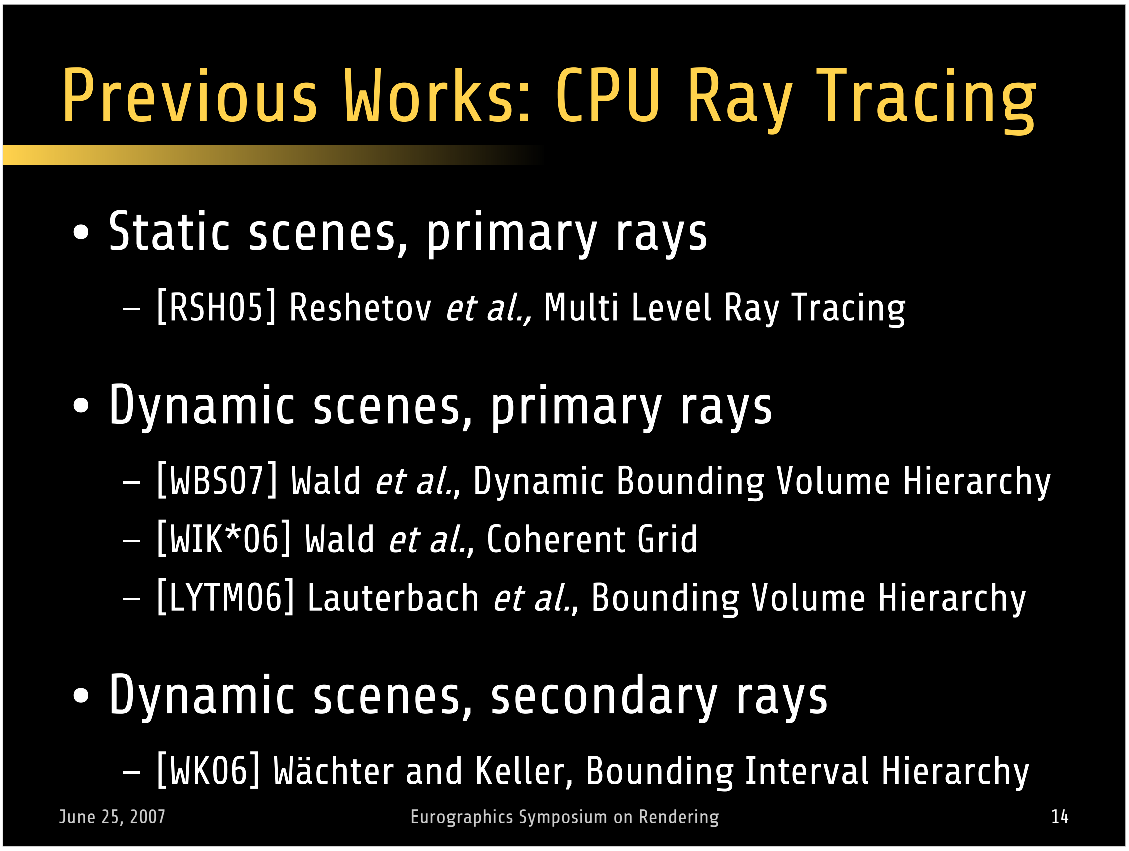
Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Conclusion

This is the structure of my talk



And I'll start with the previous works



Here are the recent previous work in interactive raytracing on CPU.

They all rely on a stucture on the scene (hierarchy or grid).

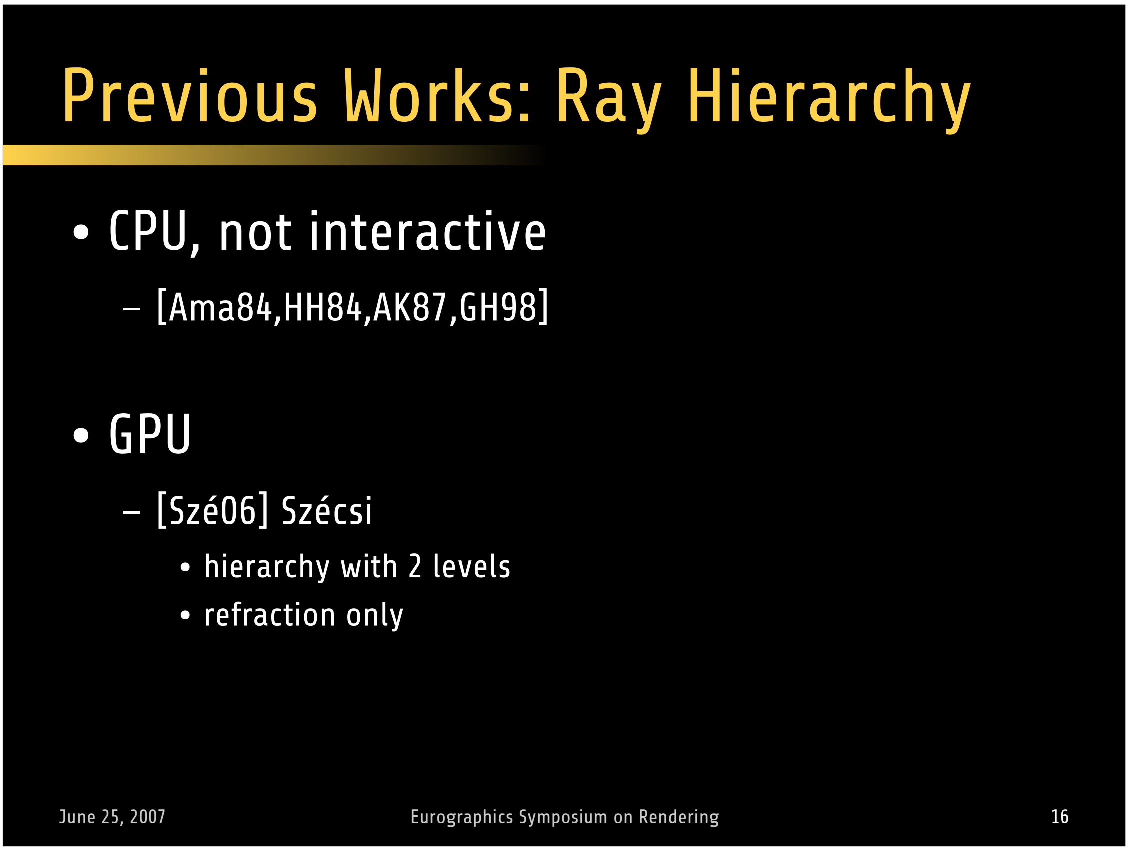
For dynamic scenes, bounding volumes hierarchies are the most popular.

Previous Works: GPU Ray Tracing

- Static scenes, kd-tree
  - [FS05] Foley and Sugerman
  - [HSHH07] Horn *et al.*
- Static scenes, Bounding Volume Hierarchy
  - [TS05] Thrane and Simonsen
- Dynamic scenes, primary rays
  - [CHCH06] Carr *et al.*, Geometry Images

On GPU,

the papers focus mostly on static scenes and rely on a scene stucture: kd-tree or bounding volume hierarchy



On the other hand ray hierarchies are less popuar.

Several papers use them on CPU, but do not aim at interactive rendering

Szecsi is the most closely related work.

Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Conclusion

Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Conclusion

Main Features

- Rasterization for primary rays
- Ray tracing for secondary rays
- Ray hierarchy: rebuilt at each frame
- Runs completely on GPU

Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

Here are the main steps of our algorithm.

Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

In one rasterization pass, we render the scene and spawn the secondary rays, which will be the leaves of our hierarchy

Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

Completely rebuilt at each frame.
This step is very fast.

Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

Main Step.

Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

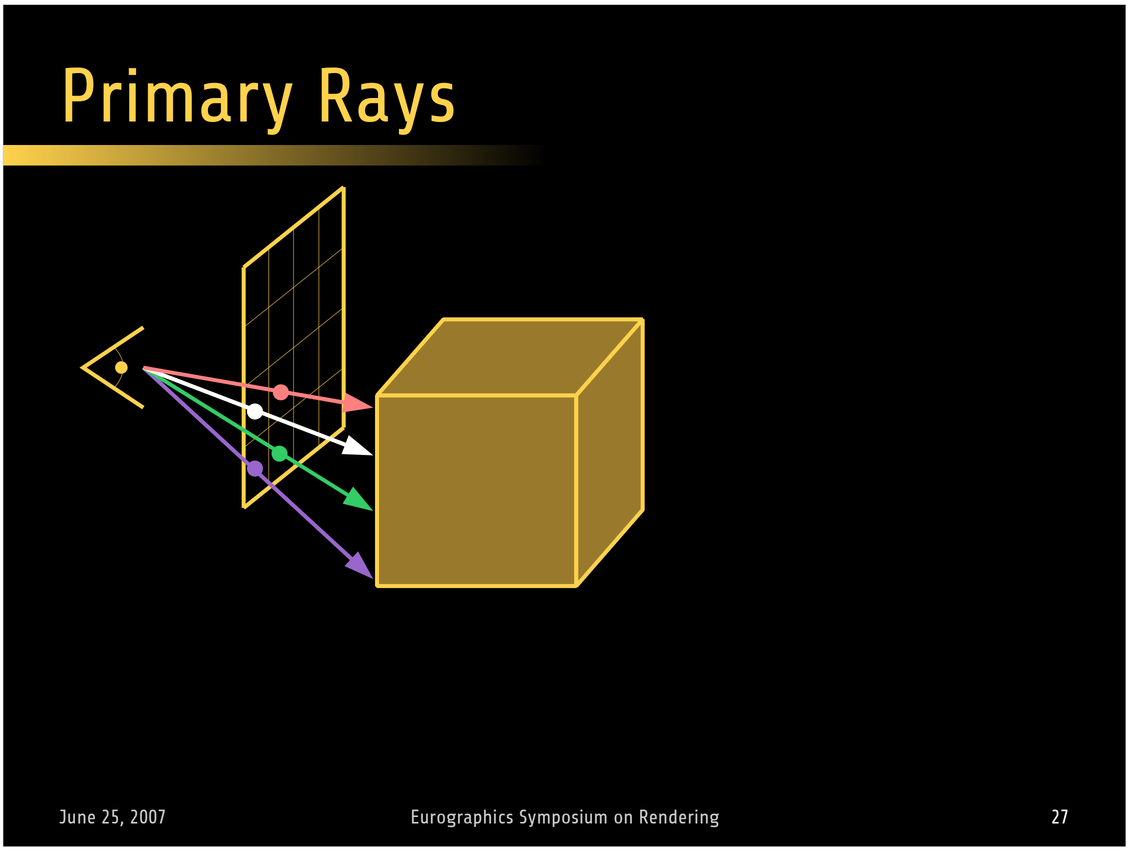
Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

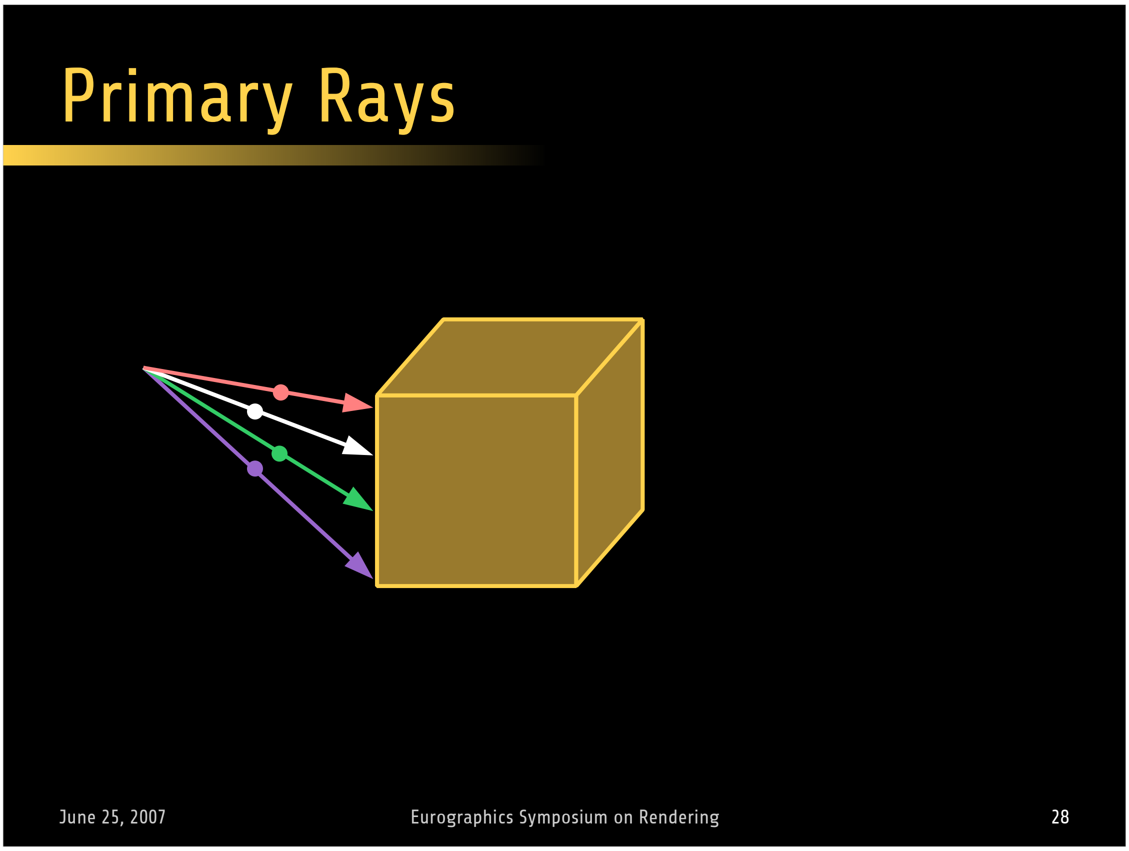
And rebuild a new hierarchy.

Algorithm Overview

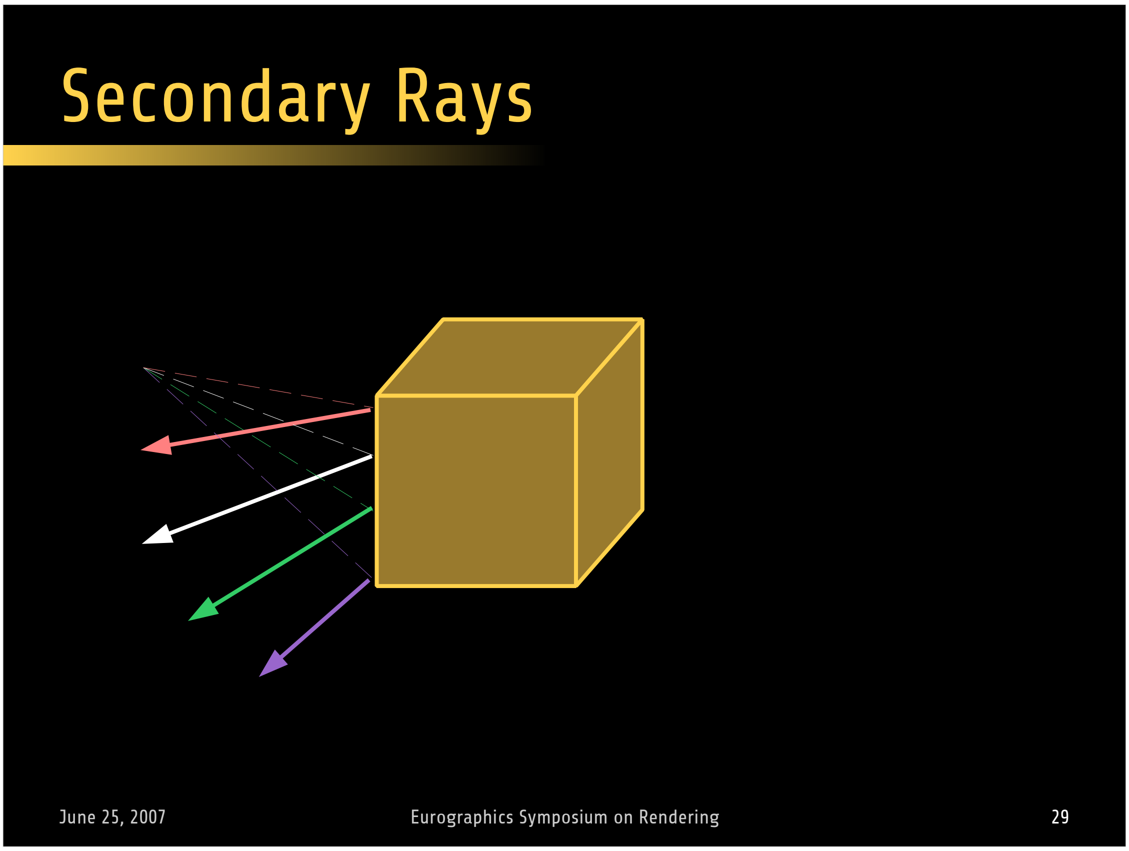
1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays



Lets consider the four rays at the bottom left corner of the picture.

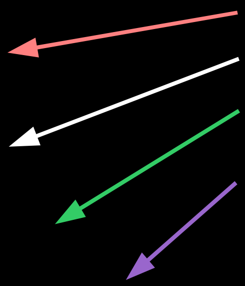


They hit the scene.



And spawn secondary rays, reflection here for example.

Secondary Rays



Secondary Rays

Leaves of the hierarchy

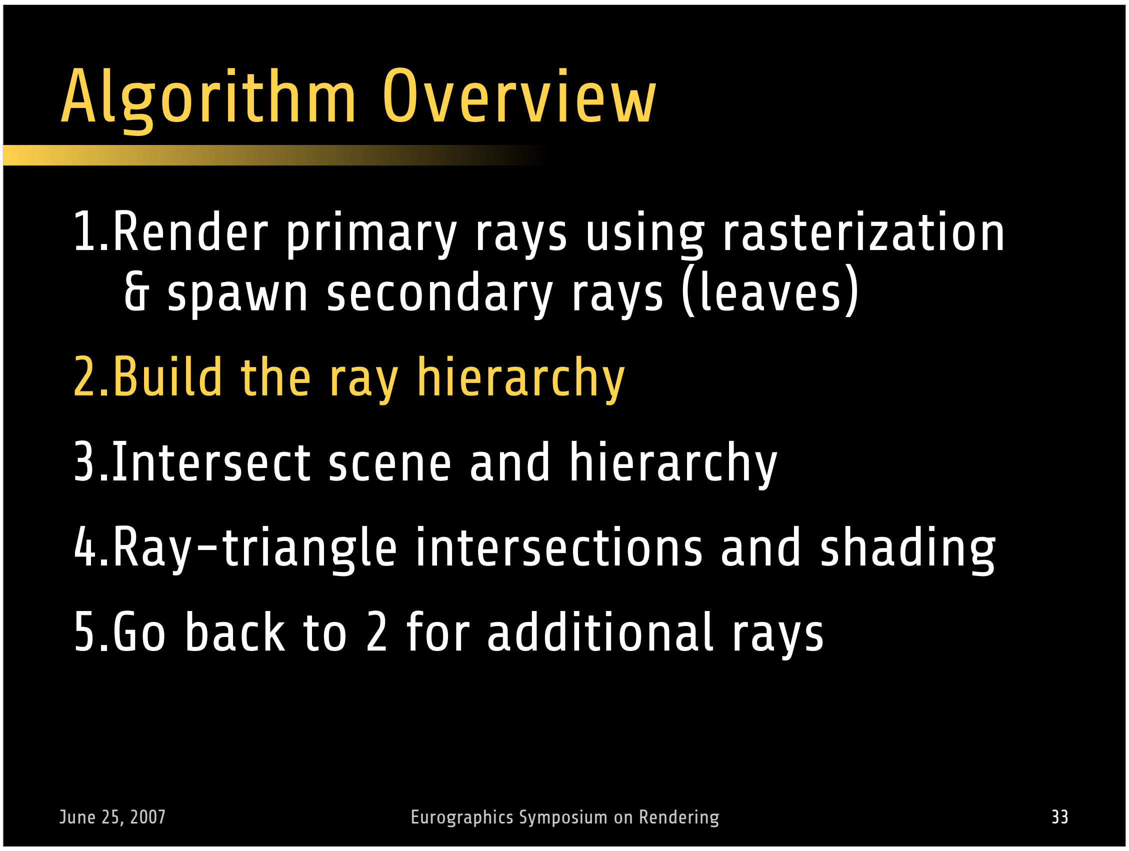
Corresponding pixels

Stored in 2 textures (position + direction)

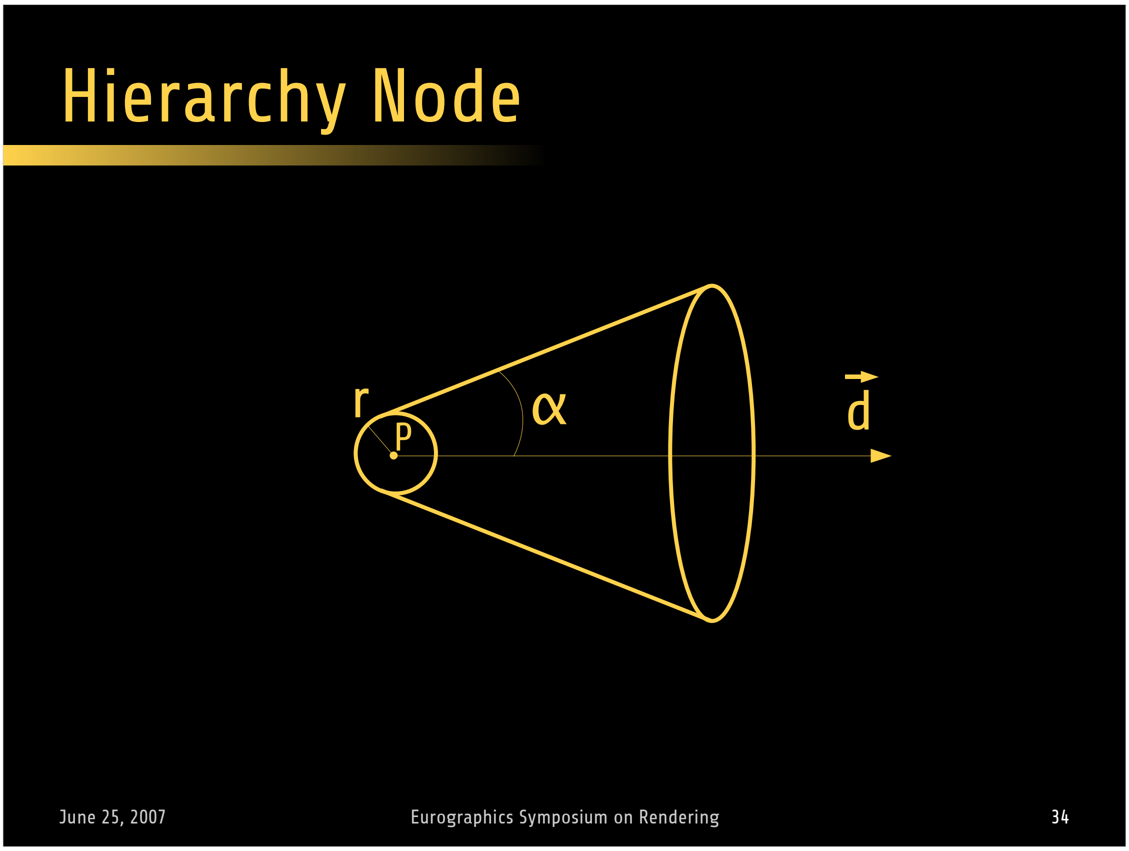
Same size as the screen

Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays



Then we build the ray hierarchy from the leaves



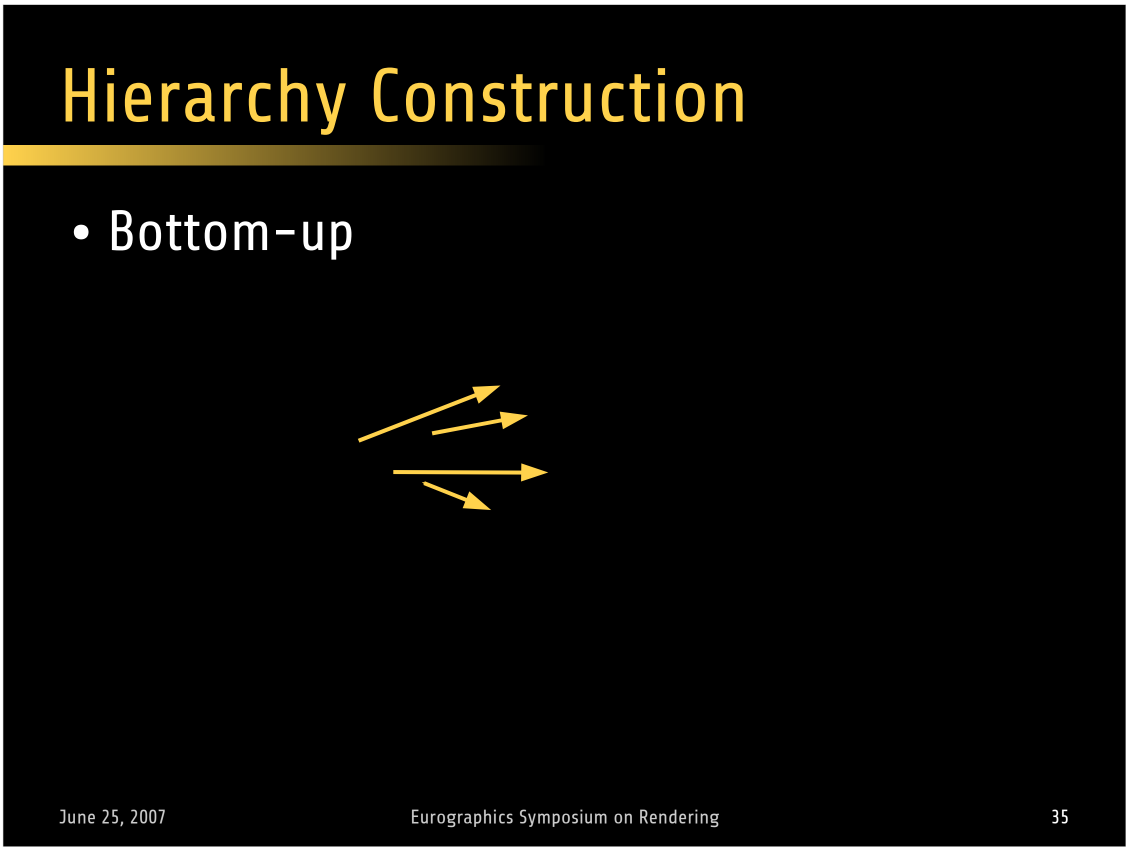
All nodes of our hierarchy are cones capped by spheres defined by their center, direction, radius and spread angle.

They can be stored as 8 floating point values.

Leaves (rays) are seen as cones with null radius and angle.

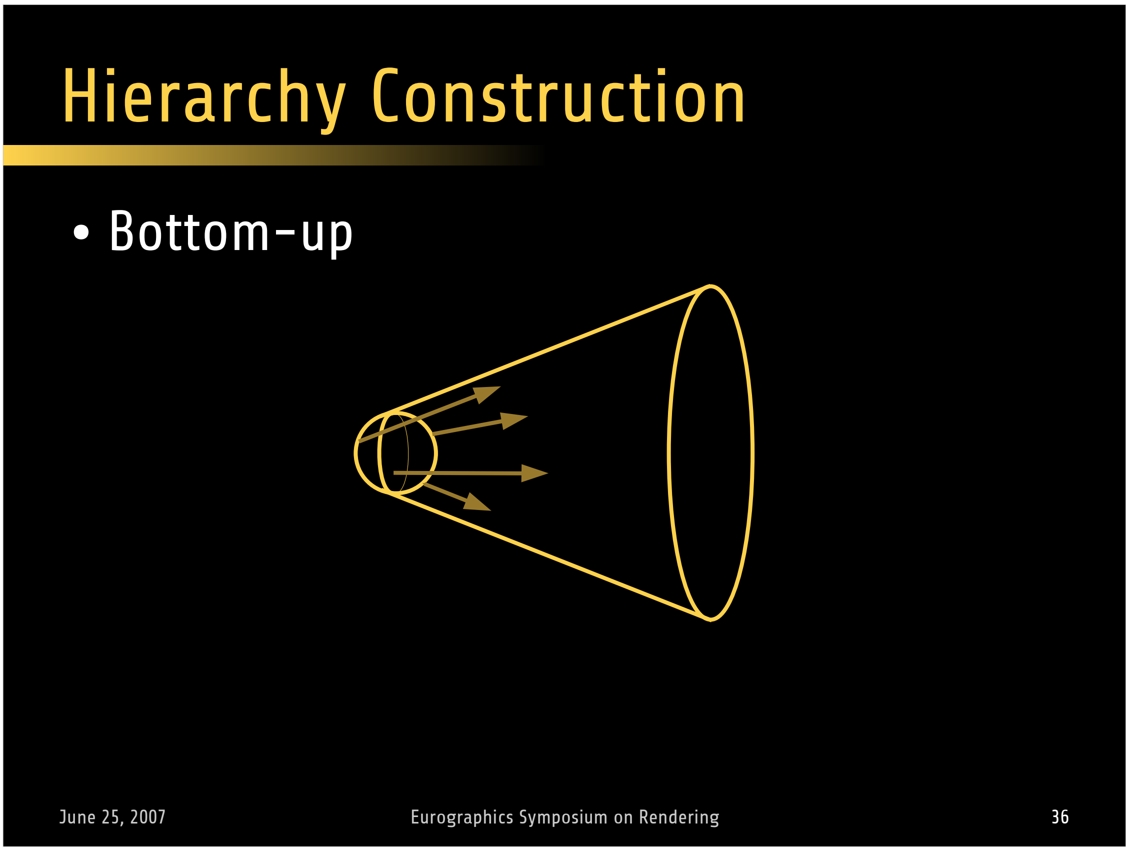
We have chosen this shape because:
- fast construction
- fast intersection

But pretty much any other shape would work.



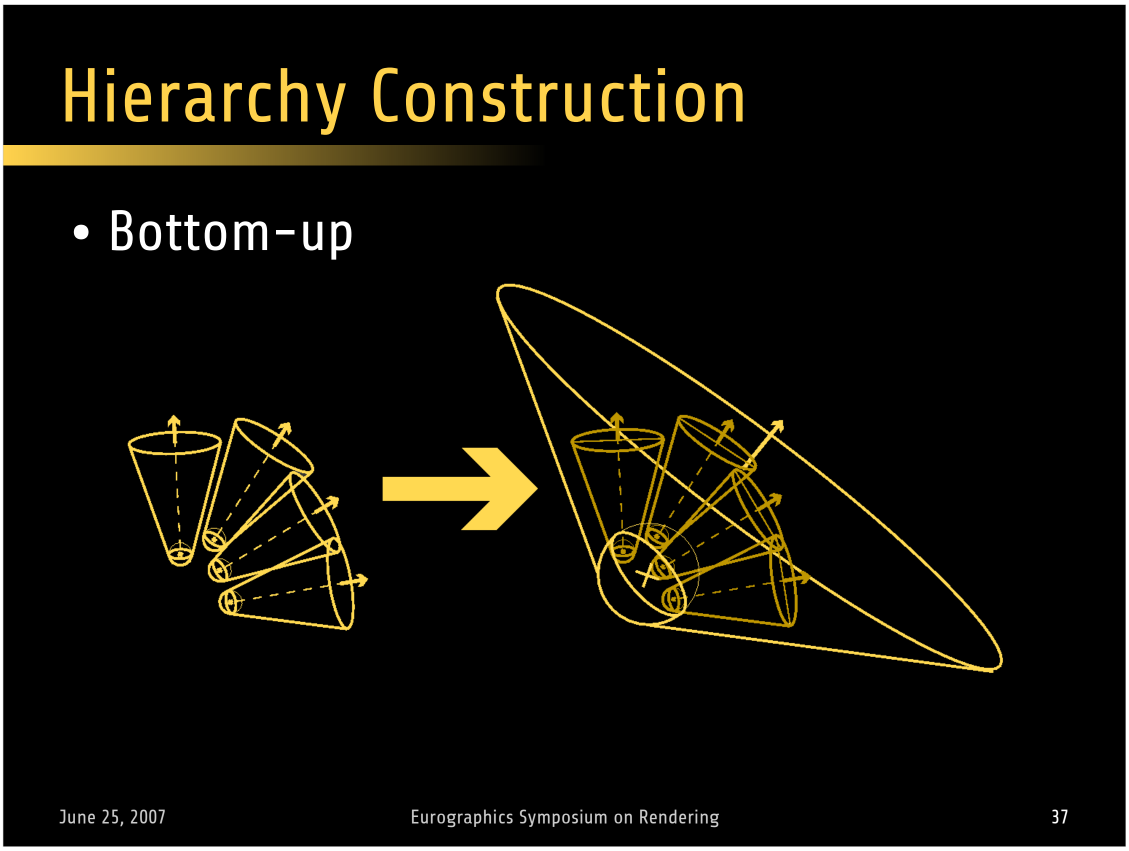
The hierarchy is constructed bottom up, from the leaves to the root.

The leaves are the secondary rays, and are seen as cones with null angle and radius.

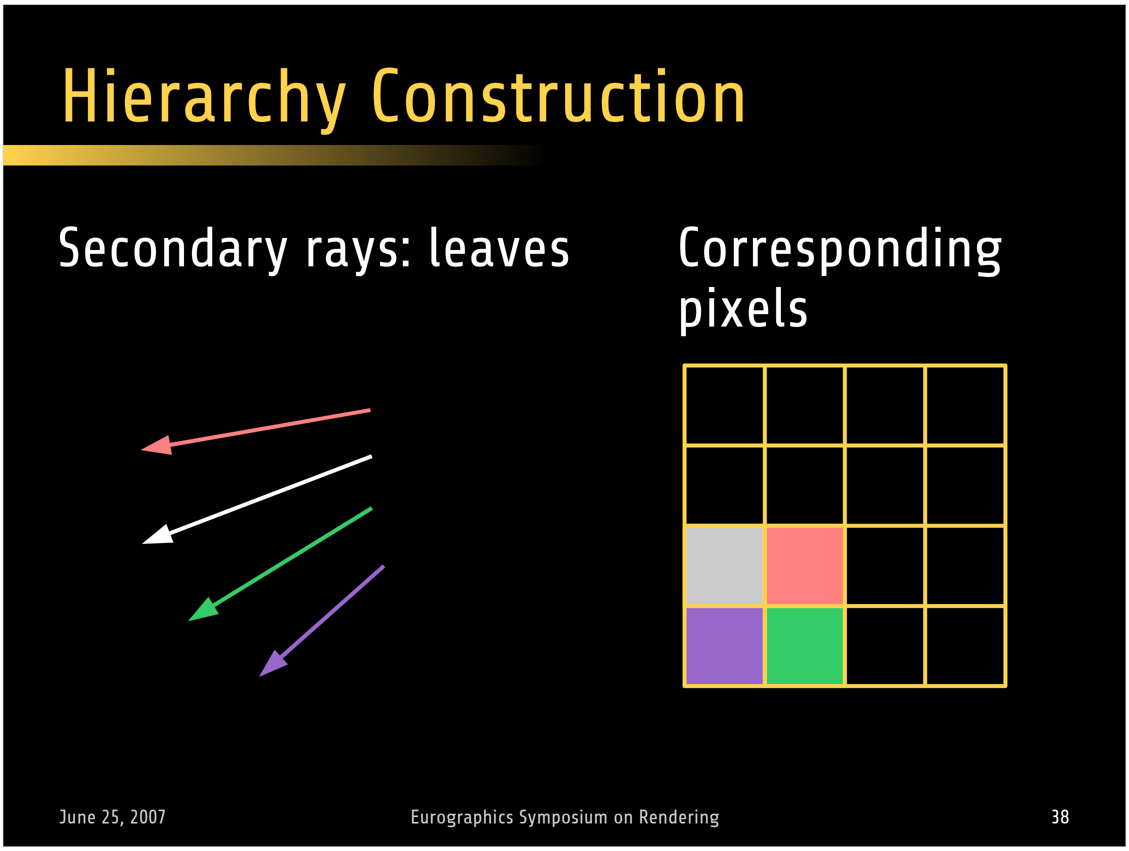


To compute the parent node, we compute a cone that encloses all the rays.

We try do find the smallest one, but we use an approximation to speed up the computation.

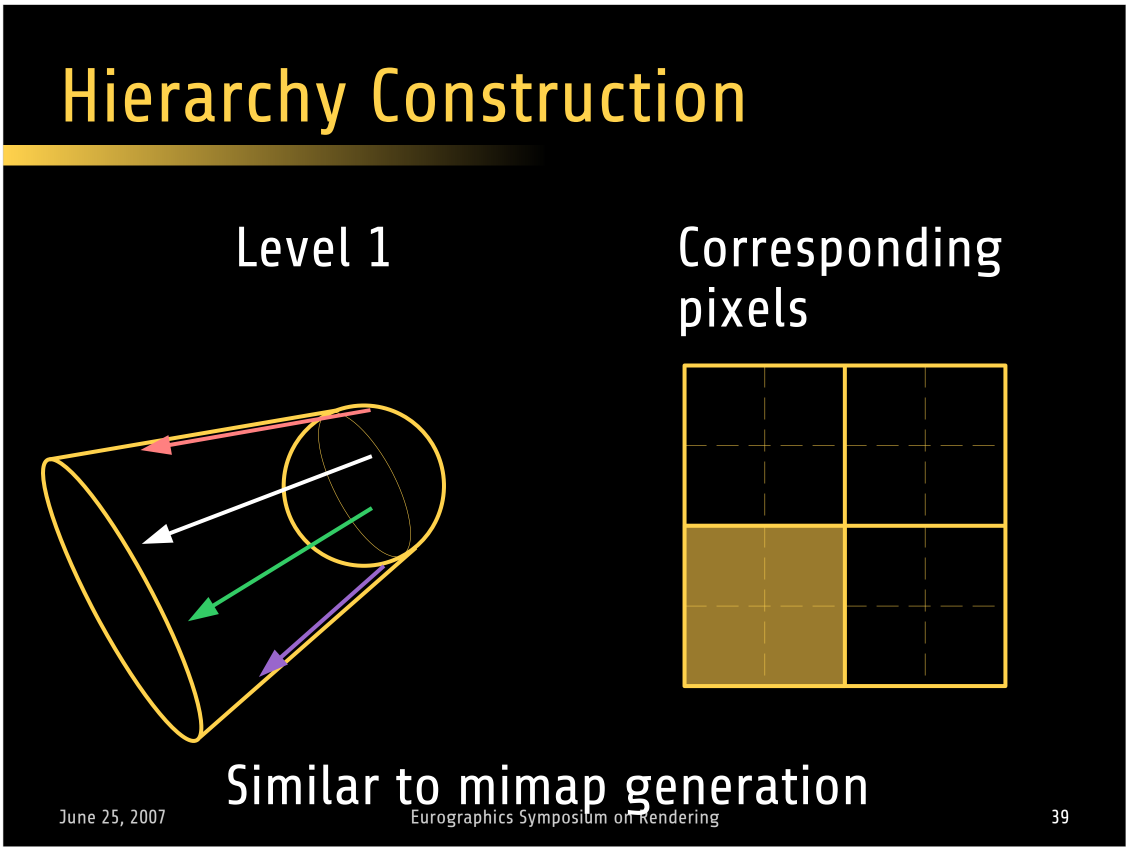


The same process is applied at each level to build the full hierarchy.



As I said, the secondary rauys are stored in a texture.

We choose to build our hierarchy as a pyramid of 2x2 squares because it is very fast to implement on GPU.

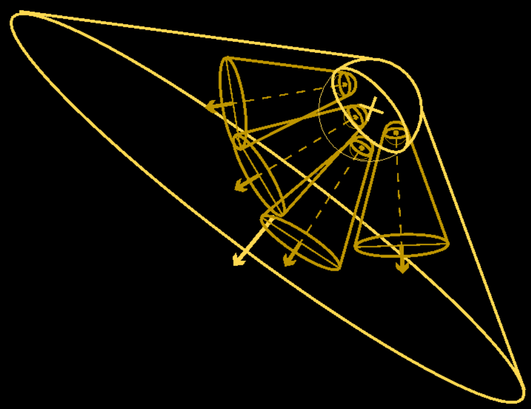


The process is similar to mip map generation.

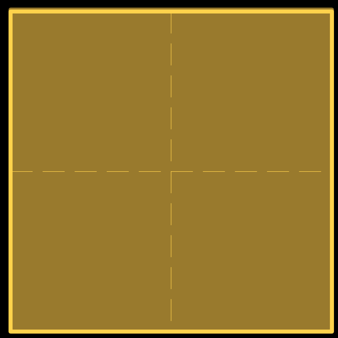
Each new level of the hierarchy can be stored in a texture of half resolution, and easily computed in one rendering pass using a fragment shader.

Hierarchy Construction

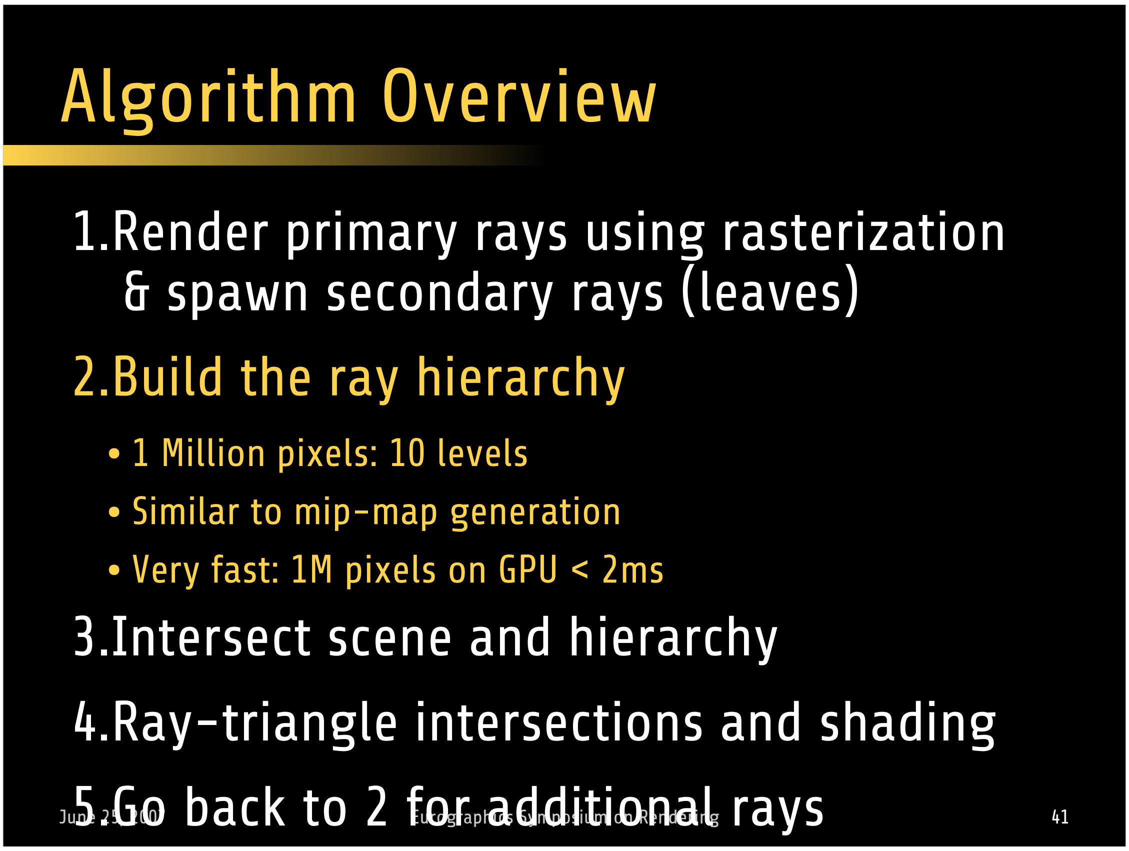
Root



Corresponding pixels



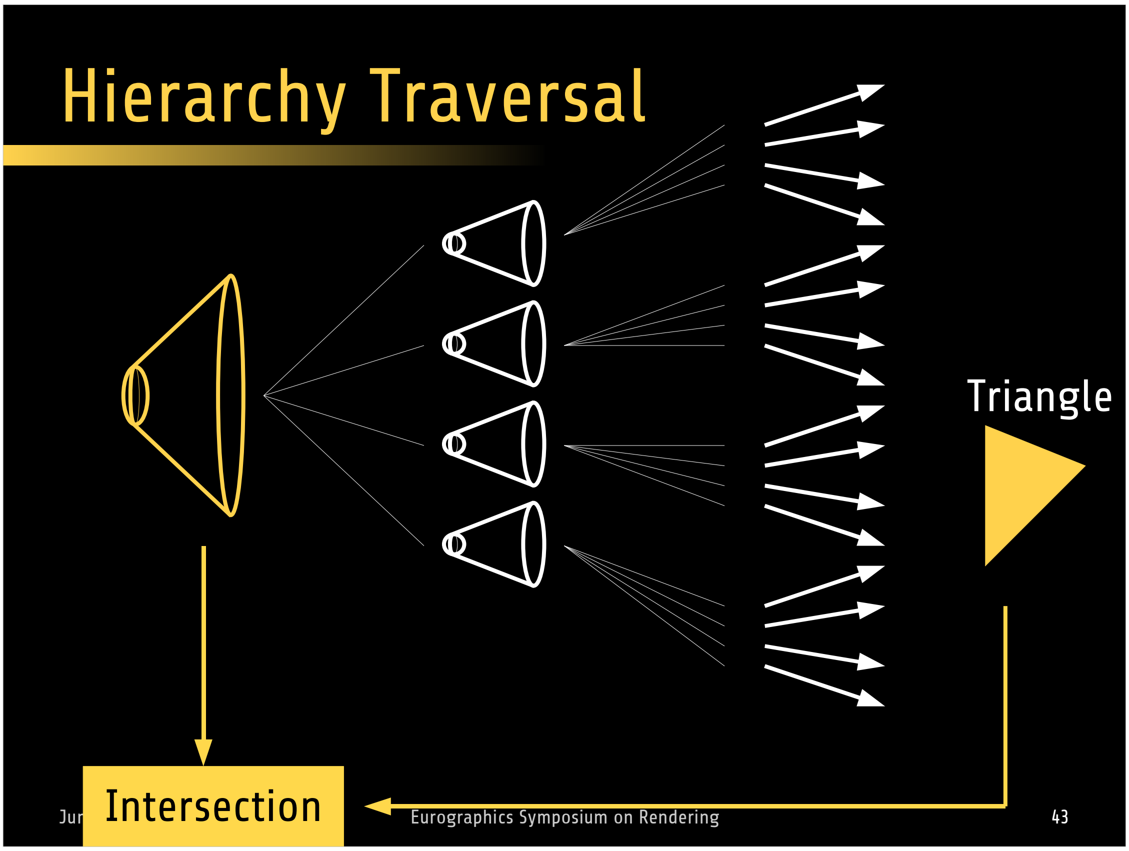
Similar to mimap generation



Rebuilt entierly at each frame

Algorithm Overview

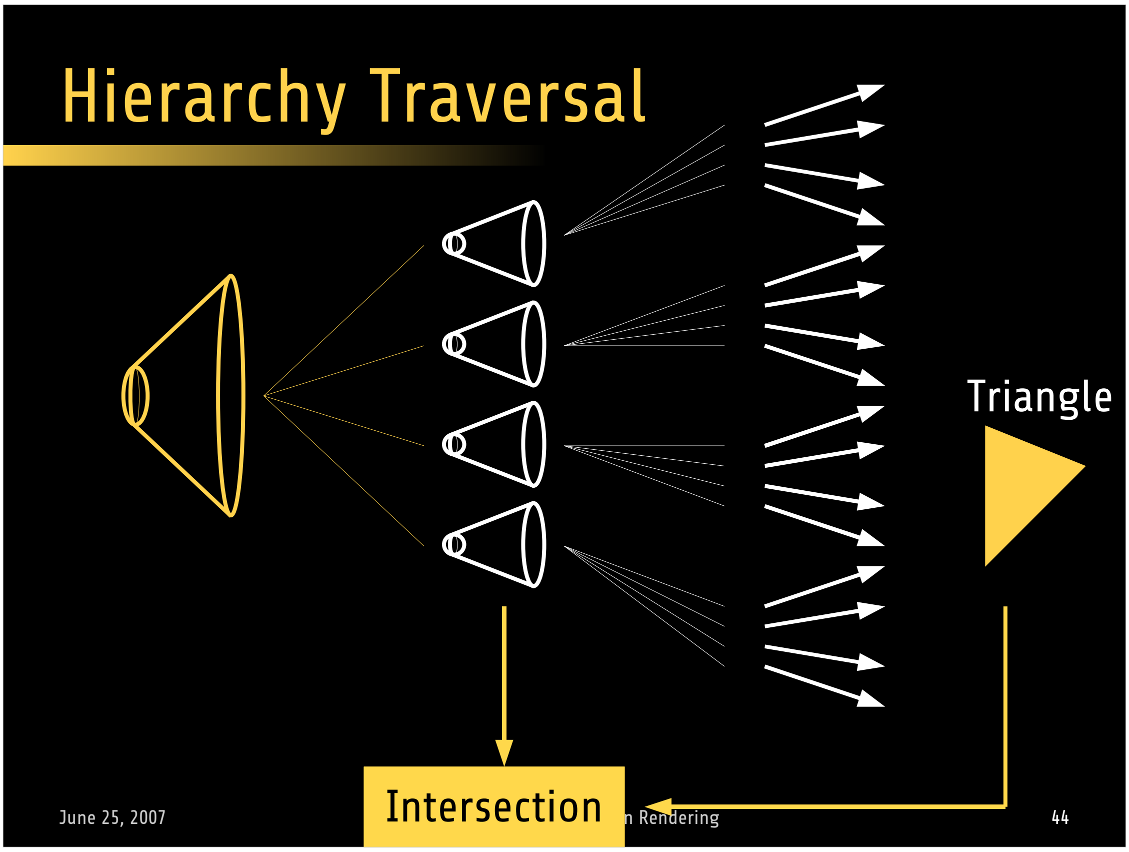
1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays



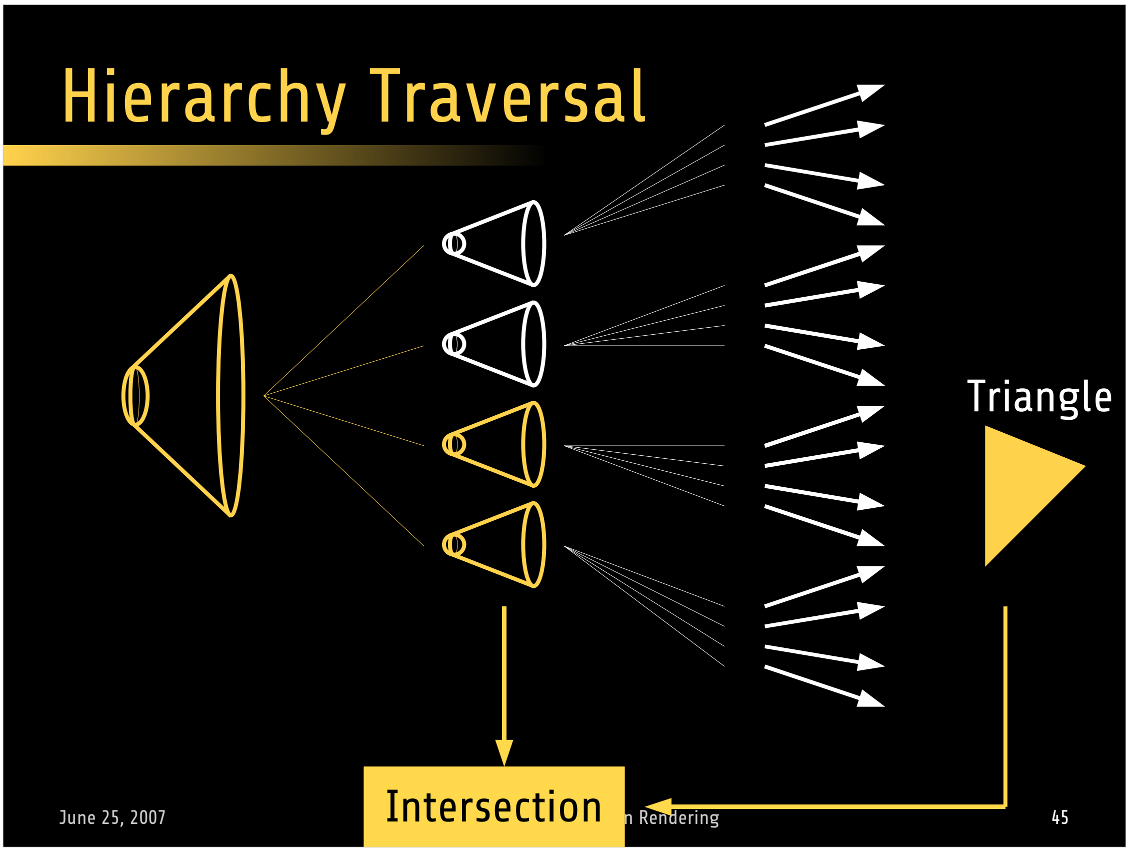
Hierarchy is traversed top down, from the root to the leaves.

Here is a triangle of the scene, on the right of the screen.

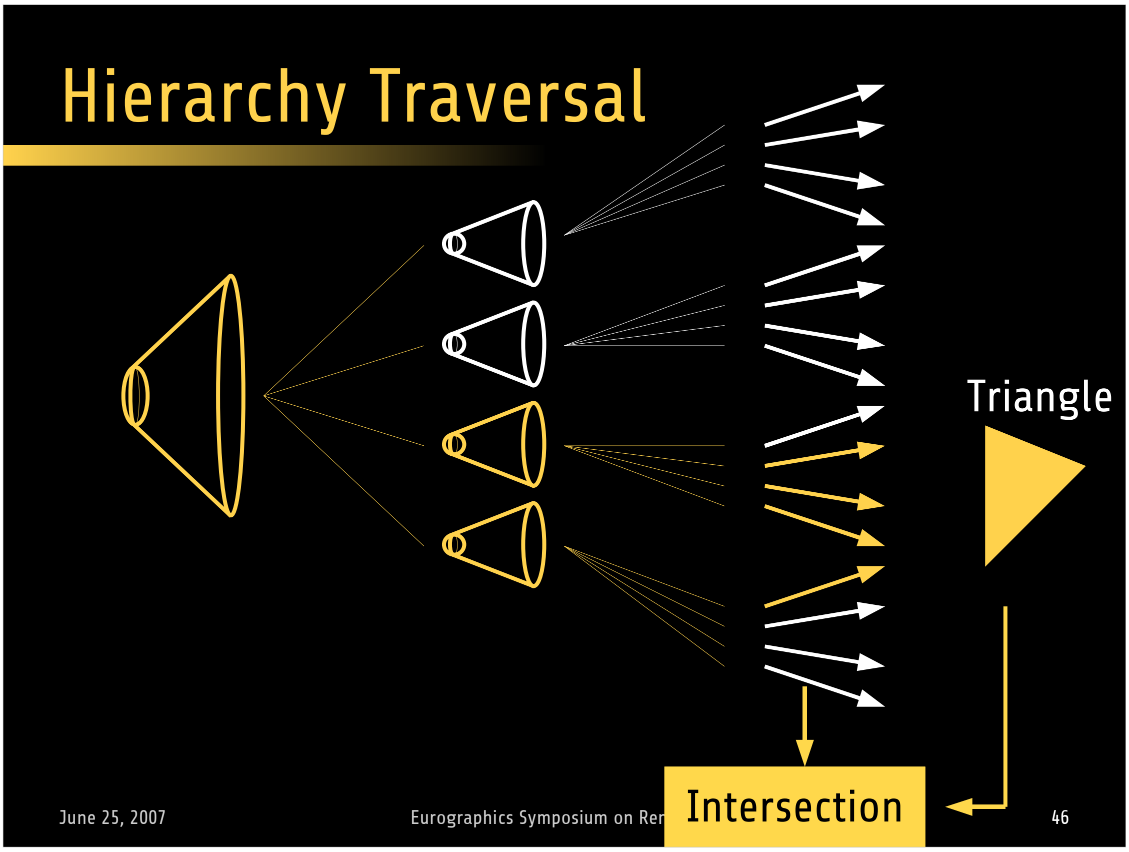
The triangle intersects the root.



Then it is tested against the 4 children.



2 cones intersect the triangle (in yellow), the other 2 are discarded.



We repeat this process until we reach the leaves, which are also the secondary rays.

All triangles can be processed in parallel because each one is independant from the others.

They also undergo the same number of intersection tests.

Algorithm Overview

1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
   - Process triangles in parallel
   - Same execution path length, minimal branching
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

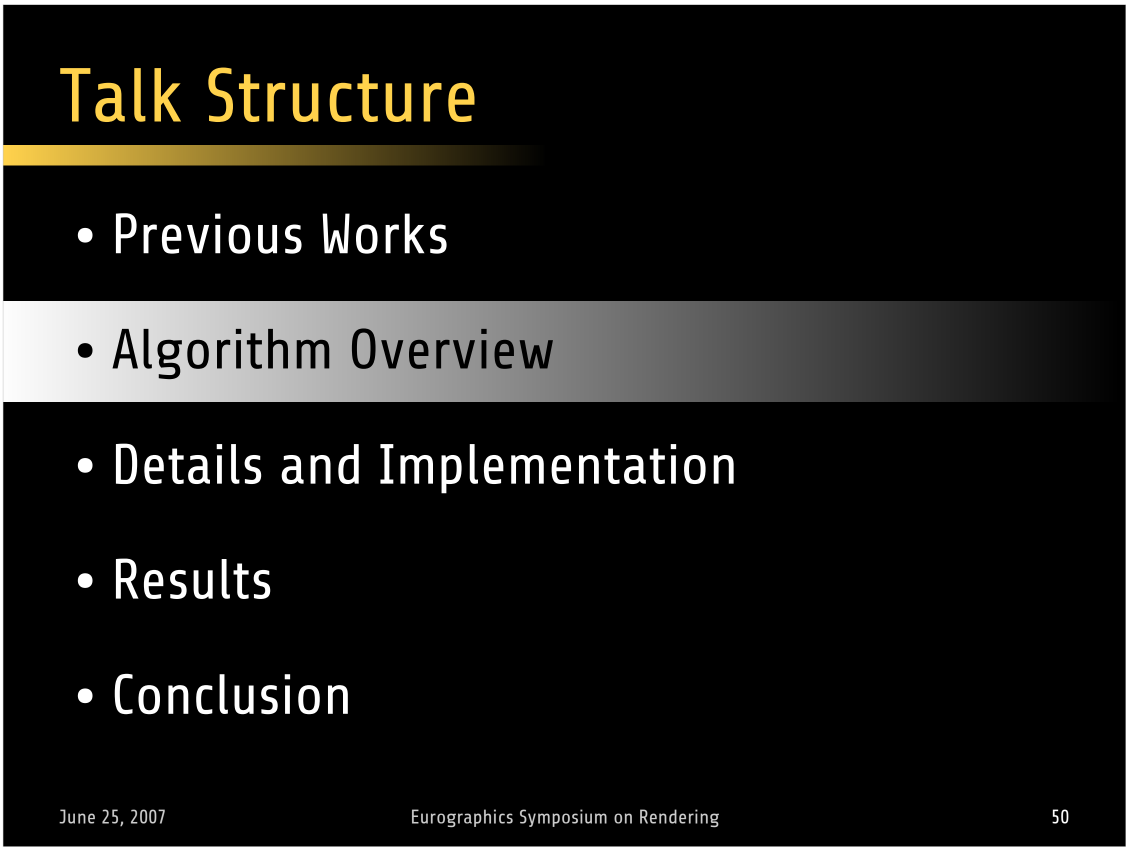
This step is thus very well suited for GPU implementation

Algorithm Overview

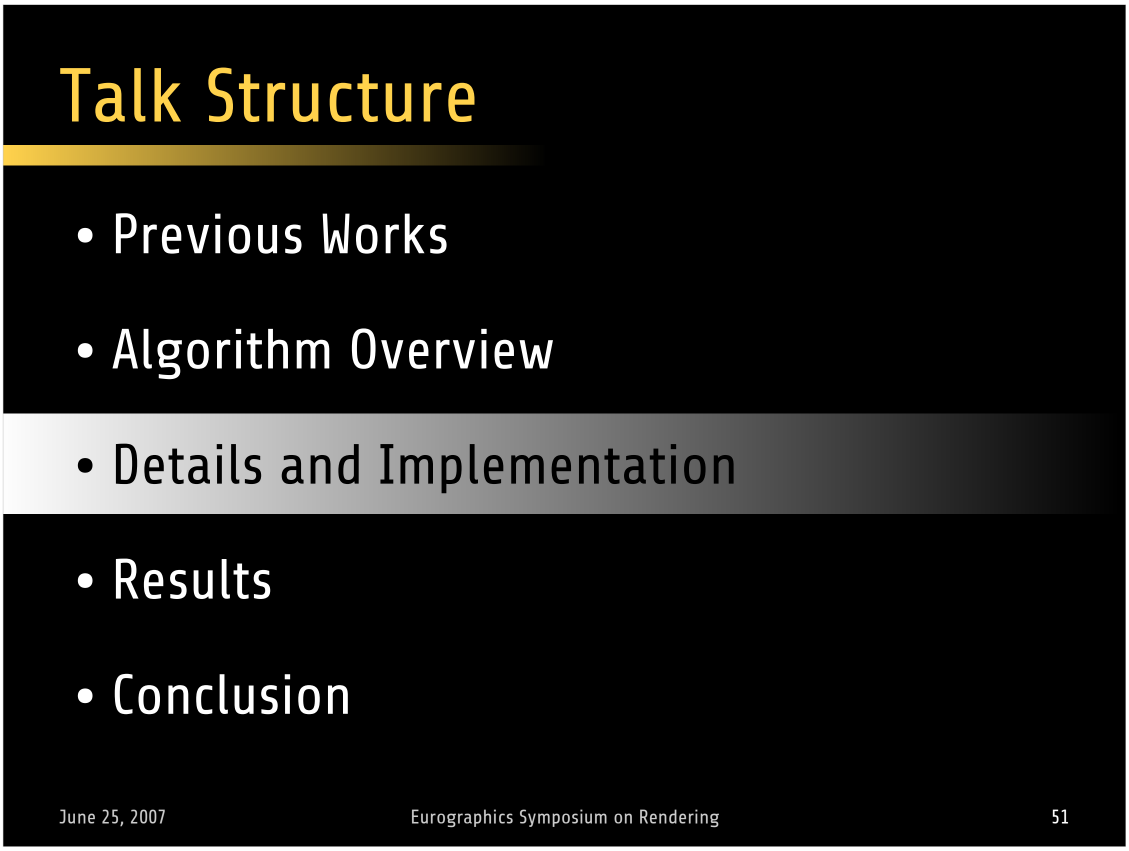
1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays

Algorithm Overview

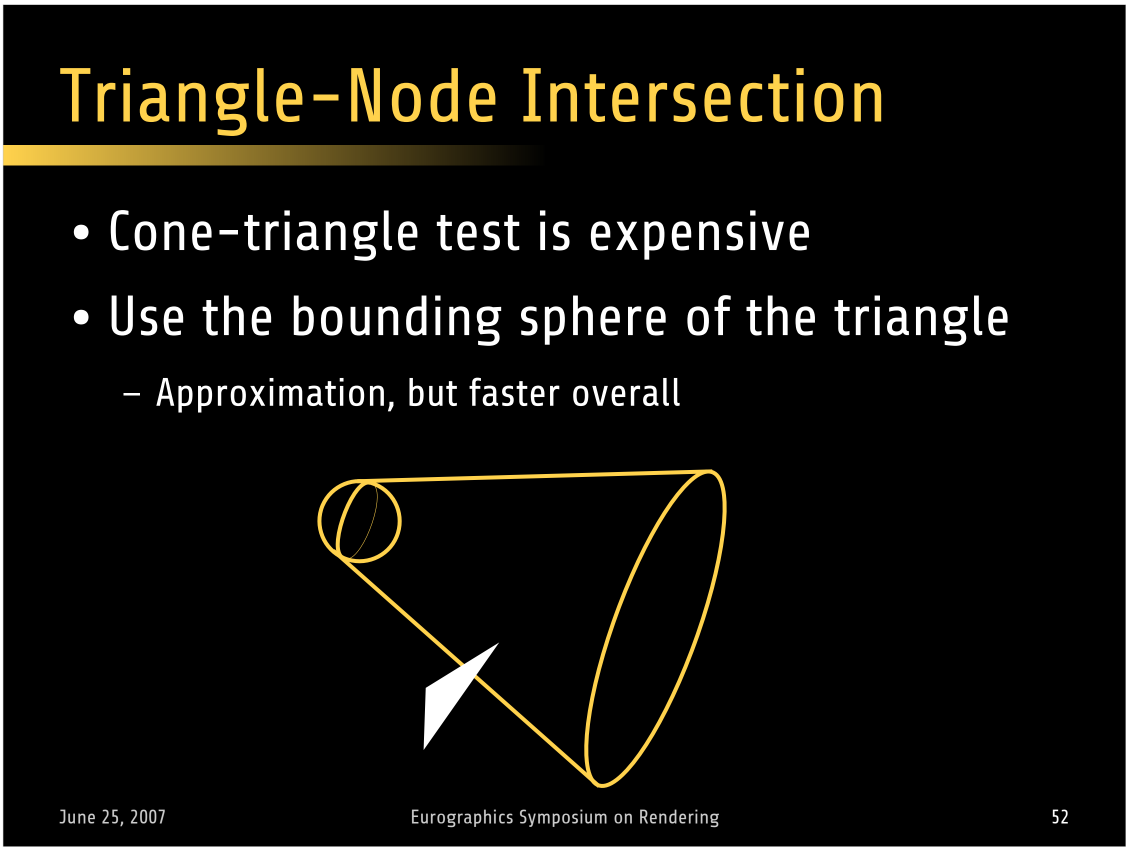
1. Render primary rays using rasterization & spawn secondary rays (leaves)
2. Build the ray hierarchy
3. Intersect scene and hierarchy
4. Ray-triangle intersections and shading
5. Go back to 2 for additional rays



Now that I have described the structure of the algorithm,



I will go into some details and implementation issues

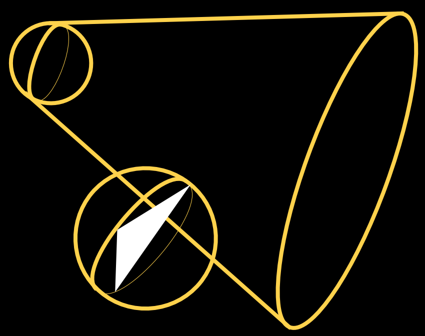


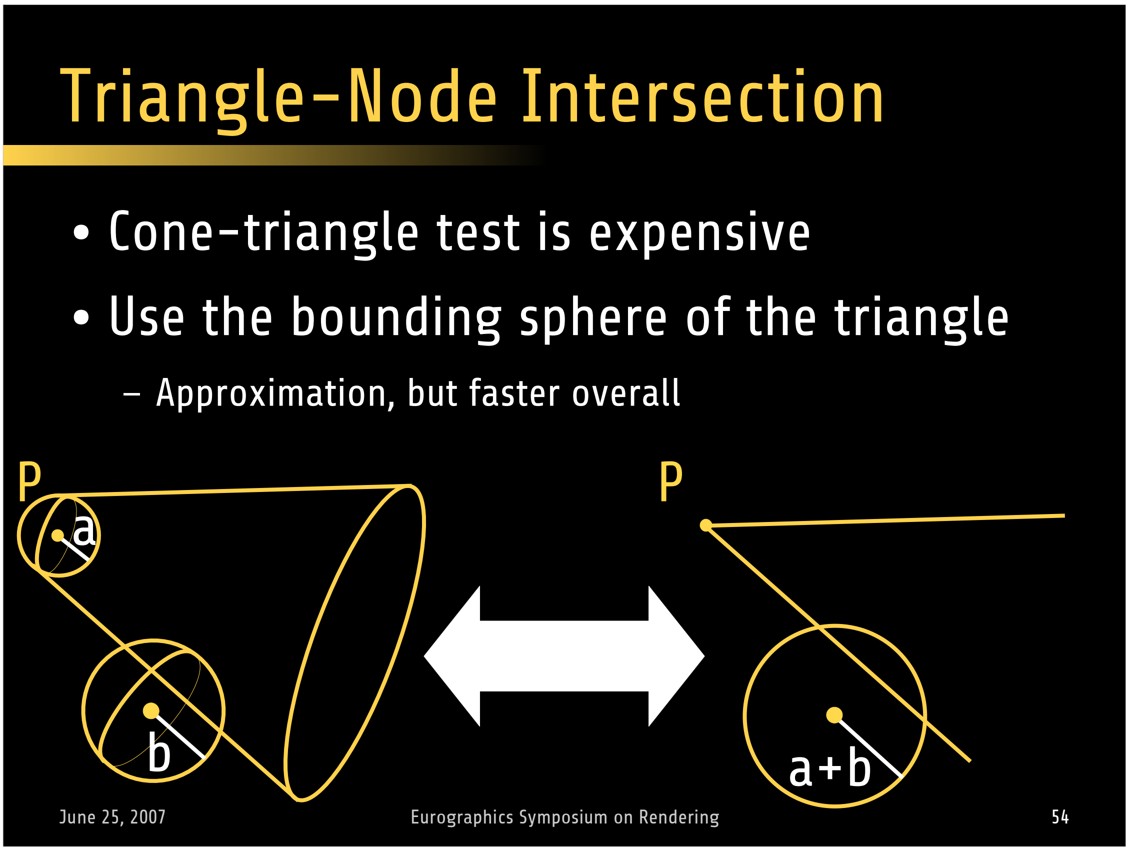
I begin with the Triangle-Node intersection Test.

Actually, cone triangle intersection is expensive ...

Triangle-Node Intersection

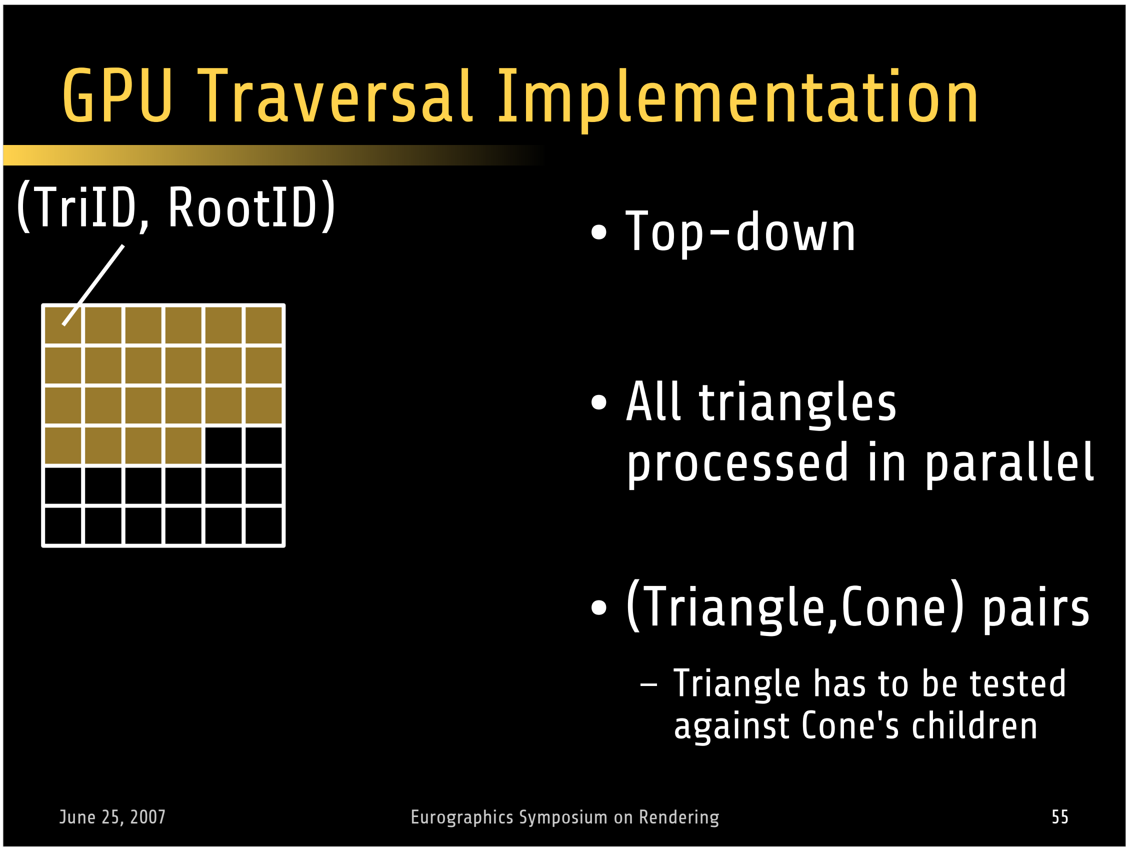
- Cone-triangle test is expensive
- Use the bounding sphere of the triangle
  - Approximation, but faster overall





The intersection of a capped-cone and a sphere is a 3D problem but reduces to a very simple 2D problem :

The intersection of a 2D cone and a larger circle

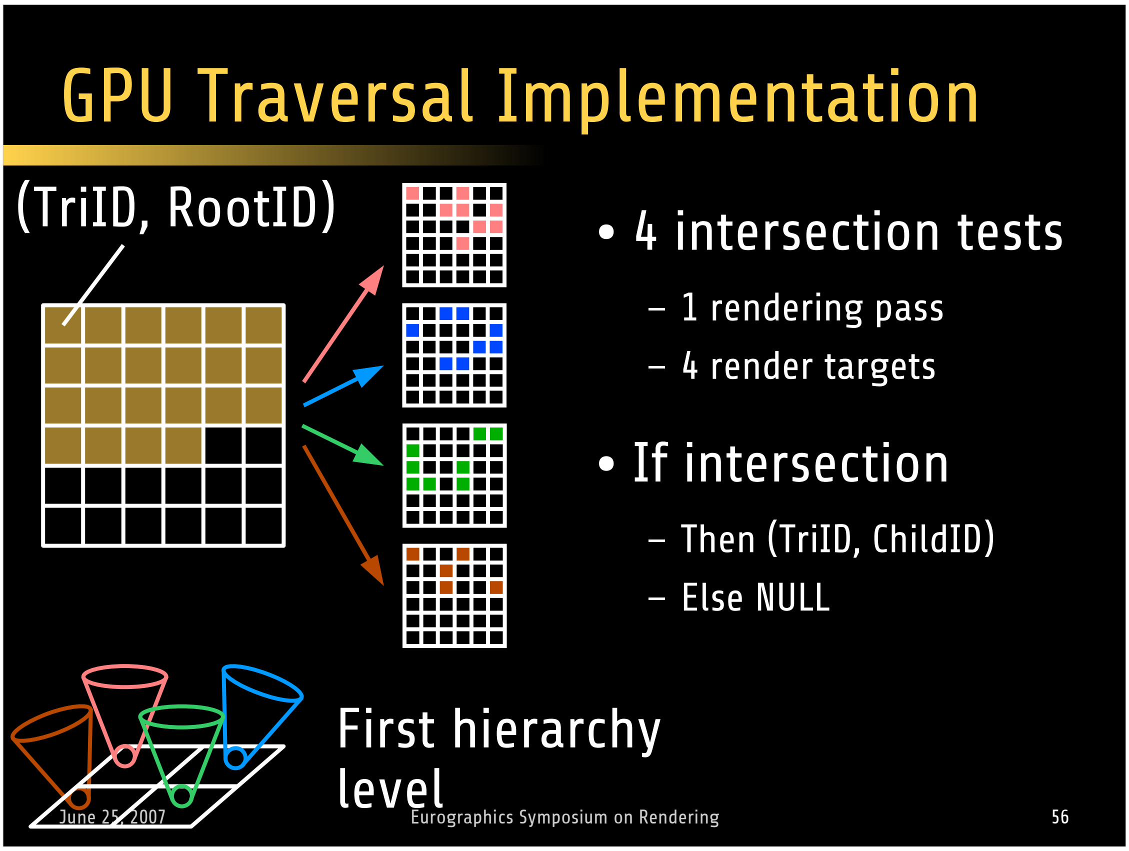


Now, I will present you another implementation detail: the GPU traversal of the hierarchy.

As I said, the triangles are processed independantly, and we use the GPU parallelism here.

All triangles are stored in a texture and then they will go down the hierarchy, by one level at each rendering pass.

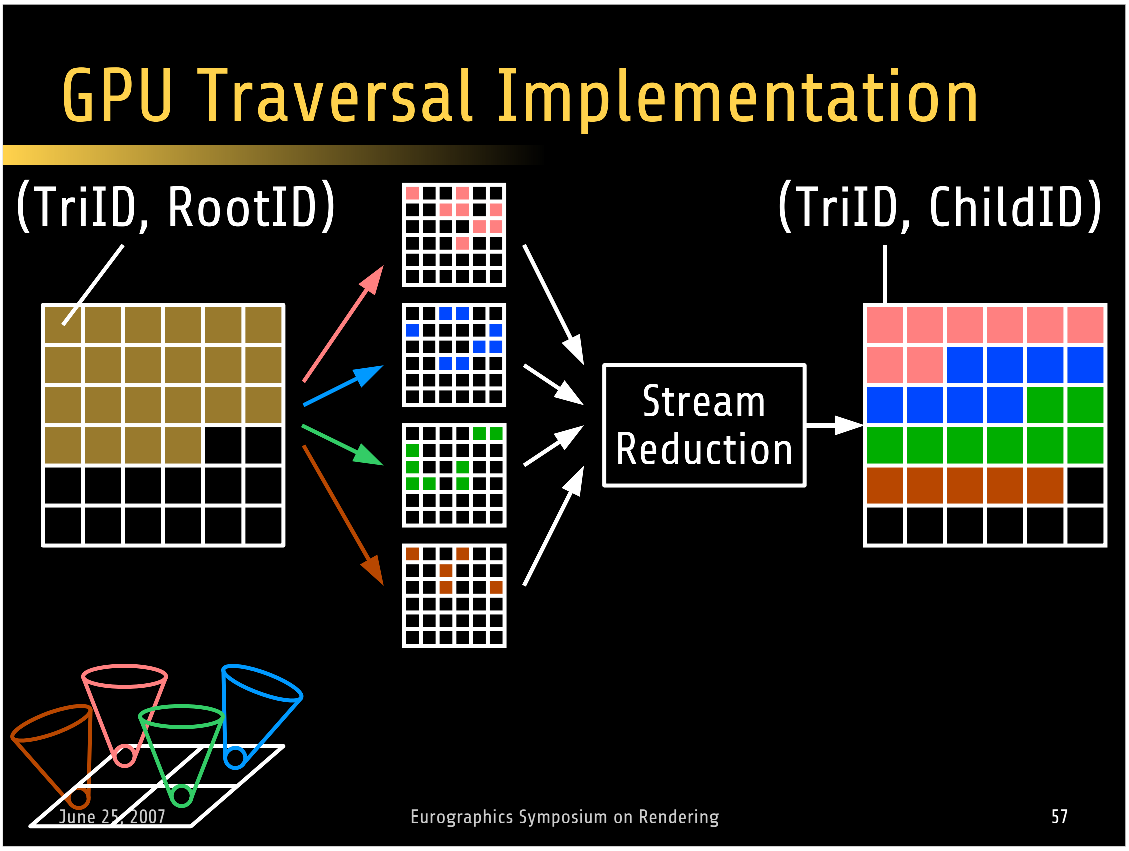
The triangles are paired with the node they intersect. To initialize our algorithm, they are all paired with the root, as you can see on the left.



In a fragment shader with 4 render targets, we perform the intersection of the triangle with the 4 children.

Each render target correspond to the intersection with one child.

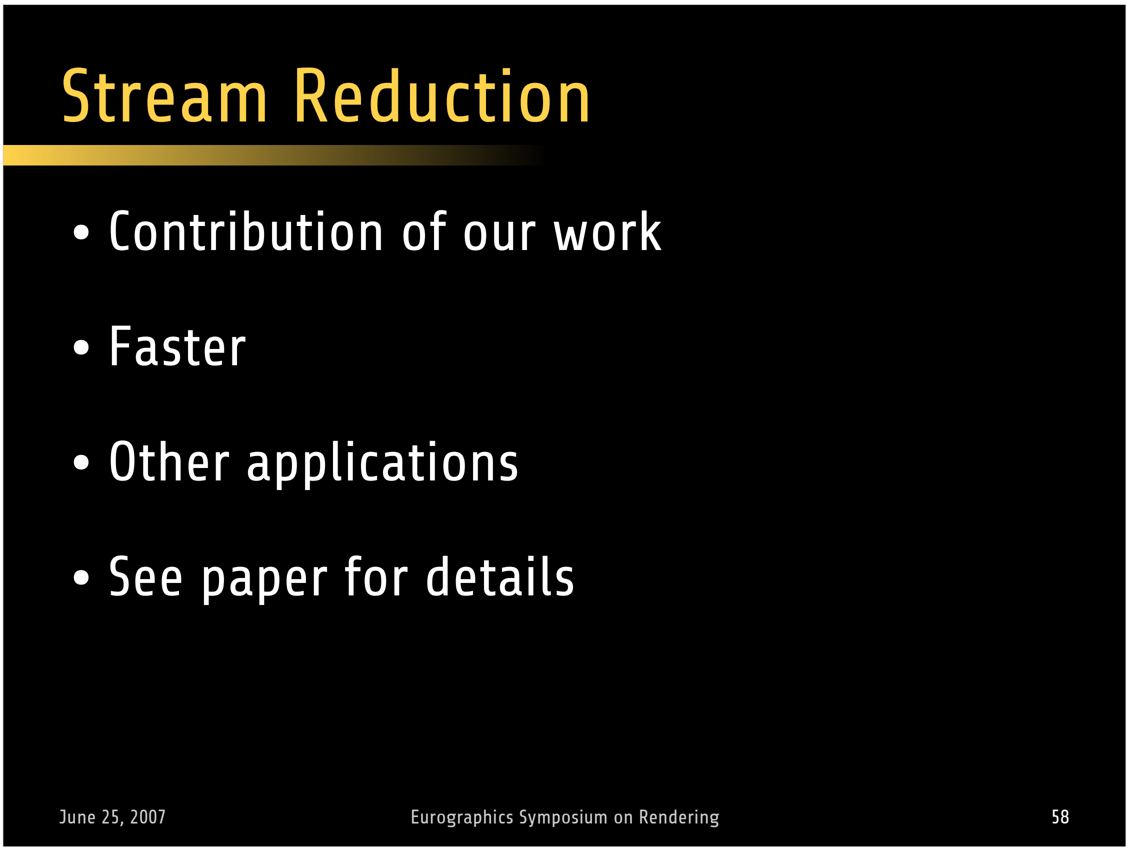
NULL correspond to a black texel on my figure.



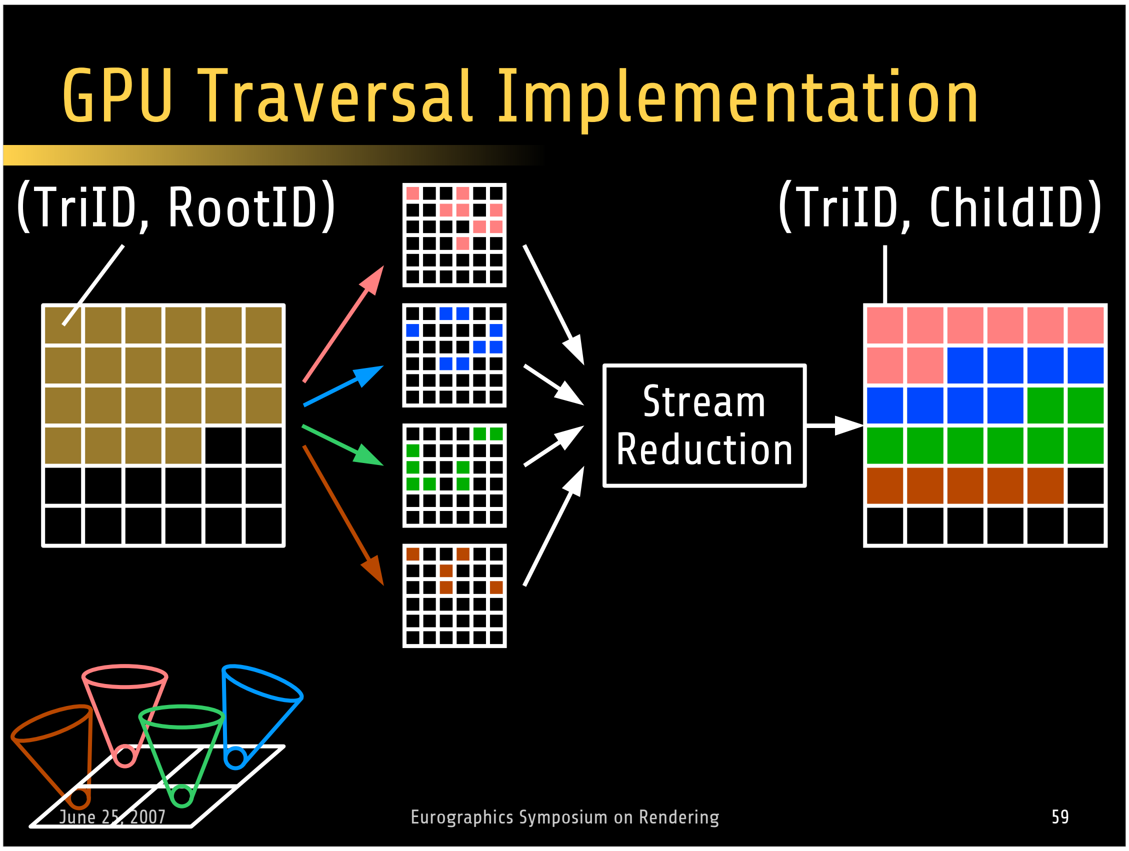
We use a stream reduction pass to merge these four textures into one, removing all the empty texels.

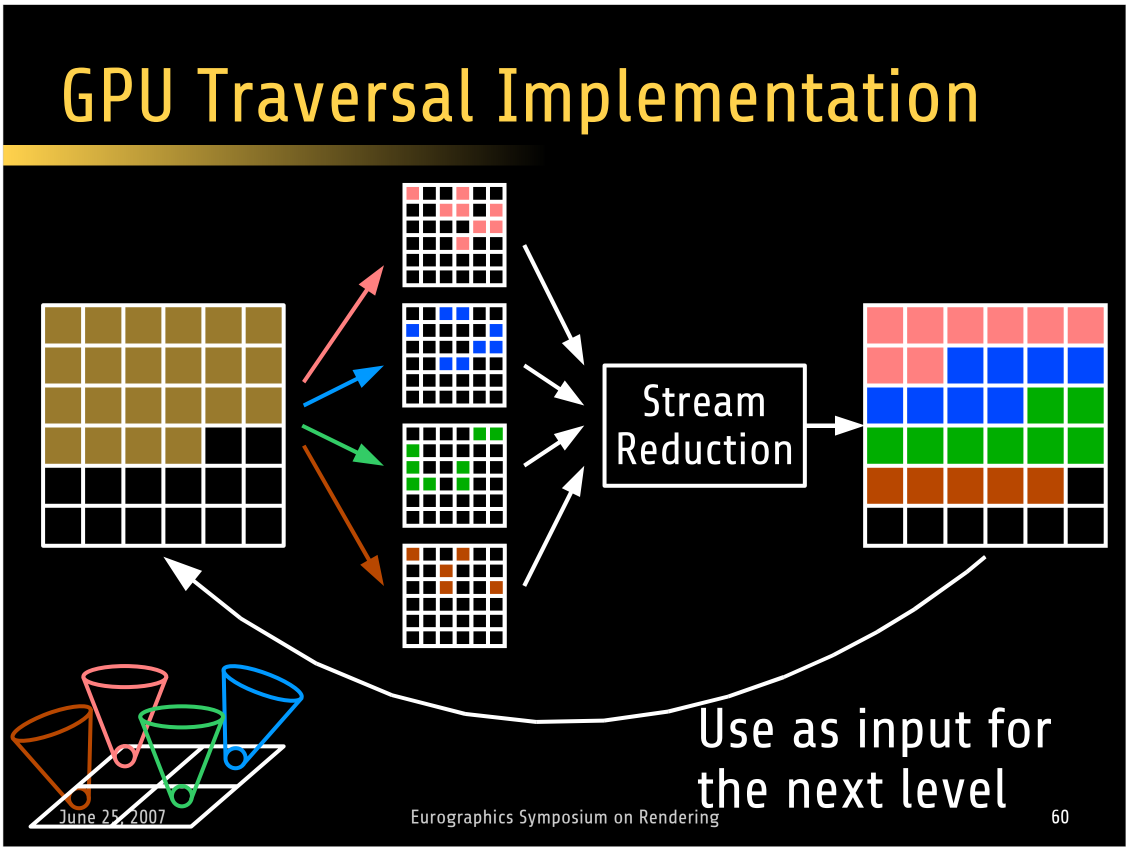
Removing the empty texels is essential: without it, the complexity would be exponential, as the number of pairs would be multiplied by four at each level of the hierarchy.

Stream reduction is not trivial and requires several passes in itself.

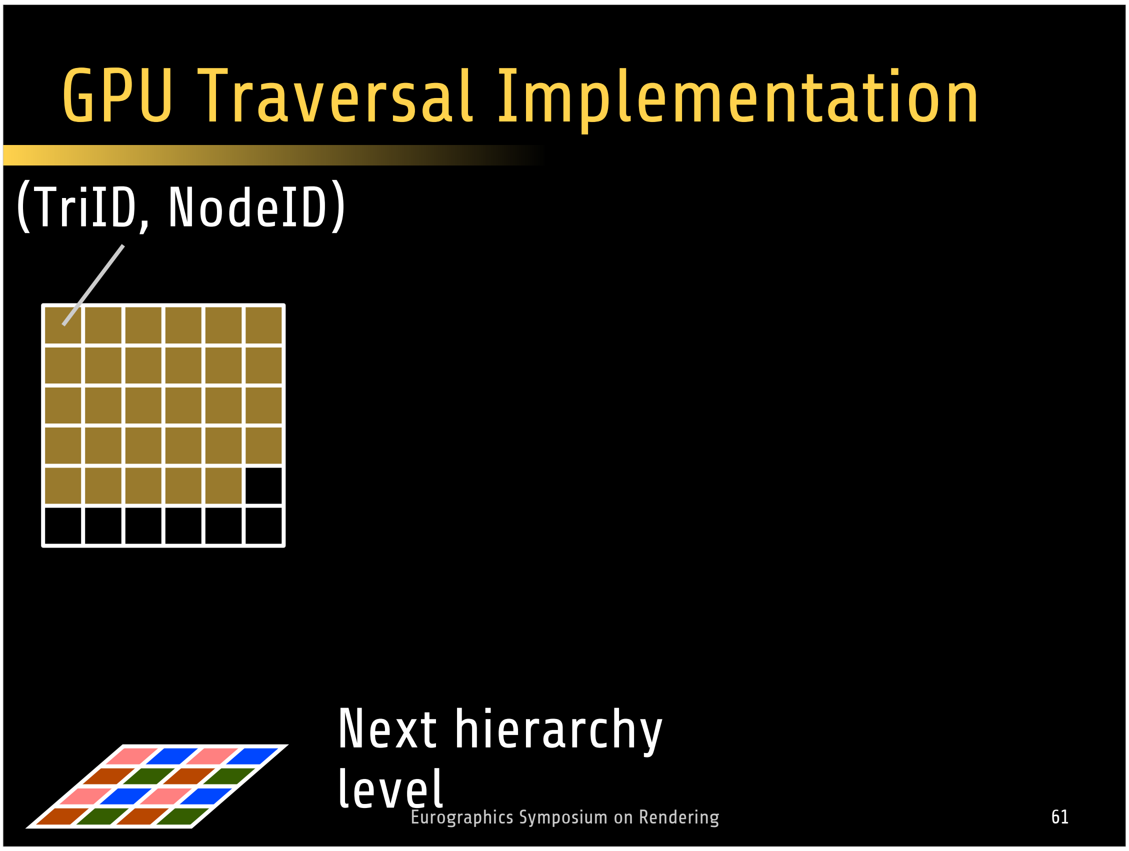


The stream reduction method we developped:
is a contribution of our work
is faster than previously known techniques
has other application beside this particular ray tracing algorithm


GPU Traversal Implementation
(TriID, RootID)
(TriID, ChildID)
Stream Reduction



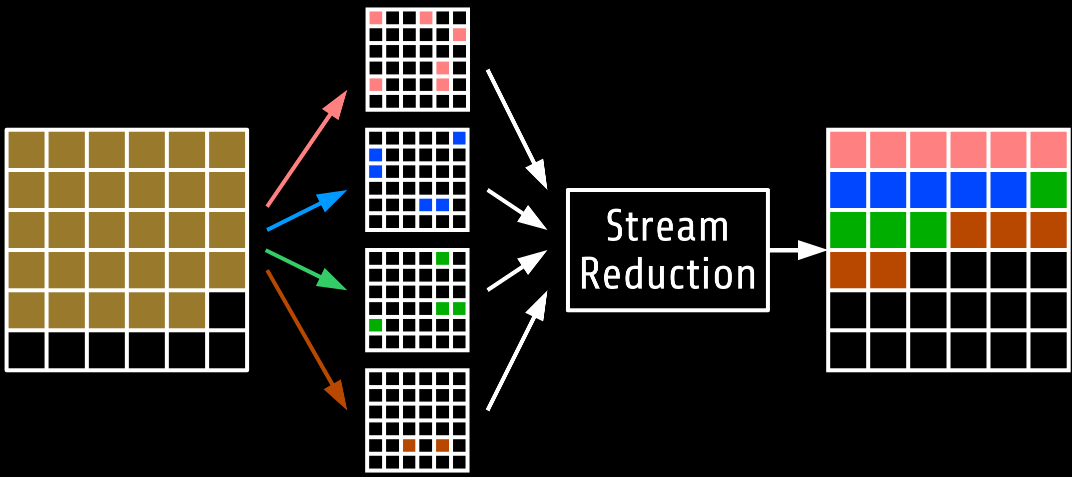
The resulting texture is used as input for the next level of the hierarchy.

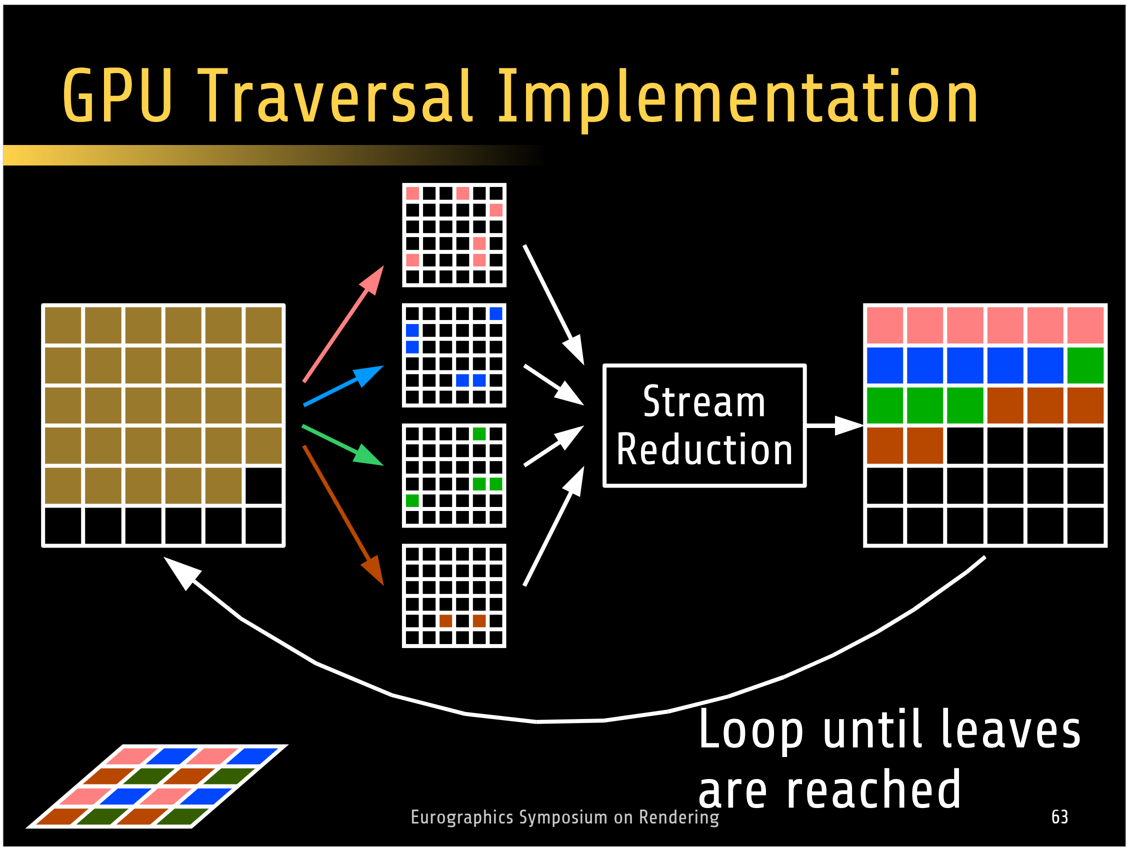


The next hierarchy level is processed in a new pass.

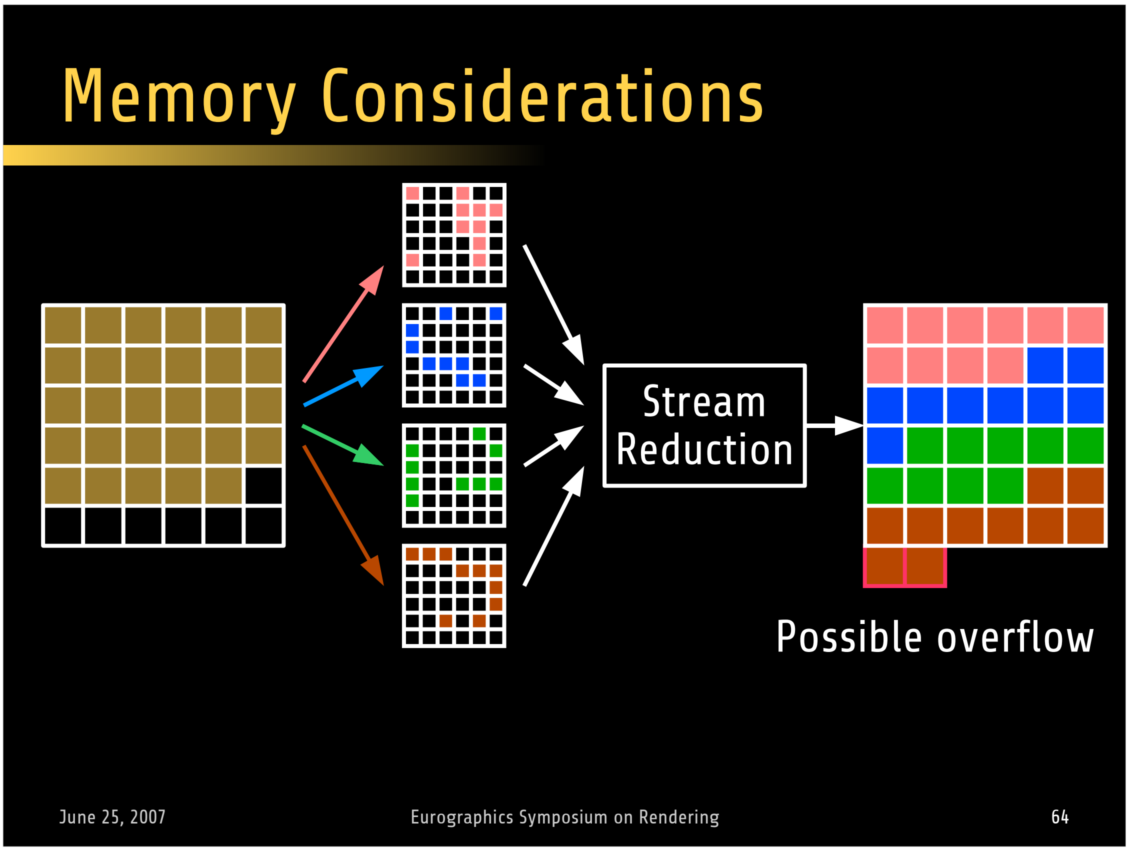
A triangle can now appear several times, paired with different nodes.

GPU Traversal Implementation





Loop until leaves are reached and we have (Triangle,Ray) pairs



On GPU, all pairs have to fit in one texture of fixed size.

As you can see on the right, memory overflow can happen.

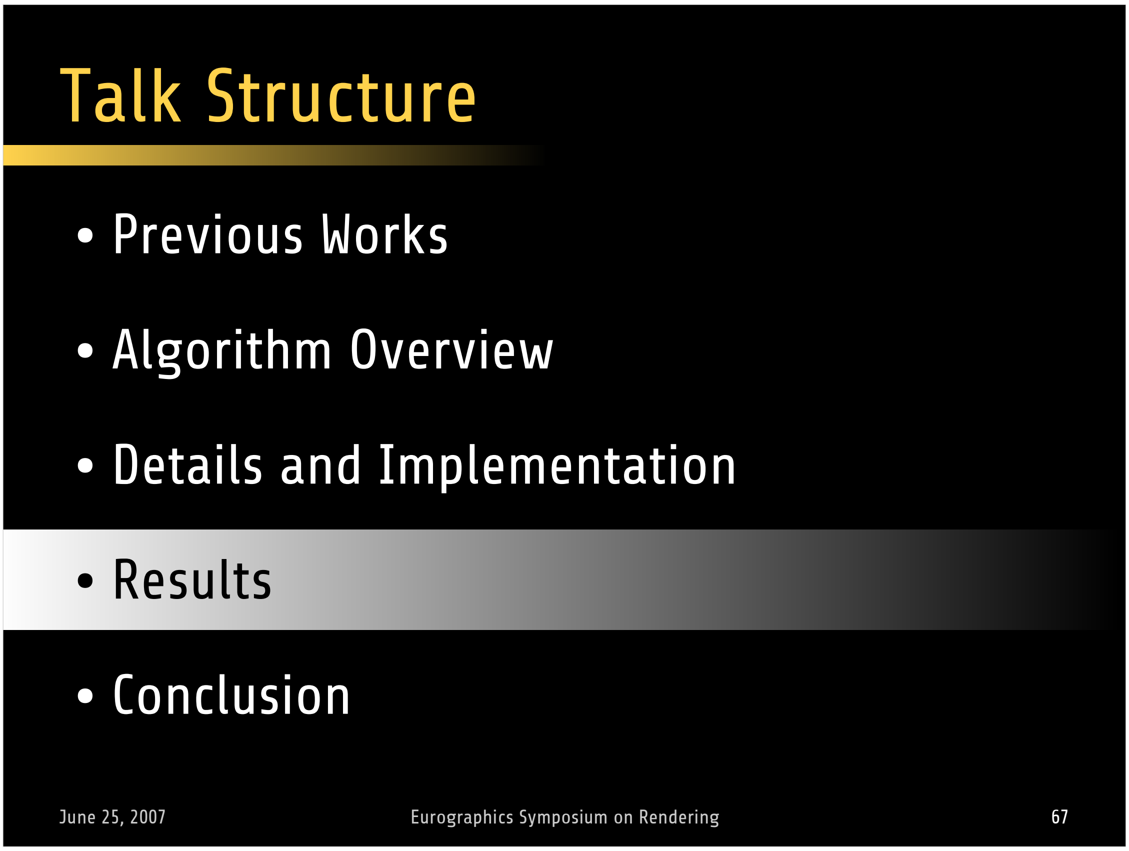
In this case we have designed a workaround.

Memory Considerations

- Simple workaround:
  - Subdivide scene in batches
  - Process the batches one after the other
  - Combine the results
- Constant overhead per batch (≈ 30ms)
- Allows large scene rendering too !
  - Even if it does not fit into GPU memory

Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Conclusion



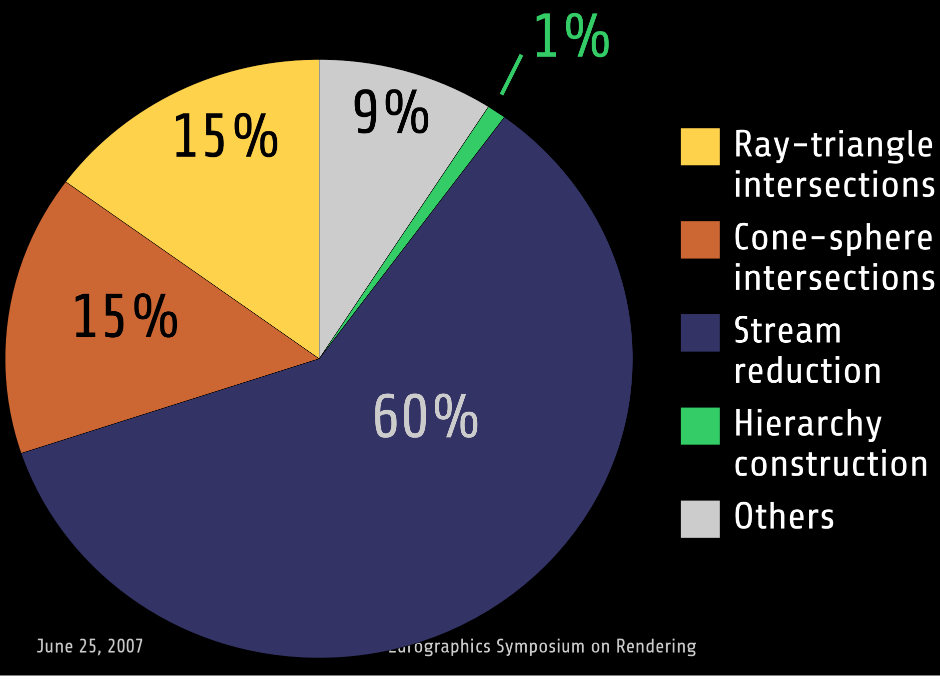
Lets move to the results part now.

Hardware

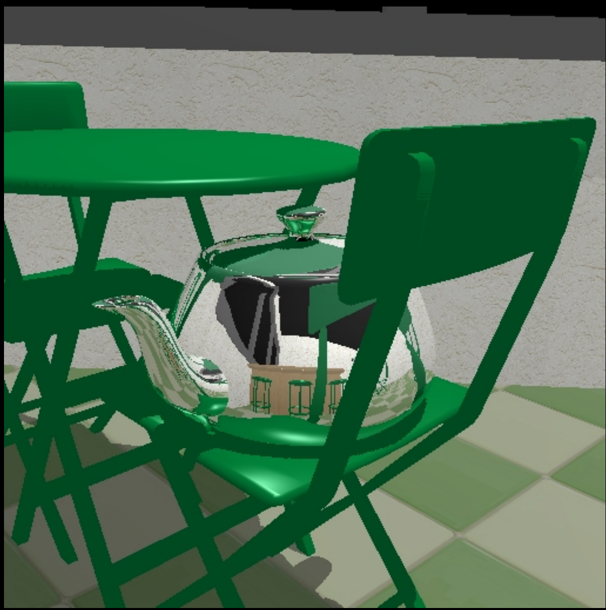
- GPU: GeForce 8800 GTS
  - 640 Mo RAM
- CPU: Intel Pentium 4
  - 3 GHz
  - 2 Go RAM

All the result in this talk were computed using ...

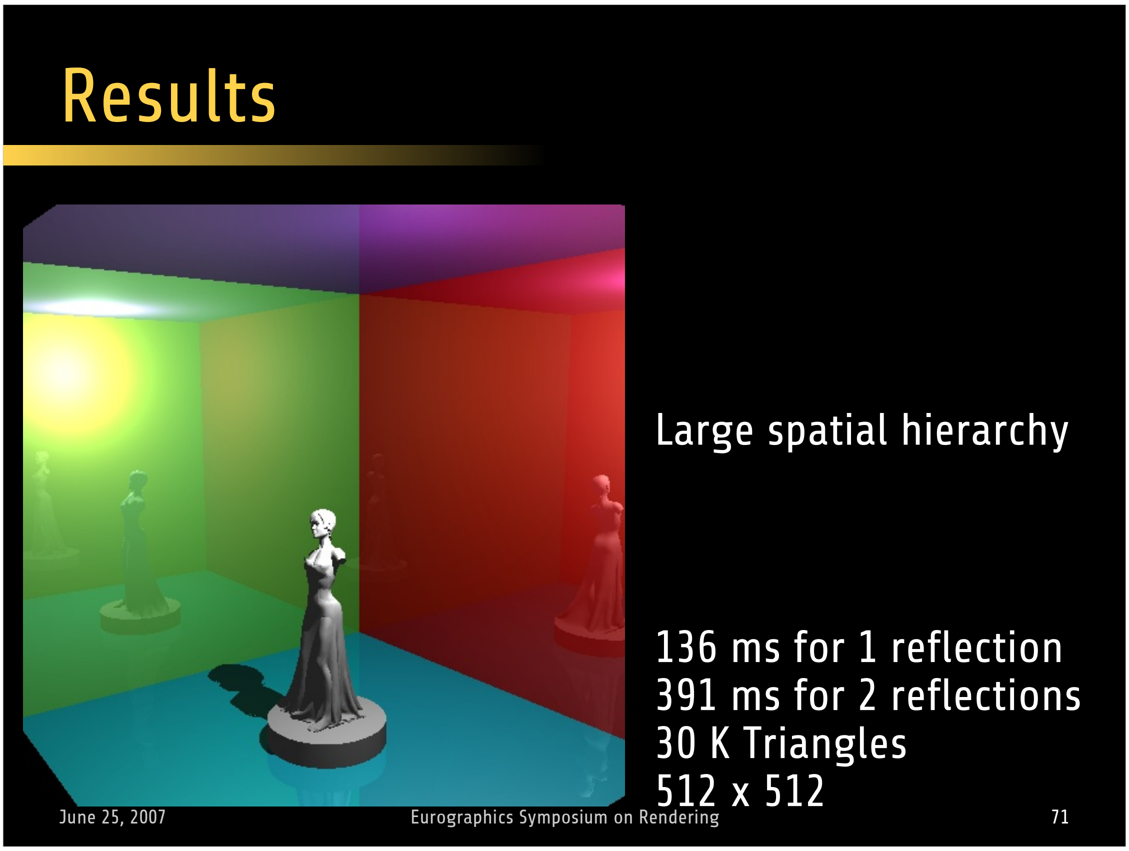
Time Repartition



Results



79 ms
20 K Triangles
512 x 512



On this picture, the reflector covers the whole scene.

Thus, the higher level of the hierarchy have a very large spatial extent and are less efficient.

Lower levels remain efficient though.

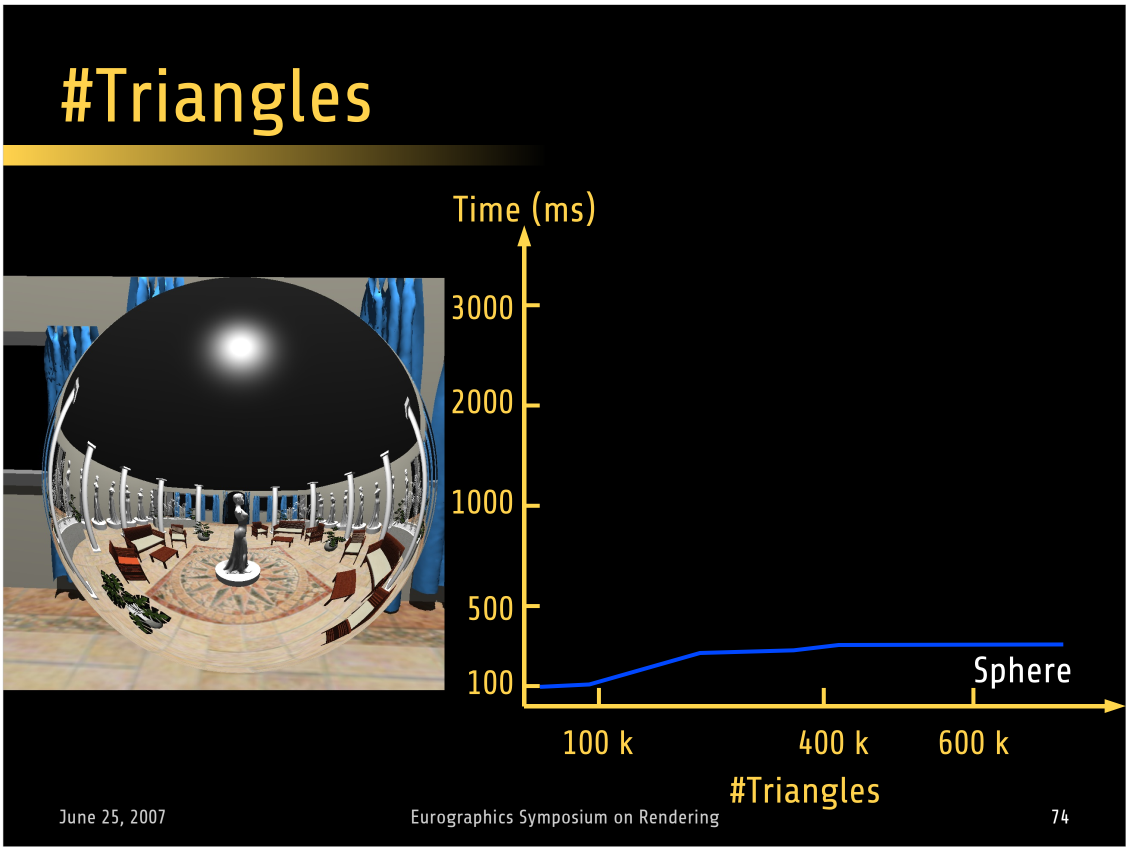
Results

Several reflectors:
Discontinuities in the top levels of the hierarchy

302 ms
83 K Triangles
512 x 512

Results

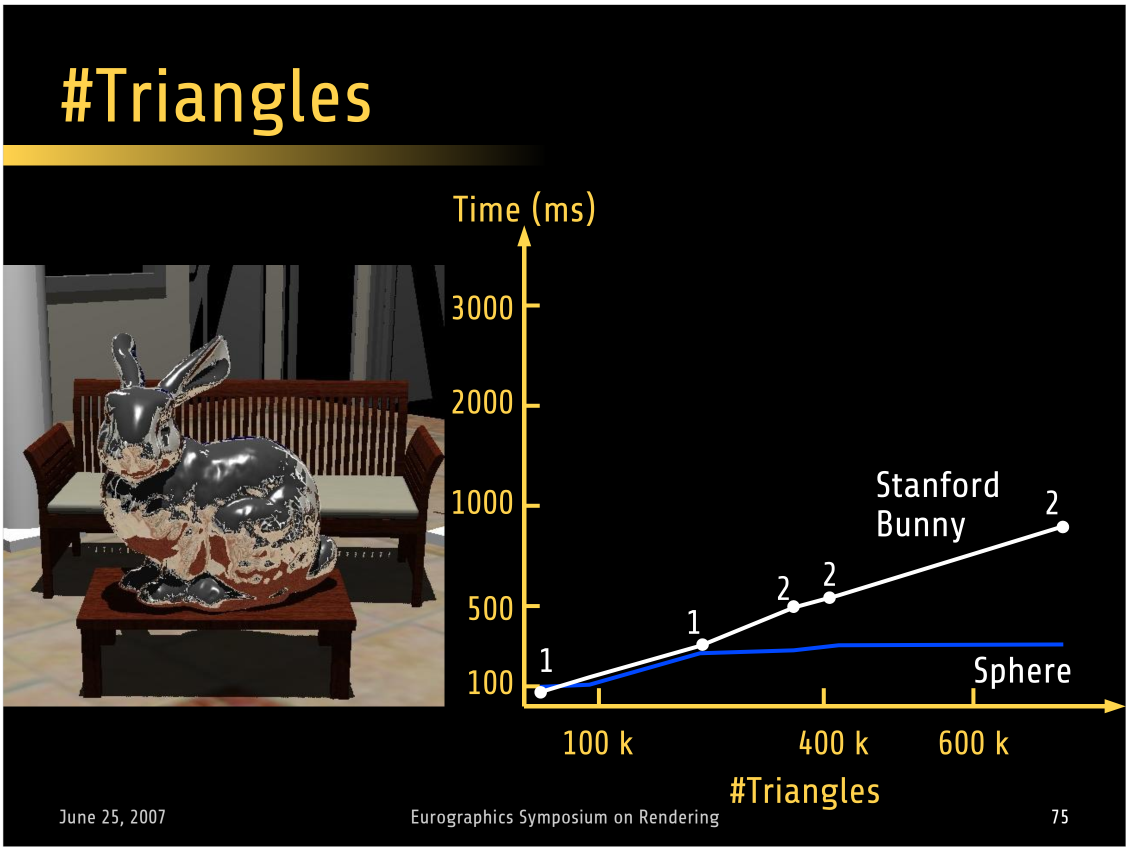
Video: Dynamic Scene



We put a reflecting sphere in the patio scene, and increased the number of triangles by adding furnitures like chairs, curtains or plants.

Here is the rendering time for 1 frame.

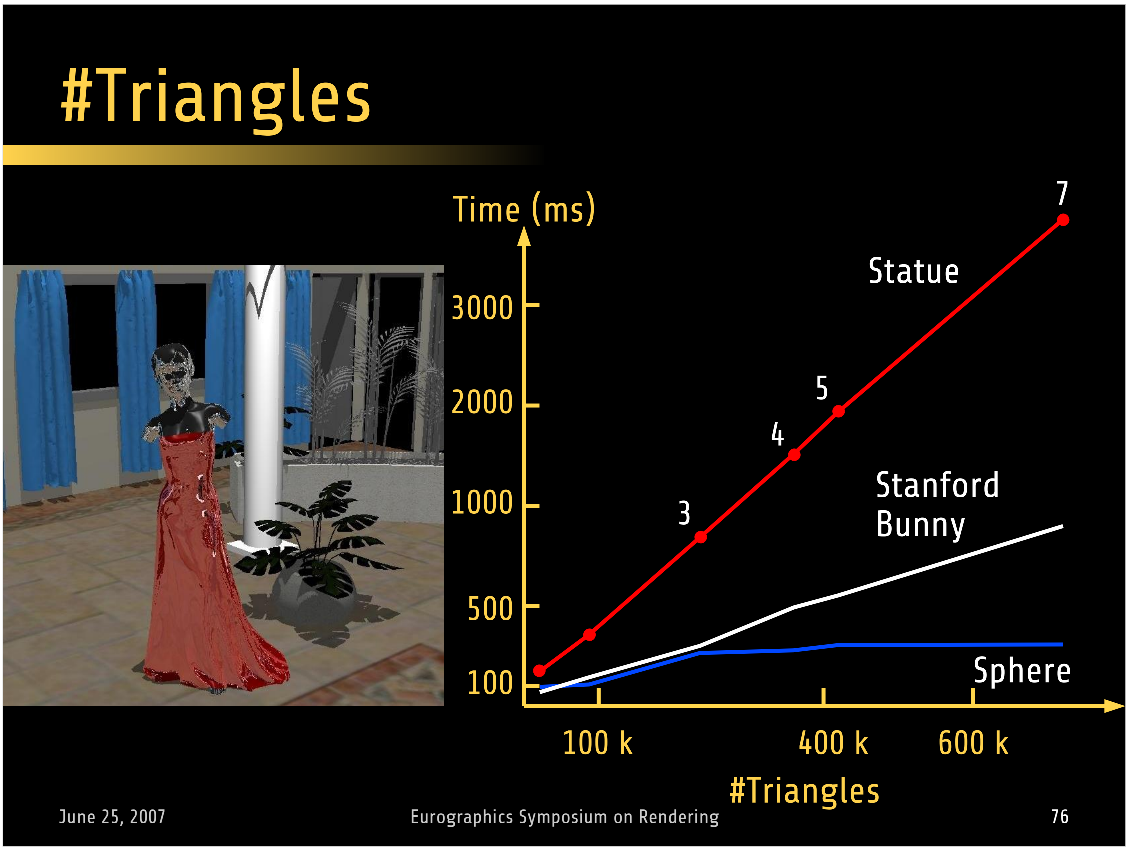
300 ms for more than 700 K triangles



I put the number of batches we used. Here, for more than 400k triangles, we had to split the scene in two batches to avoid memory overflow.

The slope is greater, because rays have less coherency.

Less than 1s for more than 700 k triangles.

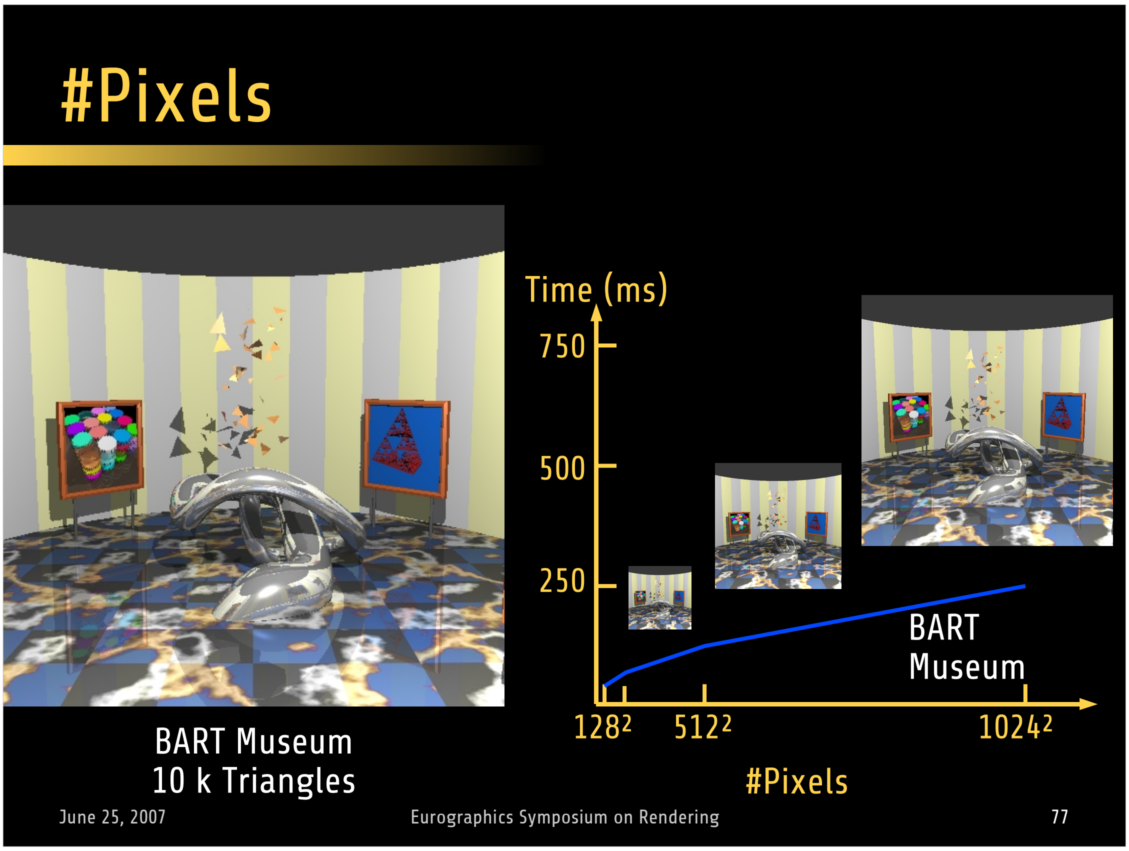


Model with a lot of discontinuities (hair, face, folds of the dress).

Around 4s, and we had to use up to 7 batches.

Our algorithm performs linearly with the number of polygons.

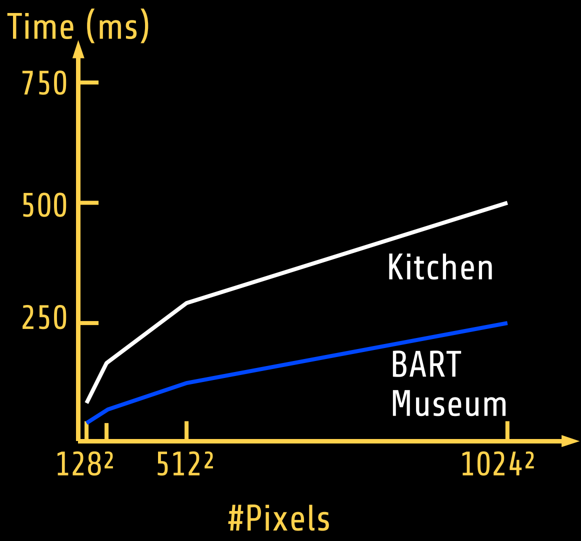
The slope is determined by the size and the shape of the specular objects.

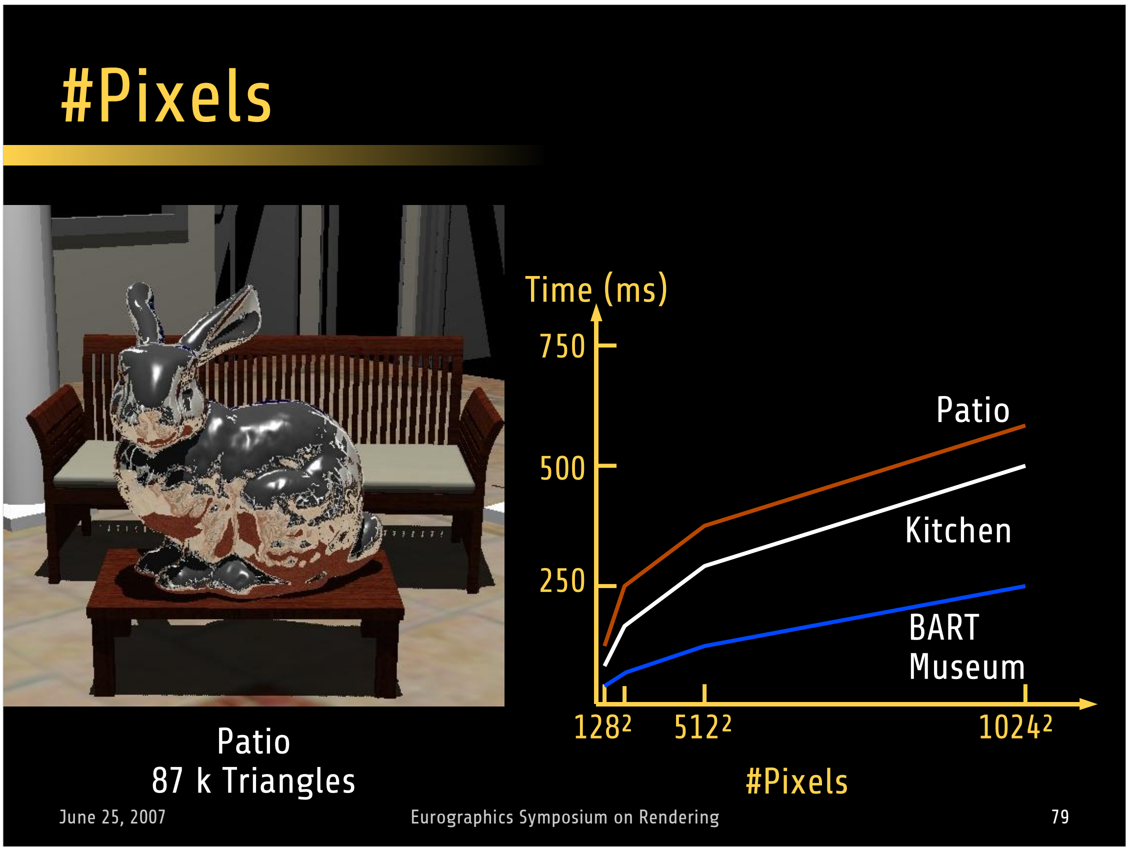


We render the same scene wirth different picture resolution.

#Pixels

Kitchen
83 k Triangles





You can see that the algorithm performs sub linearly with the number of pixels.

The time is sublinear, but unfortunately the memory footprint is linear !

Thus we were not able to render larger picture than 1M pixels, because our GPU had not enough memory.

Large Scene

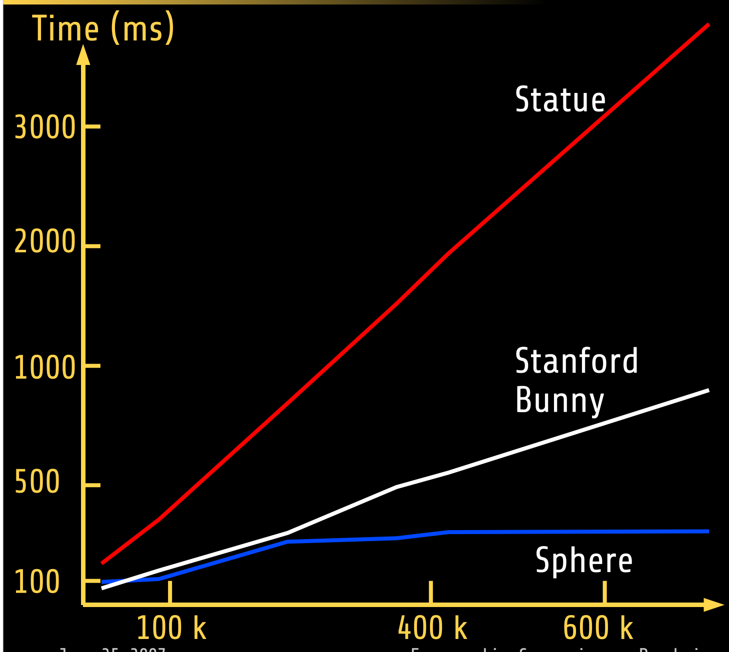
50 batches

18.5 s
2.3 M Triangles
512 x 512

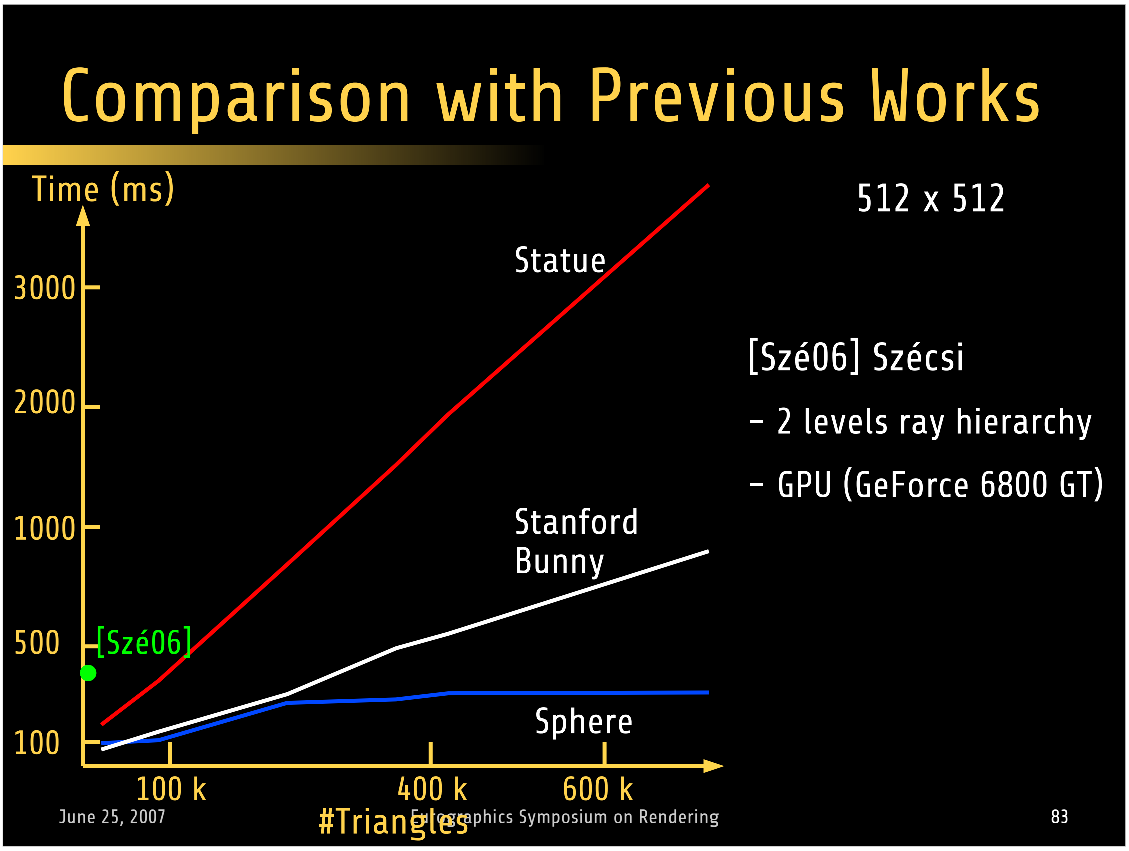
Comparison with Previous Works

- Very few papers report figures for both:
  - Dynamic scenes
  - Secondary rays
- We compared to papers that do at least one
- Hard to compare

Comparison with Previous Works

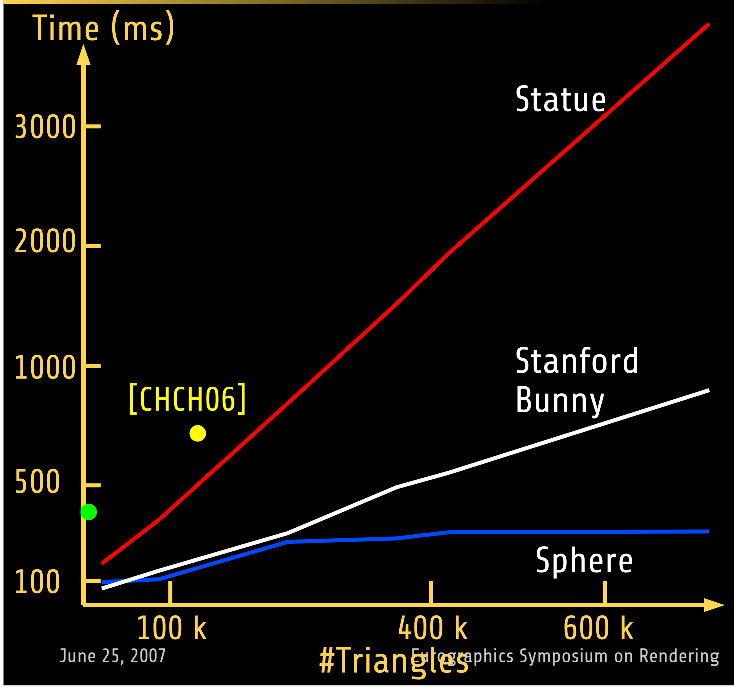


512 x 512



Comparison with Previous Works
Time (ms)
3000
2000
1000
500
100
[Szé06]
Statue
Stanford
Bunny
Sphere
100 k
400 k
600 k
#Triangles
512 x 512
[Szé06] Szécsi
– 2 levels ray hierarchy
– GPU (GeForce 6800 GT)
June 25, 2007
Eurographics Symposium on Rendering
83

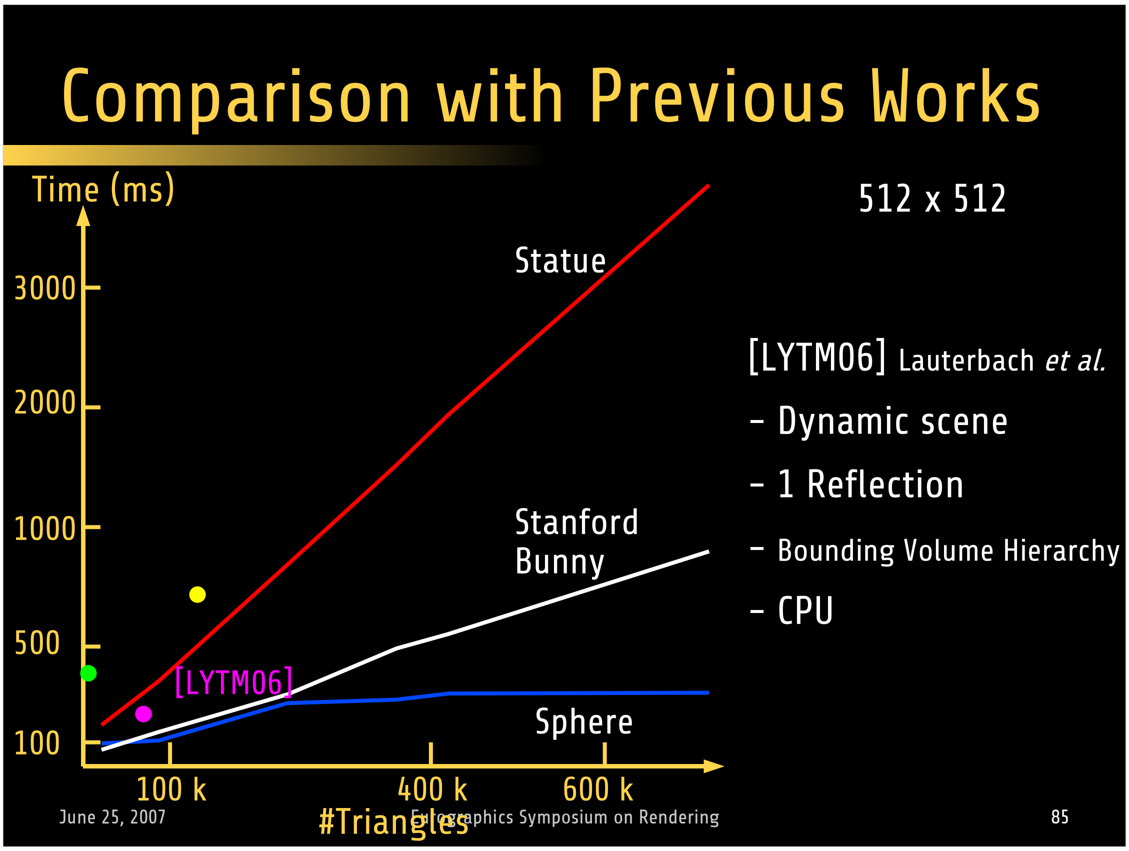
Comparison with Previous Works



512 x 512

[CHCH06] Carr *et al.*

- Dynamic scene
- Primary rays
- Geometry images
- GPU (GeForce 6800 Ultra)

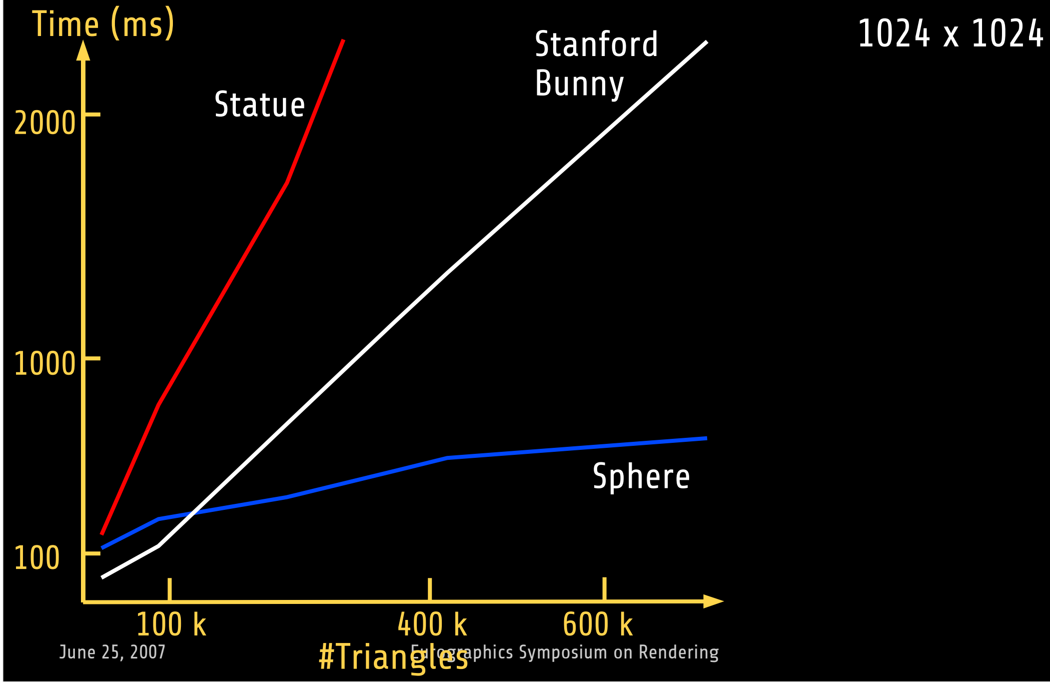


Lauterbach and others do both dynamic scene and secondary rays using a bounding volume hierarchy on the CPU.

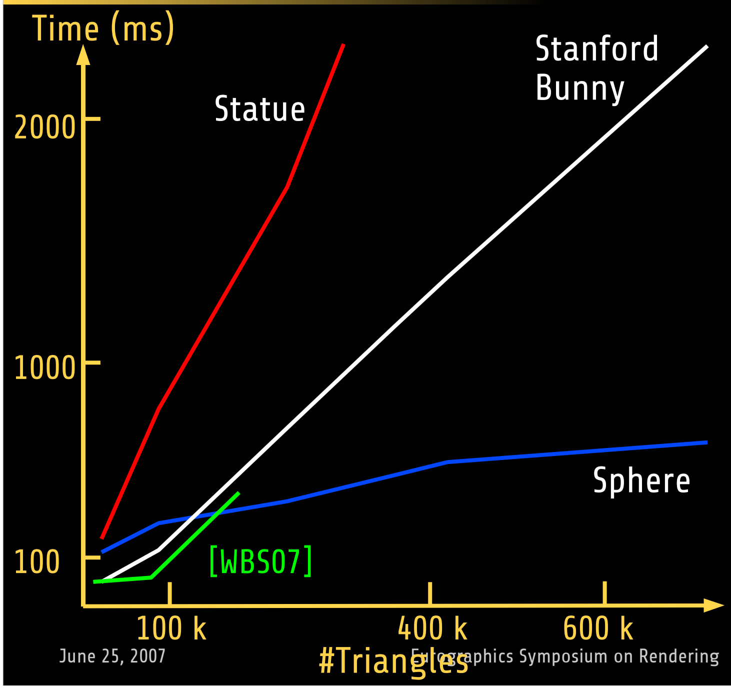
On their scene, they are faster than our statue scene, but slower than our bunny scene.

We also provide figures for much larger scenes.

Comparison with Previous Works



Comparison with Previous Works

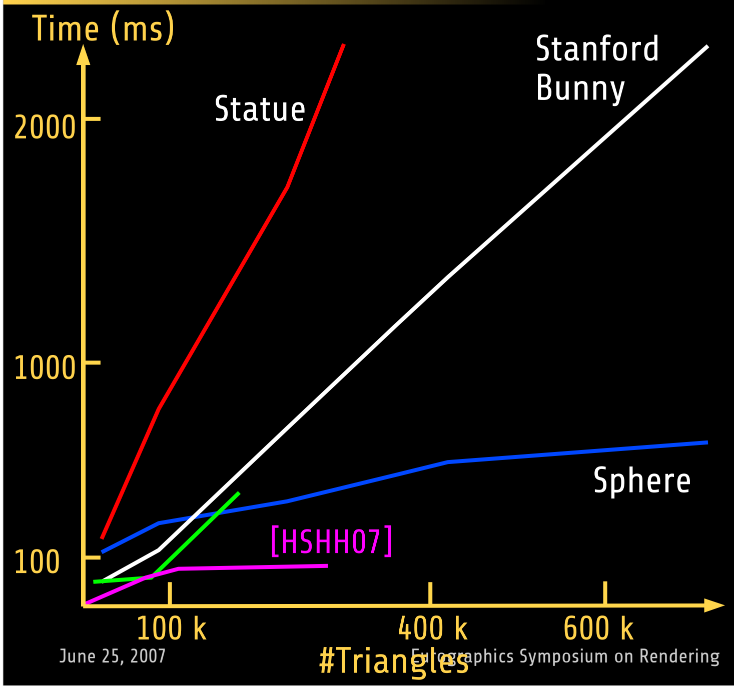


1024 x 1024

[WBS07] Wald *et al.*

- Dynamic scene
- Primary rays
- Bounding Volume Hierarchy
- CPU

Comparison with Previous Works



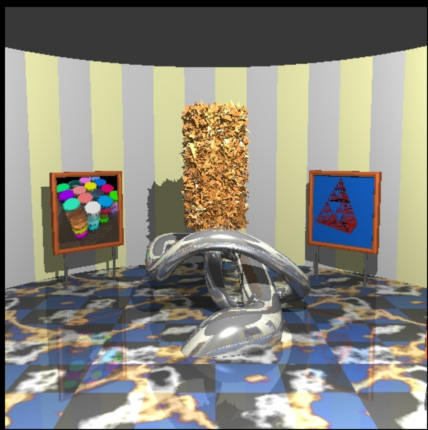
1024 x 1024

[HSHH07] Horn *et al.*

- Static scene
- 1 Reflection
- Kd tree
- GPU (ATI X1900 XTX)

Comparison with Previous Works

- [WK06] Wächter and Keller, Bounding Interval Hierarchy, CPU
  - Dynamic scene + secondary rays



Reflection + shadow

	Us	[WK06]
Museum3 10 k Tris	289 ms	1282 ms
Museum8 76 k Tris	3330 ms	2040 ms

This the museum scene from the BART benchmark.

It is designed to break hierarchies and is difficult to render fast.

High occlusion kills us when there are too many triangles in the soup.

Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Conclusion

Talk Structure

- Previous Works
- Algorithm Overview
- Details and Implementation
- Results
- Conclusion

Pros and Cons

- Pros
  - Dynamic scenes
  - No precomputation
  - Scales well with resolution
  - Large scenes support
- Cons
  - No early ray termination
  - Linear in #Triangles
  - Ray hierarchy looser than scene hierarchy (ray coherency)

Contributions

- Interactive ray tracing algorithm
  - Secondary rays
  - Dynamic scenes
  - No precomputation
  - Scales well with resolution
  - Large scenes support
- Faster stream reduction

A faster stream reduction method that has applications outside of our ray tracing algorithm.

Future Works

- Cone tracing:
  - anti aliasing
  - soft shadows
  - glossy reflections
- Scene Structure
  - Regular grid, bounding volumes
  - More complex structure ? Hierarchy ?

Thank You