# KD Tree Path Tracer Optimization

**Rony Edde**

- Shooting rays through a complex scene with a large number of polygons can be heavy and exponentially slow even on the GPU. Axis Aligned Bounding boxes help improve performance, hover this still doesn't solve the problem when rays intersect a complex object with millions of triangles.

# KD Tree Path Tracer Optimization

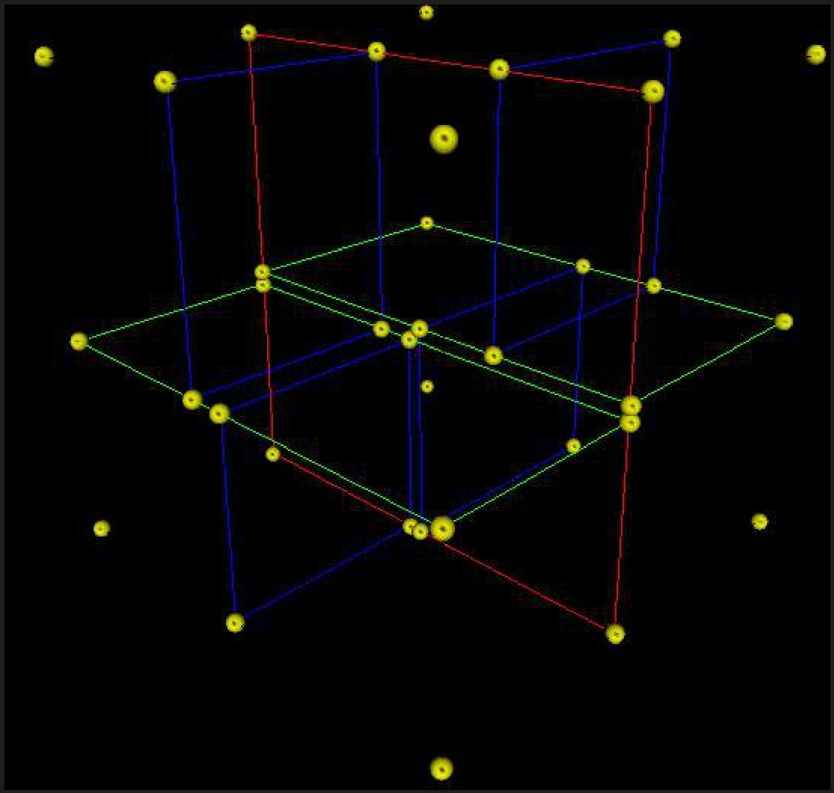
Rony Edde

- Using a K-D Tree to represent the triangles reduces the overall complexity of the lookup and the time needed to find the intersecting triangles. There is a cost of computation for generating the KD-Tree but in most cases the scene is mostly static geometry which should benefit from this data structure.

# KD Tree Path Tracer Optimization

Rony Edde

- A KD-Tree is a data structure that can help solve this problem by dividing the space into sub-spaces and checking for intersections within the subdivided regions instead of the entire geometry. (image from of Wikipedia)



# KD Tree Path Tracer Optimization

Rony Edde

Concept.

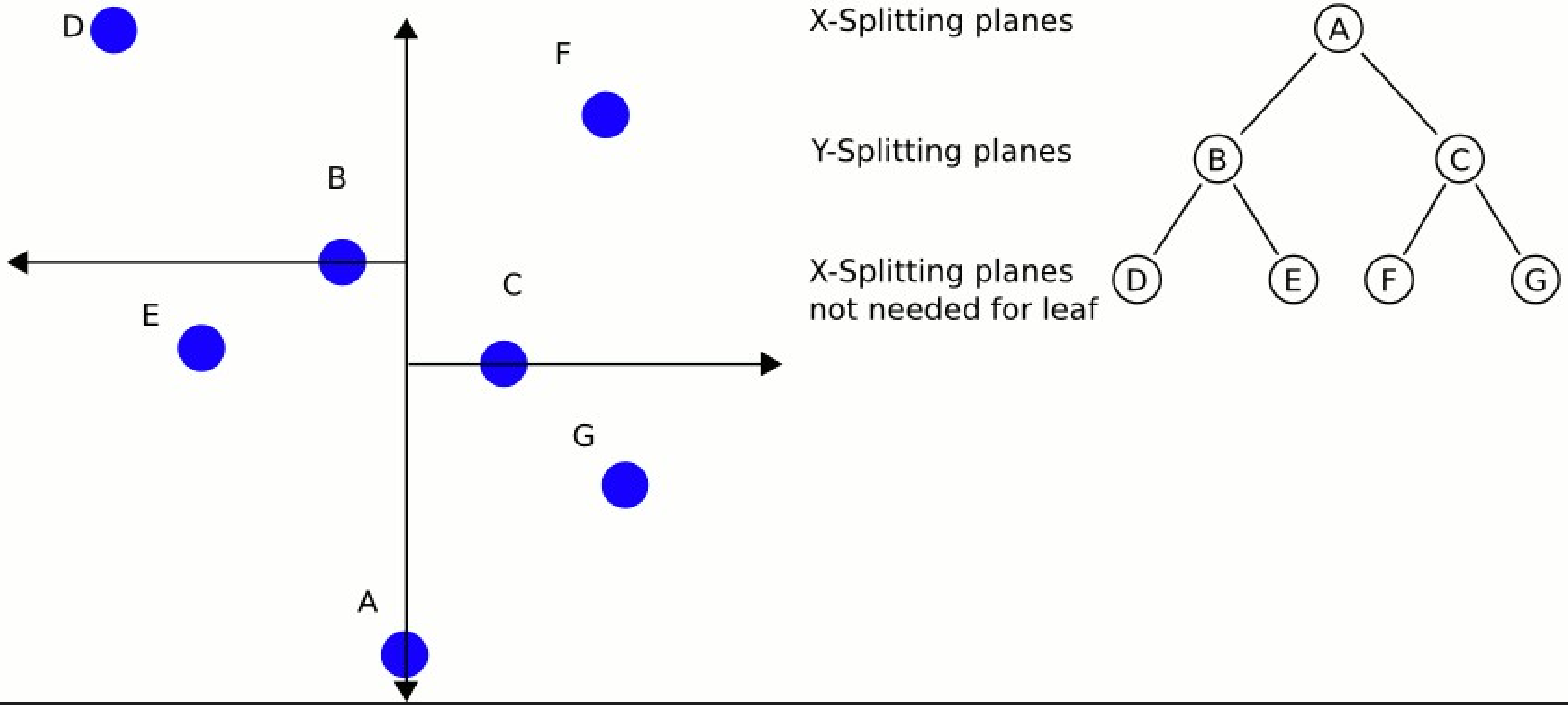
In order to build a KD-Tree, I start by generating a bounding box that encompasses all elements (triangles in our case). I then split on one axis. Every consecutive level will be split on the second axis. In 2D there are 2 axis splits, 3 in 3D. Every split divides the space in 2 sections with new bounds. Triangles that belong to the left side are moved to the left node, right otherwise. This partitions the space into a tree that can then be traversed by only checking if a hit occurred with the boundary. This makes a KD-Tree efficient for complex geometry.

# KD Tree Path Tracer Optimization

Rony Edde

Concept.

An illustration of a 2D KD-Tree traversal with points.



# KD Tree Path Tracer Optimization

Rony Edde

A few problems to solve

- 1- GPU memory limit.
- 2- Streaming solution?
- 3- No support for recursion in CUDA.
- 4- Very tight schedule…

## KD Tree Path Tracer Optimization

Rony Edde

A few problems to solve

- 1- GPU memory limit.

GPUs don't have enough memory to compete with CPUs and a typical path tracer loading millions of triangles can easily crash the graphics card. Just accessing the entire geometry without an optimized data structure can be close to impossible without resorting to optimizations. However, even after optimizations, this can remain a major issue with high risks of overwhelming the GPU memory and resulting in a crash.

# KD Tree Path Tracer Optimization

Rony Edde

A few problems to solve

- 2- Streaming solution?

In order to solve the memory limitation, we can resort to streaming. If the object loaded has 1 million triangles, it would require 192MB of GPU memory on a 64 bit machine. In today's applications and games 1 million polygons is considered low resolution. We can see how this quickly becomes an issue.

## KD Tree Path Tracer Optimization

Rony Edde

A few problems to solve

- 2- Streaming solution?

Streaming alone, however presents additional complexities for optimization. KD-Trees will need to be adaptive and fast in order to account for this. Another option is to split the complexity in sub-trees in order to fit in memory, thus reducing the impact caused by streaming. This could have an impact on the benefit of using KD-Trees and can also present issues when rendering double sided geometry.

# KD Tree Path Tracer Optimization

Rony Edde

A few problems to solve

- 3- No support for recursion in CUDA.

The lack of recursion support in CUDA presents a challenge for any tree like data structure.  Recursion is at the heart of any tree data structure.

# KD Tree Path Tracer Optimization

Rony Edde

Solving the lack of recursion

- Implementing the KD-Tree without recursion involves a rewrite. The initial implementation was made recursively. Once the tree was built and the data is correctly generated, I run tests to make sure that everything works.

# KD Tree Path Tracer Optimization

Rony Edde

Solving the lack of recursion

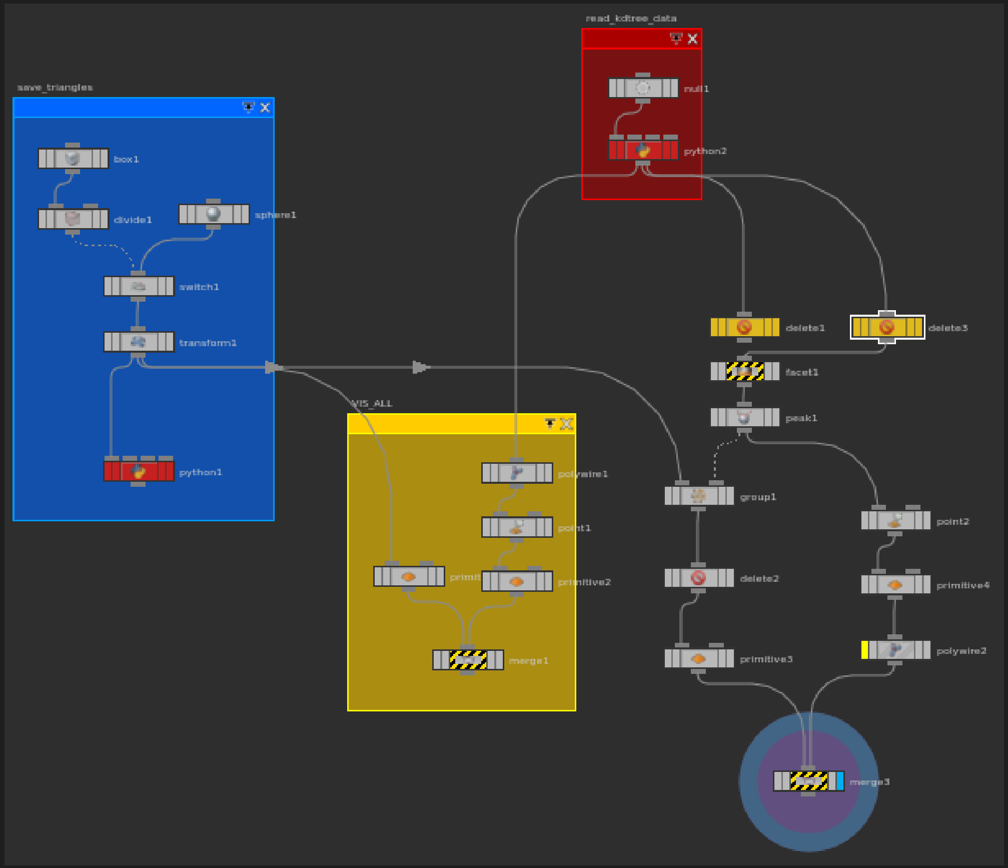
- Testing the tree using print statements is tricky so I've resorted to using Houdini to help prototype and display the KD-Tree before starting the GPU binding.
- The following slide shows a Houdini tool that was developed for this purpose.

# KD Tree Path Tracer Optimization

Rony Edde

Solving the lack of recursion

- Houdini to the rescue.

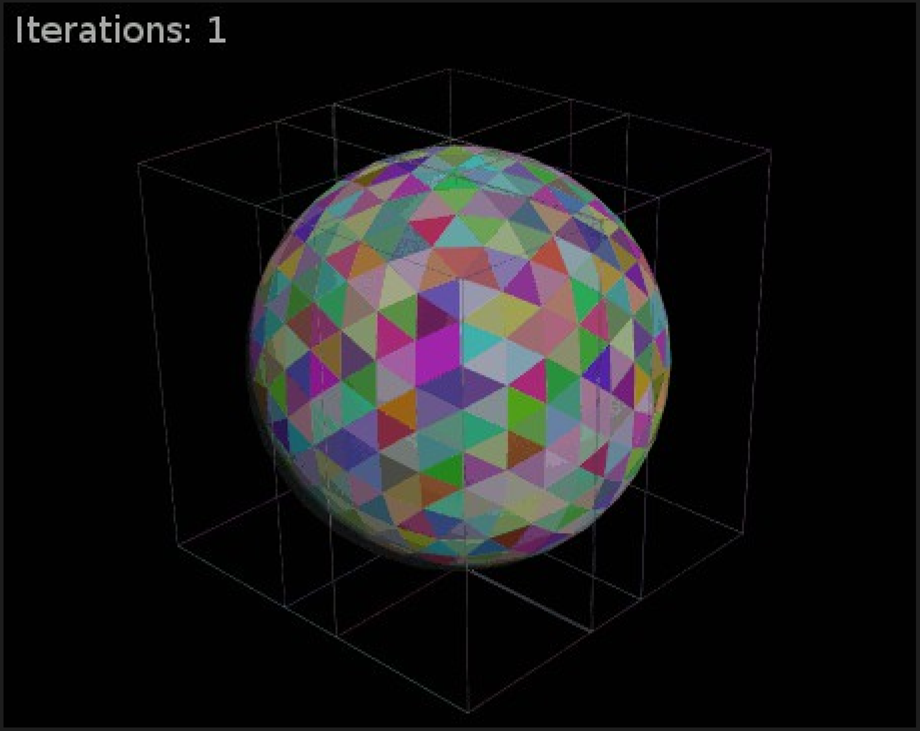


# KD Tree Path Tracer Optimization

Rony Edde

Solving the lack of recursion

- Initial results with offsets

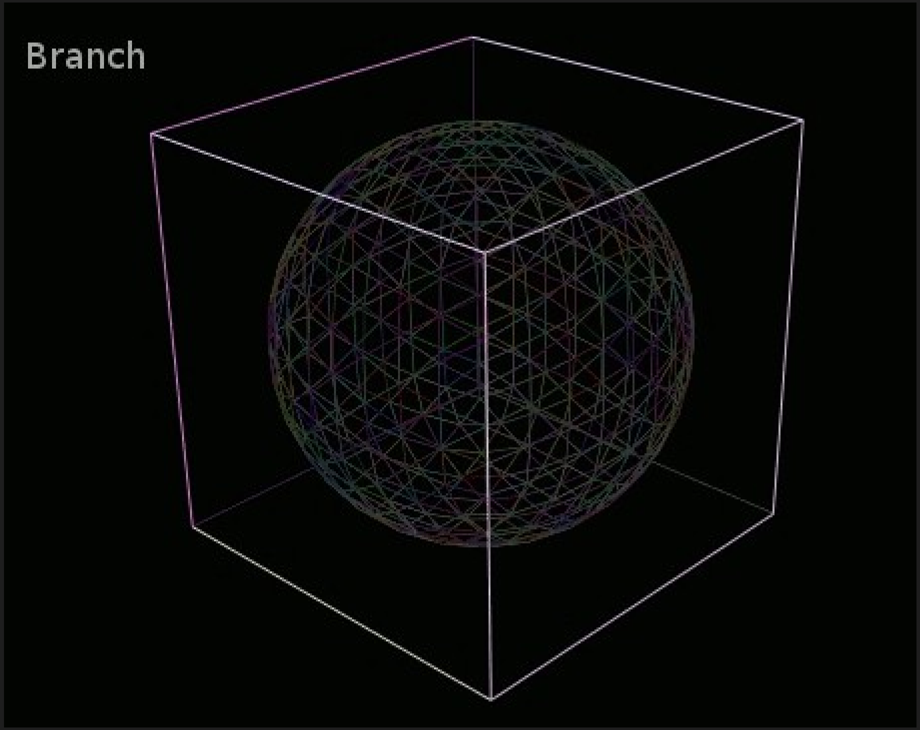


# KD Tree Path Tracer Optimization

Rony Edde

Solving the lack of recursion

- Initial results with offsets

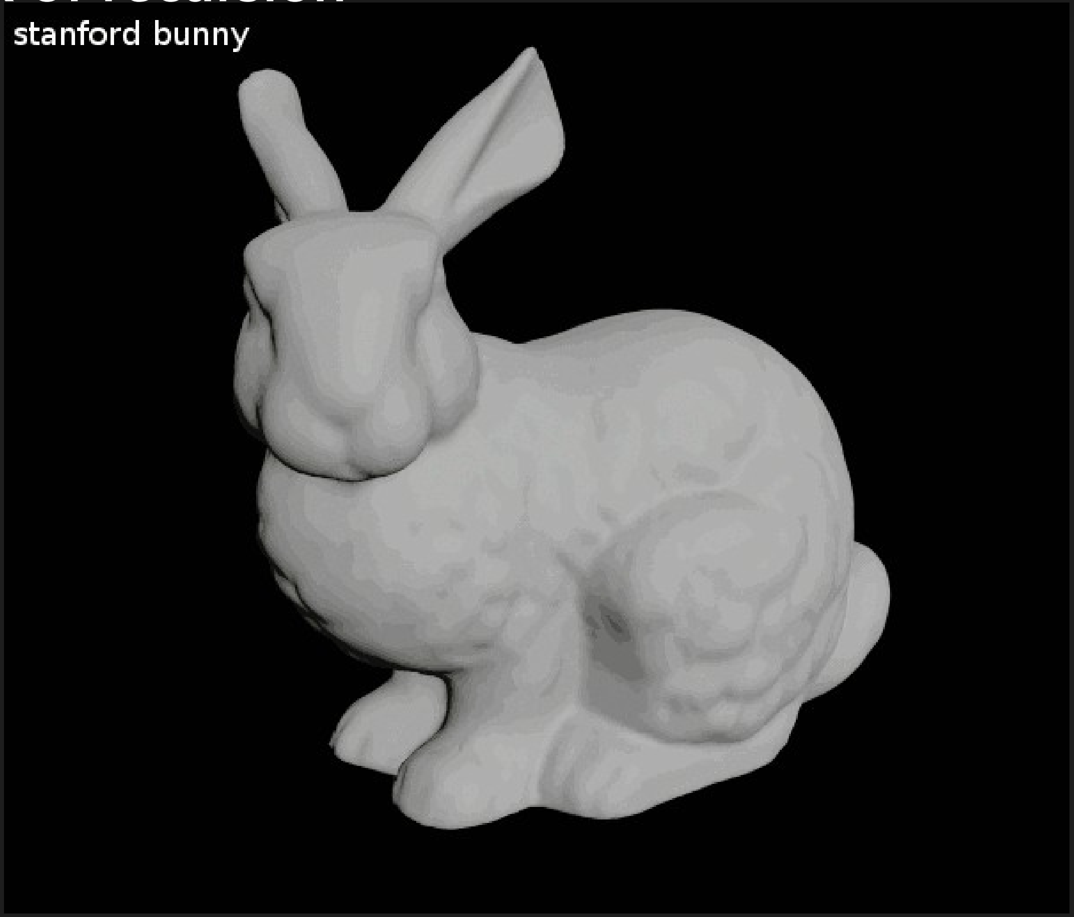


## KD Tree Path Tracer Optimization

Rony Edde

Solving the lack of recursion

- Initial results



# KD Tree Path Tracer Optimization

Rony Edde

Solving the lack of recursion

- The initial results looked correct but looking closely at the boundaries, we can see an overlap in the regions.  This was due to the bounds being resized after the split.  The debug view was very helpful in identifying problems such as this.

# KD Tree Path Tracer Optimization

Rony Edde

Solving the lack of recursion

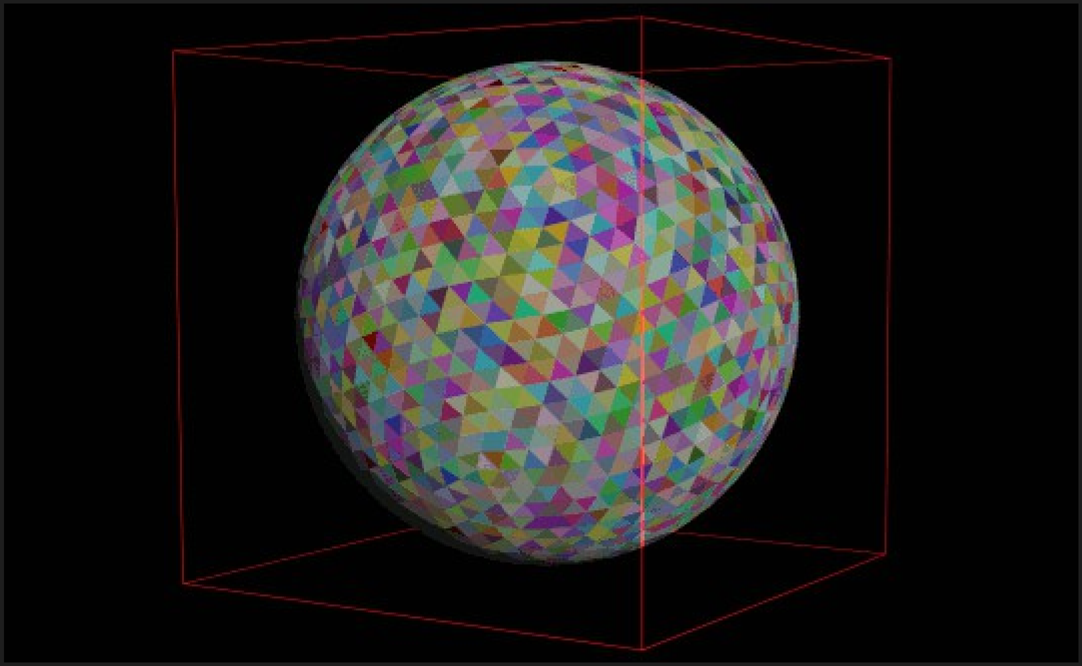
- Corrected bounds.  Resolved overlap.

# KD Tree Path Tracer Optimization

Rony Edde

Solving the lack of recursion

- Corrected bounds.  Resolved overlap.



## KD Tree Path Tracer Optimization

Rony Edde

Solving the lack of recursion

- Now that I have a semi working data structure, I start the flattening of the data so that I can traverse the tree without using recursion.

# KD Tree Path Tracer Optimization

Rony Edde

Solving the lack of recursion

- Traversing the tree was initially done using attributes that were stored in the node.  Visited was a variable added to the nodes that, once a node is visited, is set to true.
- This presented problems because it wasn't thread safe and the node data could get modified by other threads creating a race condition.

## KD Tree Path Tracer Optimization

Rony Edde

Solving the lack of recursion

- The solution was to flatten the nodes and the triangles that belonged to the leafs and using indexing in order to traverse the tree.
- The following slide illustrates this.

# KD Tree Path Tracer Optimization

Rony Edde

Solving the lack of recursion

- 1- create the KD-Tree.
- 2- save parent / left and right child indices.
- 3- save all triangles into a new array and keep offset stored.
- 4- save all nodes into a new array.
- 5- use index mapping to track visited nodes.

# KD Tree Path Tracer Optimization

Rony Edde

Solving the lack of recursion

- Ray hit tracking:

1- set a visited array for all nodes. Set all values to false.

2- Using a while loop, we check  if the ray hit the root.

3- If we hit, we set the node index lookup to true.

else we set the current node to the parent

4- If we reached a leaf, check collision with its triangles.

and update hit distance.

- 5- repeat until node parent is -1 or all nodes are visited.

## KD Tree Path Tracer Optimization

Rony Edde

Implementation issues.

- While this implementation works and has been tested, the GPU implementation failed initially due to one main problem.
- CUDA's memory management is rigid, meaning that one cannot simply allocate on the GPU without a major performance hit.
- KD-Tree storage must be hidden and only the data can be visible to CUDA.  The algorithms must be able to handle dynamic sizes without crashing.

## KD Tree Path Tracer Optimization

Rony Edde

Implementation issues.

- After several attempts that ended up crashing the GPU, the final solution was to lock the KD-Tree into the scene description and rewrite the geometry intersection algorithm while only exposing 2 types.  KDnode and KDtriangle.  This solved most of the problems that were crashing the GPU.
- Solving the non dynamic accessing of the data and the varying sizes was slightly less problematic.  The final decision was to resort to a fixed size representation of the indexing that was both small enough not to hinder performance and large enough to contain all the indices.

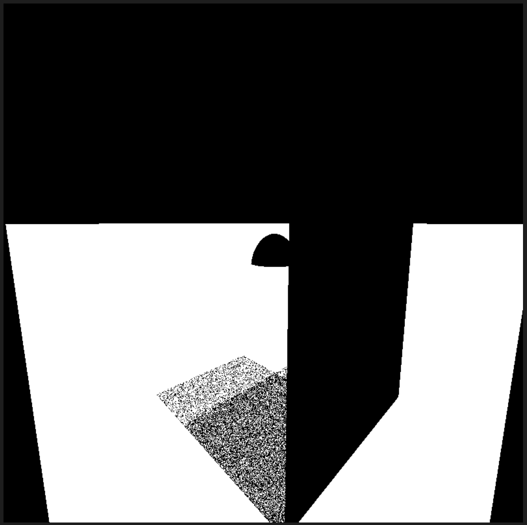
# KD Tree Path Tracer Optimization

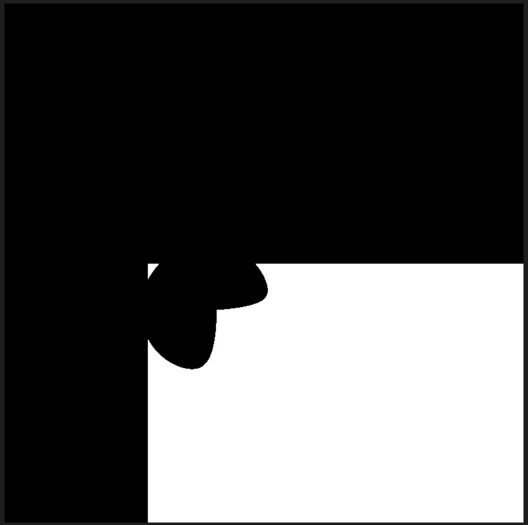
Rony Edde

Implementation issues.

- The last results might not look very promising but we can clearly see the intersections of the bounding boxes of the tree and the sections of the polygons represented by the sphere.

Note that without the KD-Tree I get 1fps, this was around 30.





# KD Tree Path Tracer Optimization

Rony Edde

## The nasty bug

- CUDA makes debugging quite challenging sometimes. NSight sometimes skips breakpoints and exits. This is when the GPU crashes. Printing things out was the only way to track some of the crashing bugs but CUDA can also crash when printing too many lines so tracking the indexing bug was the biggest hurdle.

# KD Tree Path Tracer Optimization

Rony Edde

What next?

- The next milestone will be to try fixing the collisions and using the correct colors3 for correctly bouncing rays.

## KD Tree Path Tracer Optimization

Rony Edde

Update 2

- Initial kd implementation working at this point.  Started optimizations using short stack hybrid solution.

## KD Tree Path Tracer Optimization

Rony Edde

- The working naïve traversal approach.

  The naive traversal approach works well.  It is definitely an improvement over a brute force check across all the geometry or a bounding box approach.

# KD Tree Path Tracer Optimization

Rony Edde

Checking for collision is done naïvely by traversing all the nodes that collide with the ray.  This is considered brute force because it's not always necessary to traverse all the nodes that might intersect the boxes.  This could lead to an overhead when traversing more nodes that are aligned.

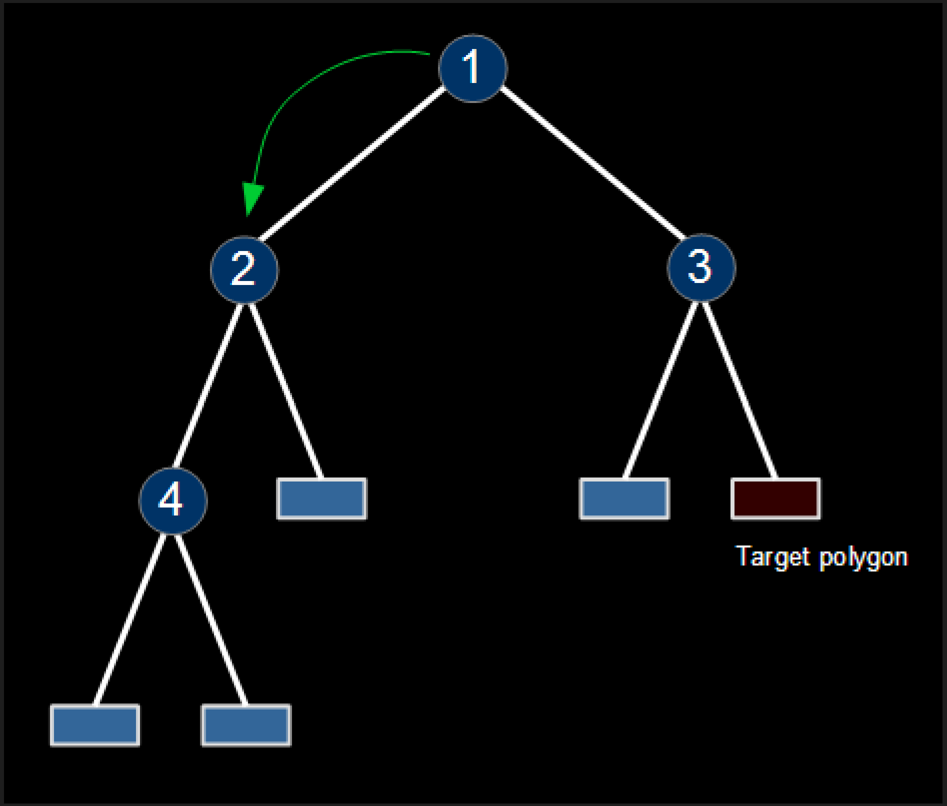
# KD Tree Path Tracer Optimization

Rony Edde

Naive algorithm:

Check if the ray collides with the root node.

If it does, check the left child.



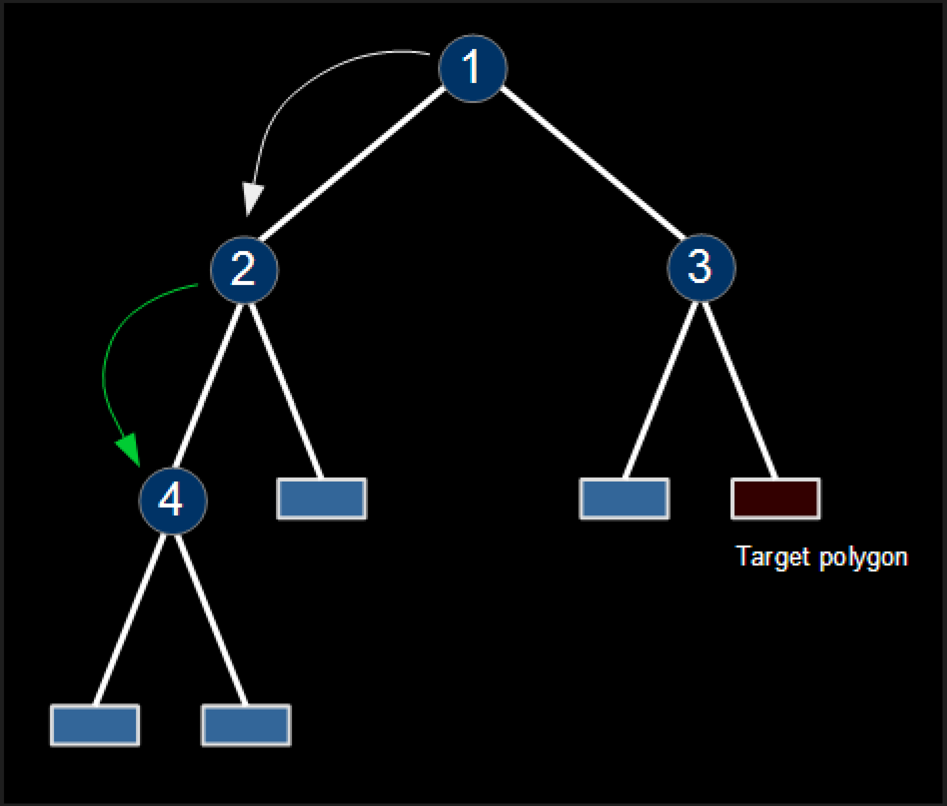
# KD Tree Path Tracer Optimization

Rony Edde

Naive algorithm:

Check again for collision.

If ray collides, check left child.



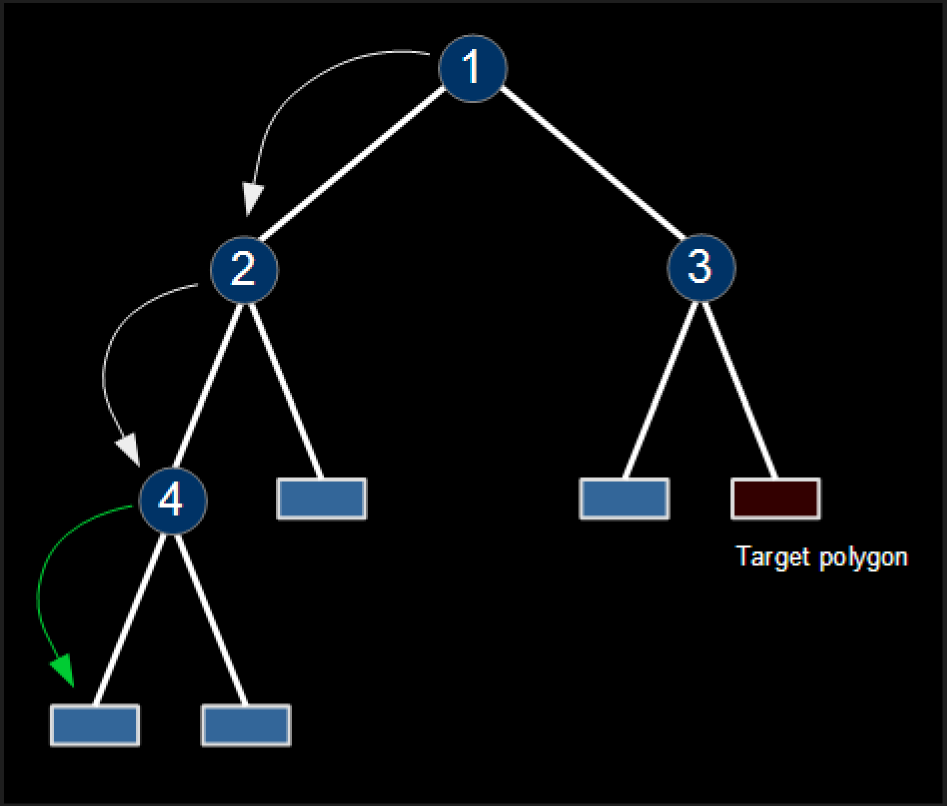
# KD Tree Path Tracer Optimization

Rony Edde

Naive algorithm:

Check again for collision.

If ray collides, check left child.



# KD Tree Path Tracer Optimization

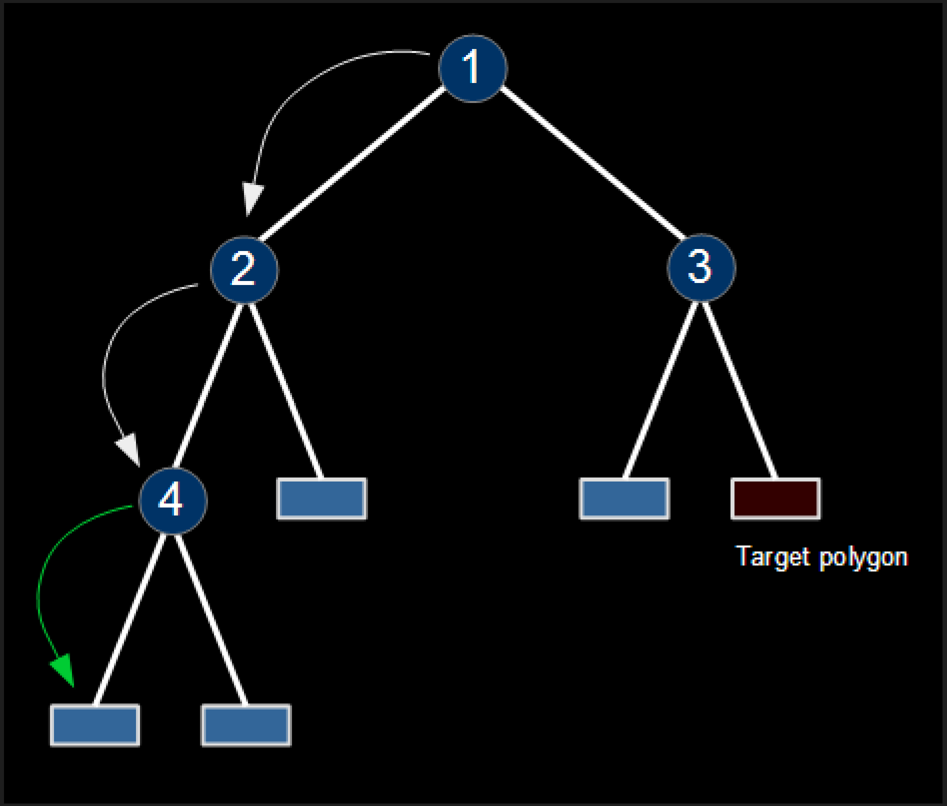
Rony Edde

Naive algorithm:

Child is a leaf, so we check for collisions with geometry.

If the geometry doesn't collide,

Step back.



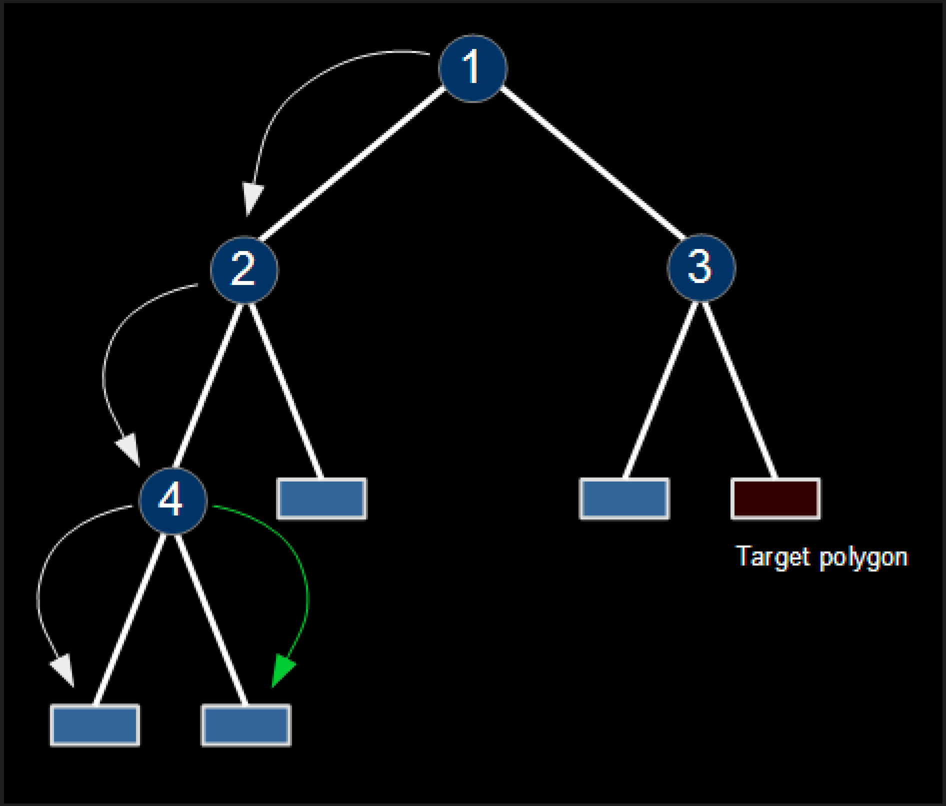
# KD Tree Path Tracer Optimization

Rony Edde

Naive algorithm:

After stepping back we check the remaining child.

In this case it's the right node.

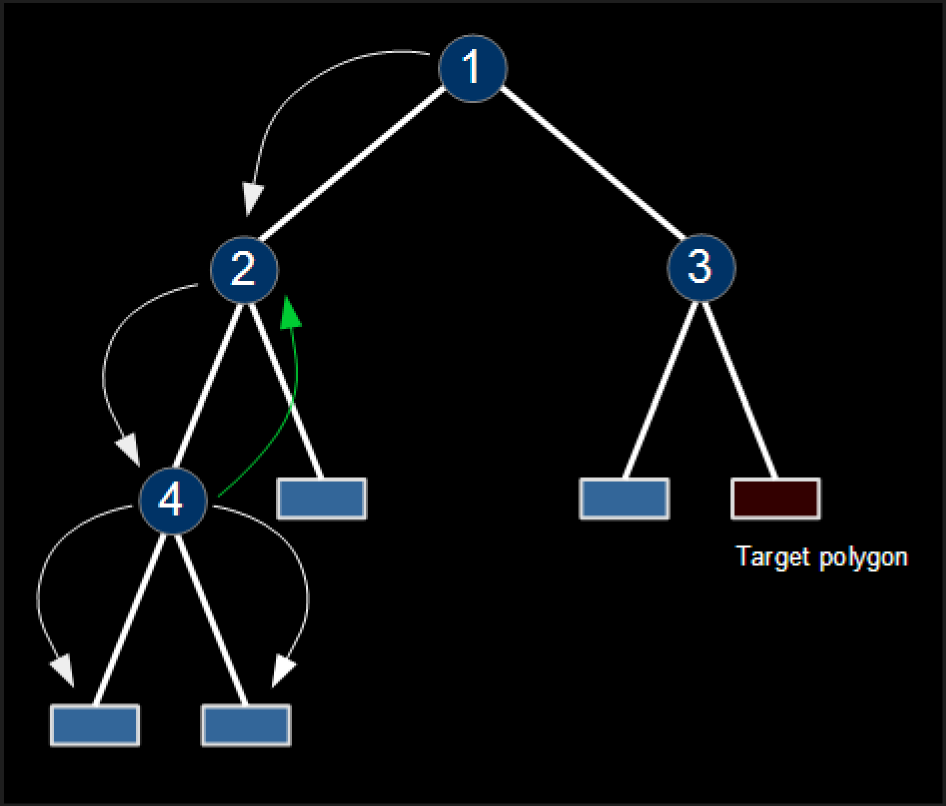


# KD Tree Path Tracer Optimization

Rony Edde

Naive algorithm:

No polygons hit so we back track.

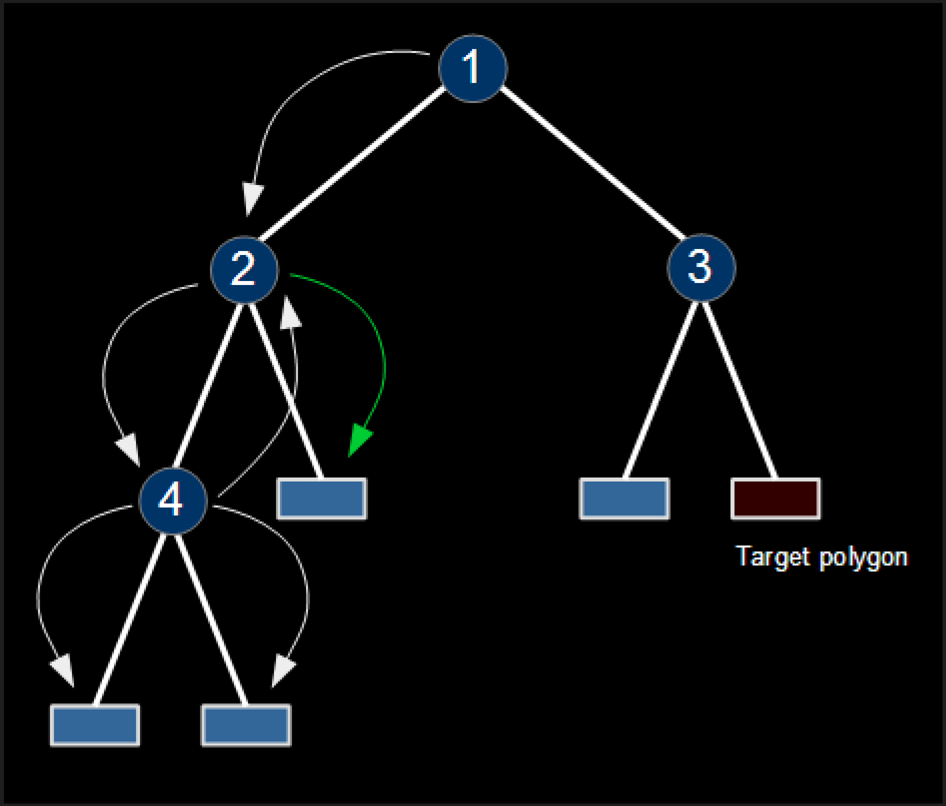


# KD Tree Path Tracer Optimization

Rony Edde

Naive algorithm:

Now we check the right child.



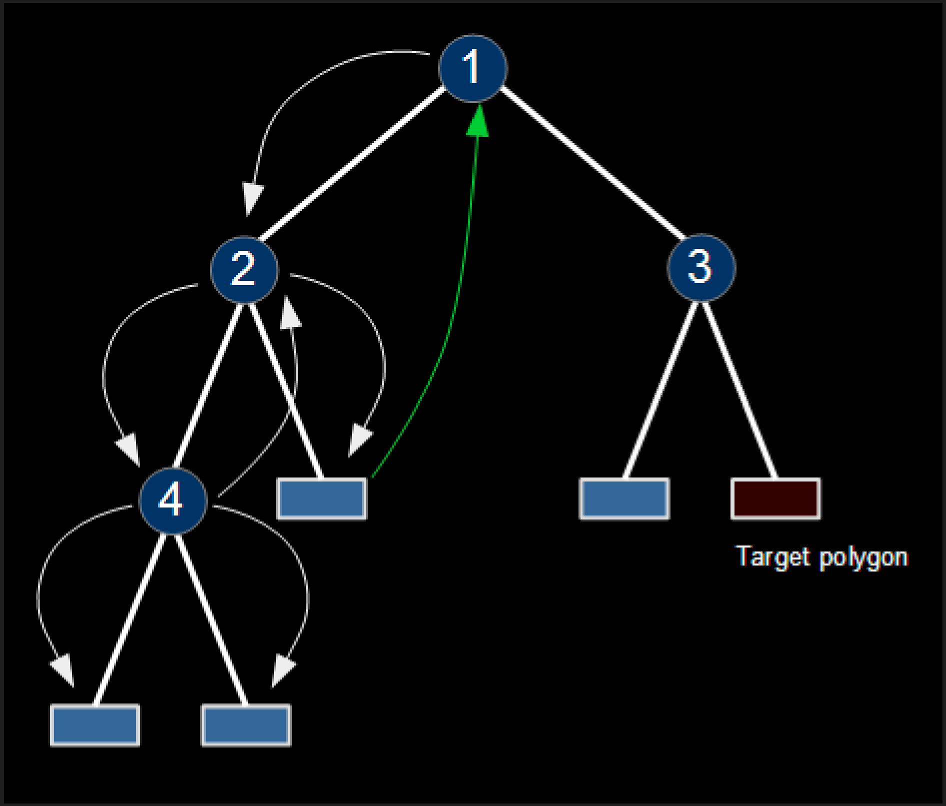
# KD Tree Path Tracer Optimization

Rony Edde

Naive algorithm:

We might have collided but we need to check the remaining nodes, so we backtrack.

Our hit might not be the closest.



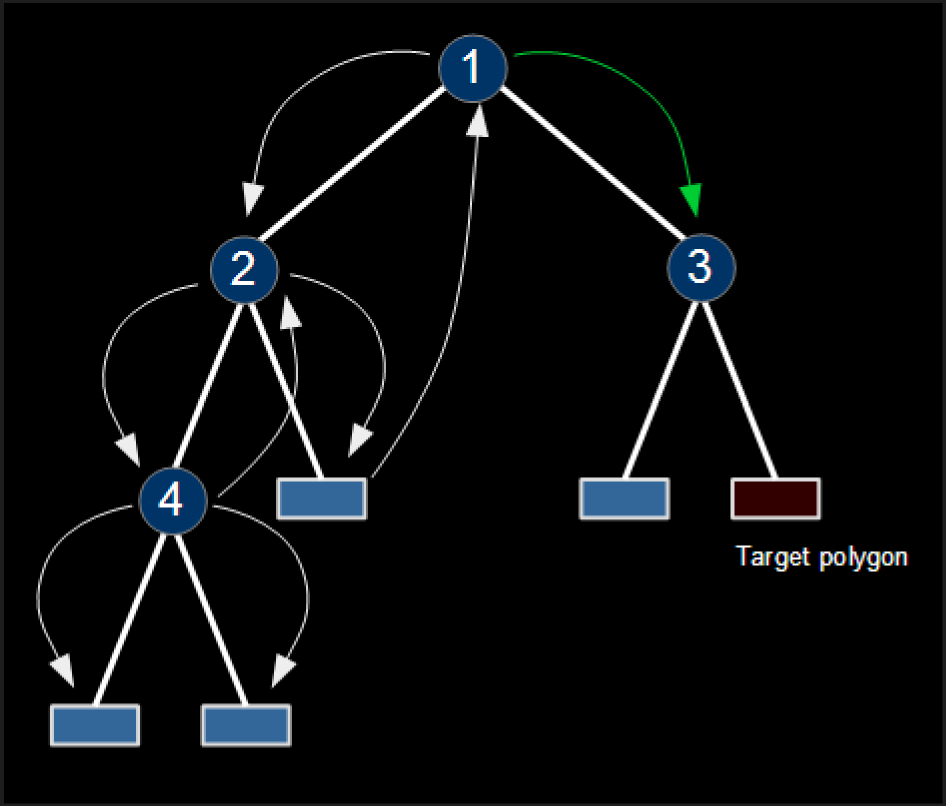
# KD Tree Path Tracer Optimization

Rony Edde

Naive algorithm:

We are back at the root node but we still haven't checked the remaining nodes.

We check the right side.



# KD Tree Path Tracer Optimization

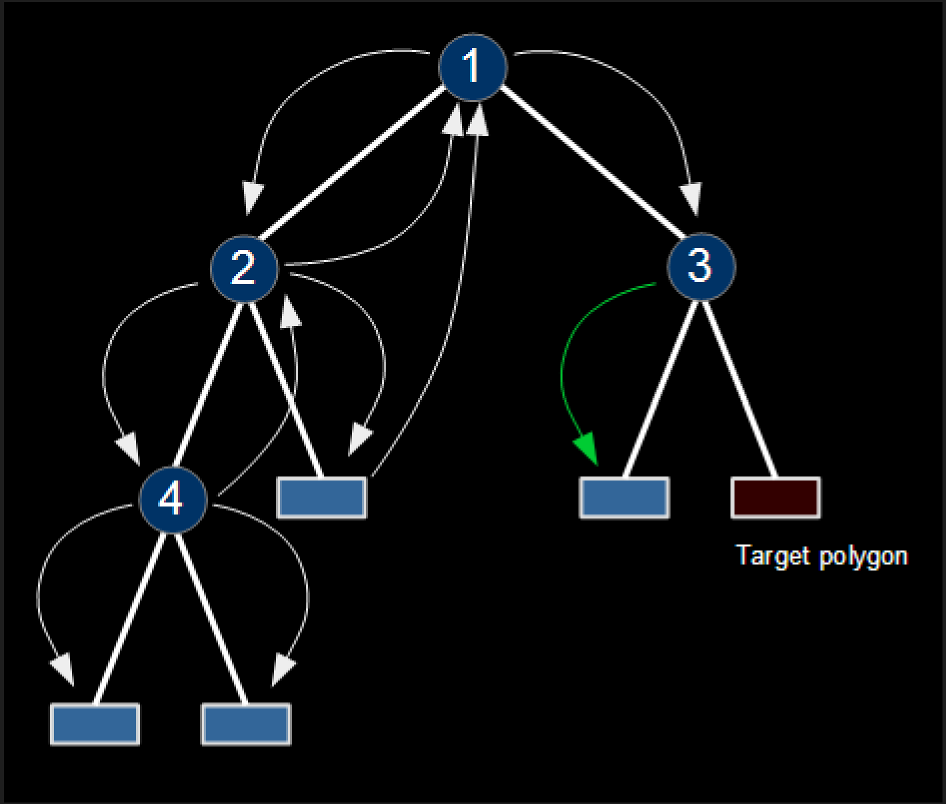
Rony Edde

Naive algorithm:

There are 2 remaining nodes to check.  We check the left and get no collision.

No need to collide with geometry.

We step back.



# KD Tree Path Tracer Optimization

Rony Edde

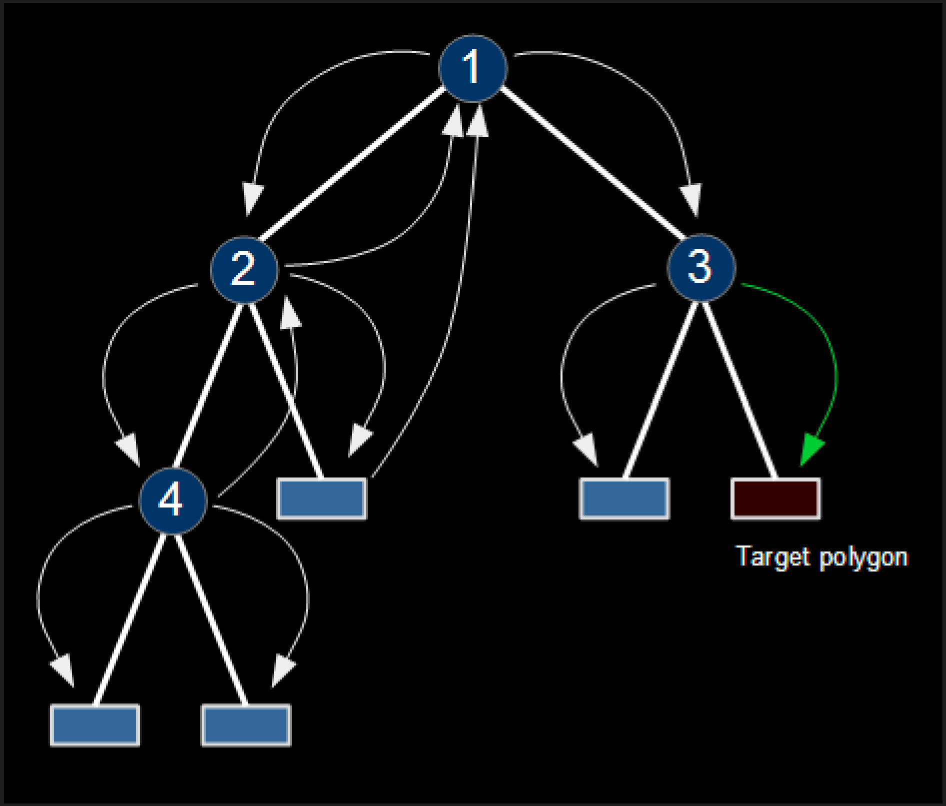
Naive algorithm:

For the last leaf node our ray hits the bounds so we check the geometry.

We get a hit that is closer.

All nodes have been visited.

Return collision point geometry.



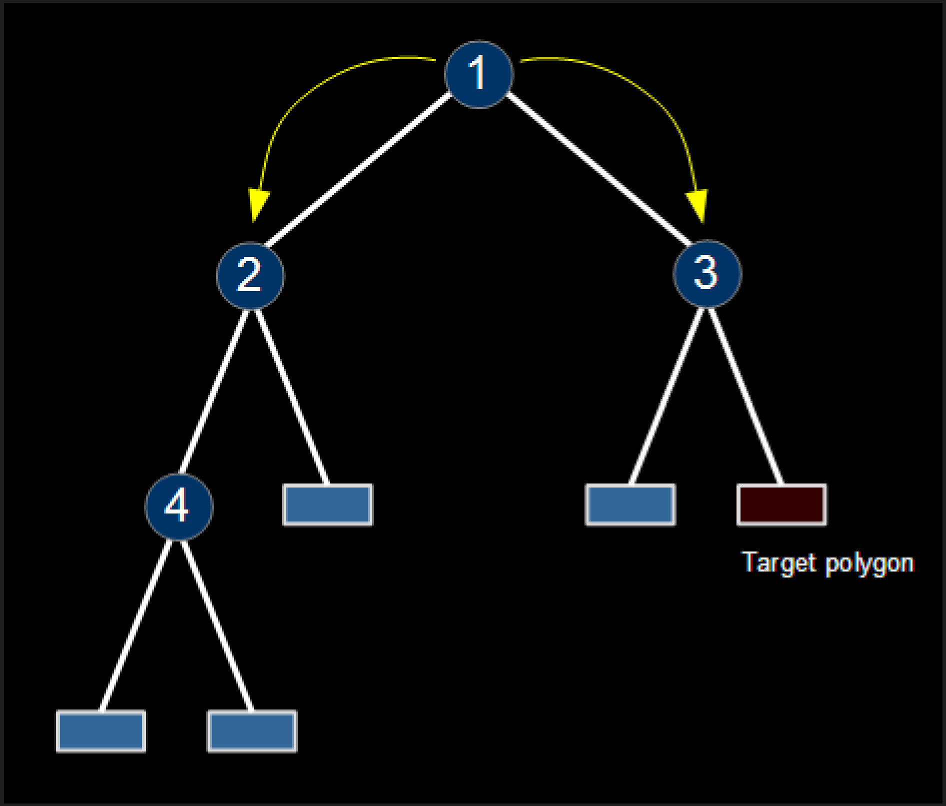
# KD Tree Path Tracer Optimization

Rony Edde

Short-stack algorithm:

Instead of traversing all nodes, we still start at the root but we check for the closest splitting plane.

By comparing the axis distance, we start looking at the closest node.



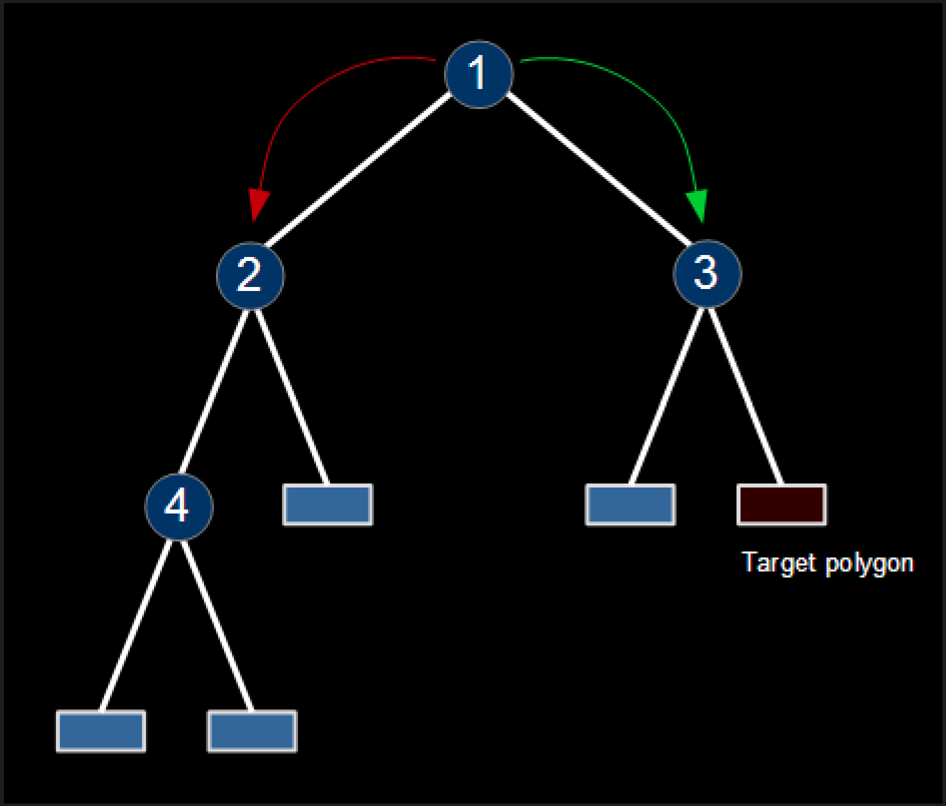
# KD Tree Path Tracer Optimization

Rony Edde

Short-stack algorithm:

We look at the splitting plane axis and compare the splitting plane with node 2 and node 3 to determine which one to start traversing.

Node 3 is the closest so we start with it and put node 2 in the stack to revisit.

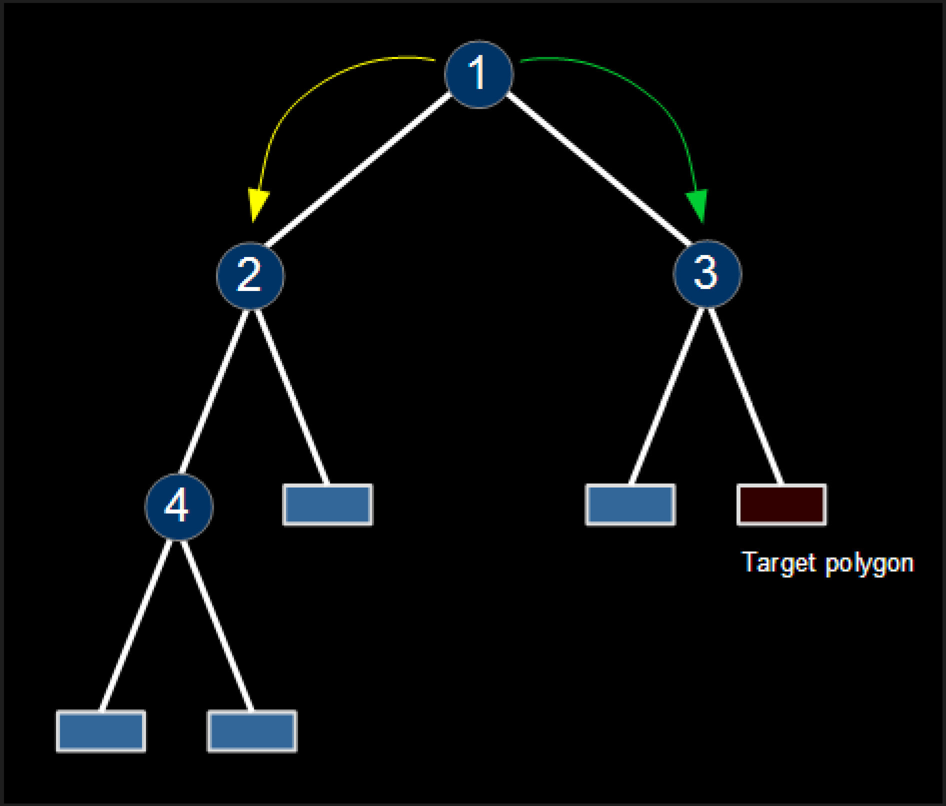


# KD Tree Path Tracer Optimization

Rony Edde

Short-stack algorithm:

Now that node2 is pushed, we check for collision with node 3's bounds.

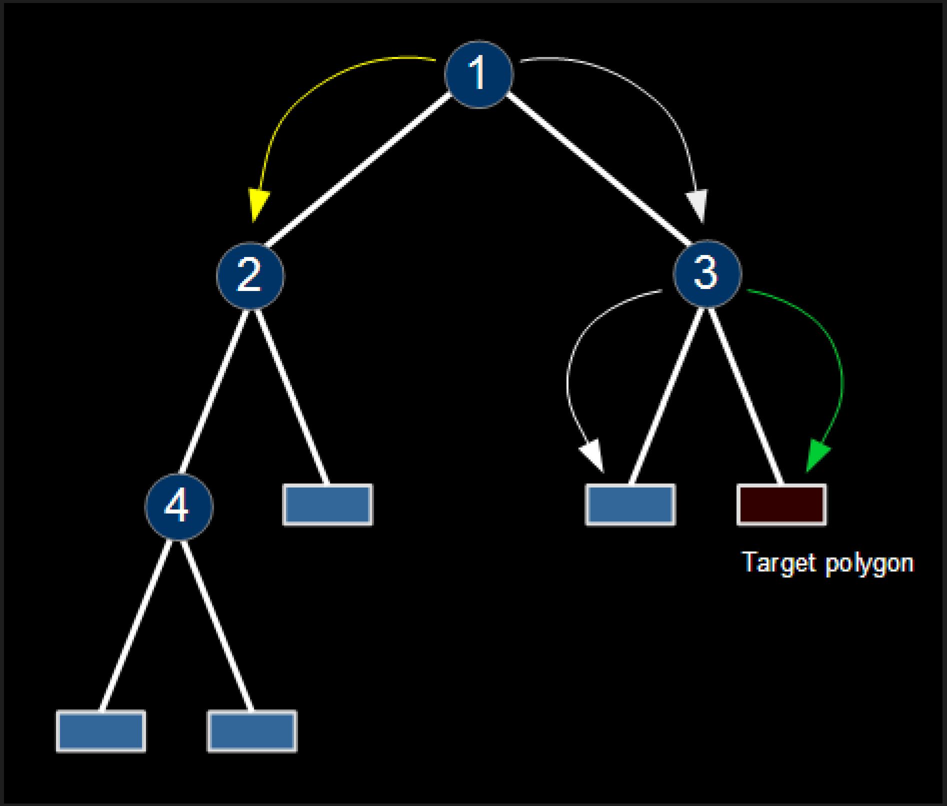


# KD Tree Path Tracer Optimization

Rony Edde

Short-stack algorithm:

Node 3 collides with the ray so we check the closest node on the current axis.  It's determined that it's the right node, so we check collision with the right node.  We get a collision and so can determine that we have collided with the closest possible polygon.

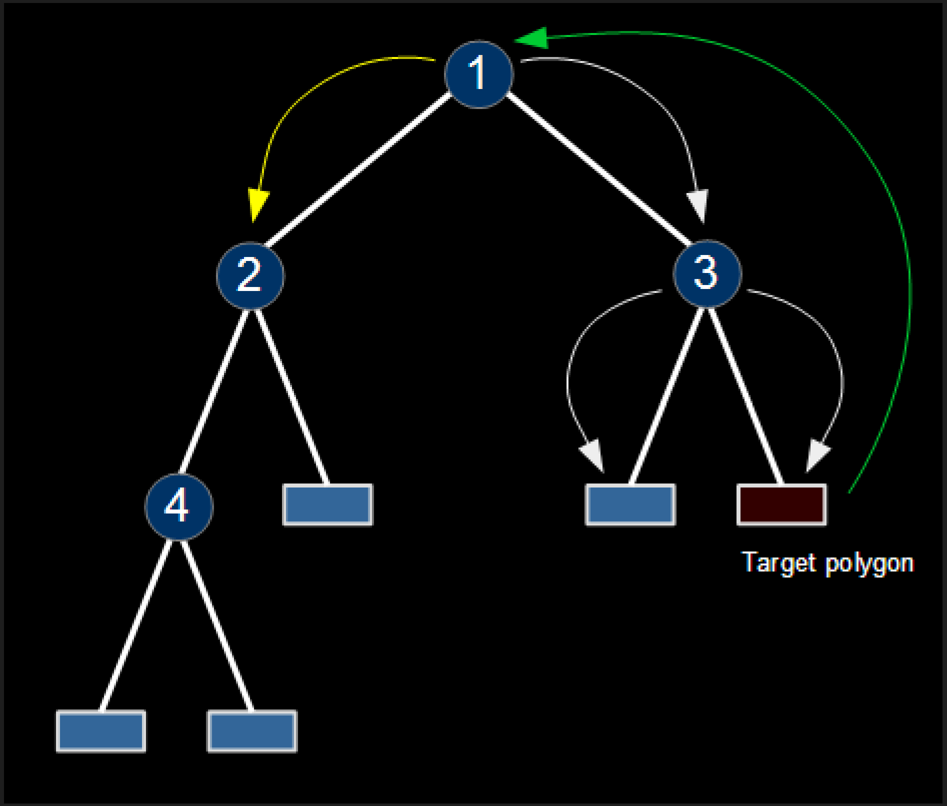


# KD Tree Path Tracer Optimization

Rony Edde

Short-stack algorithm:

We step back to check the remaining nodes. In this case it's node 2.

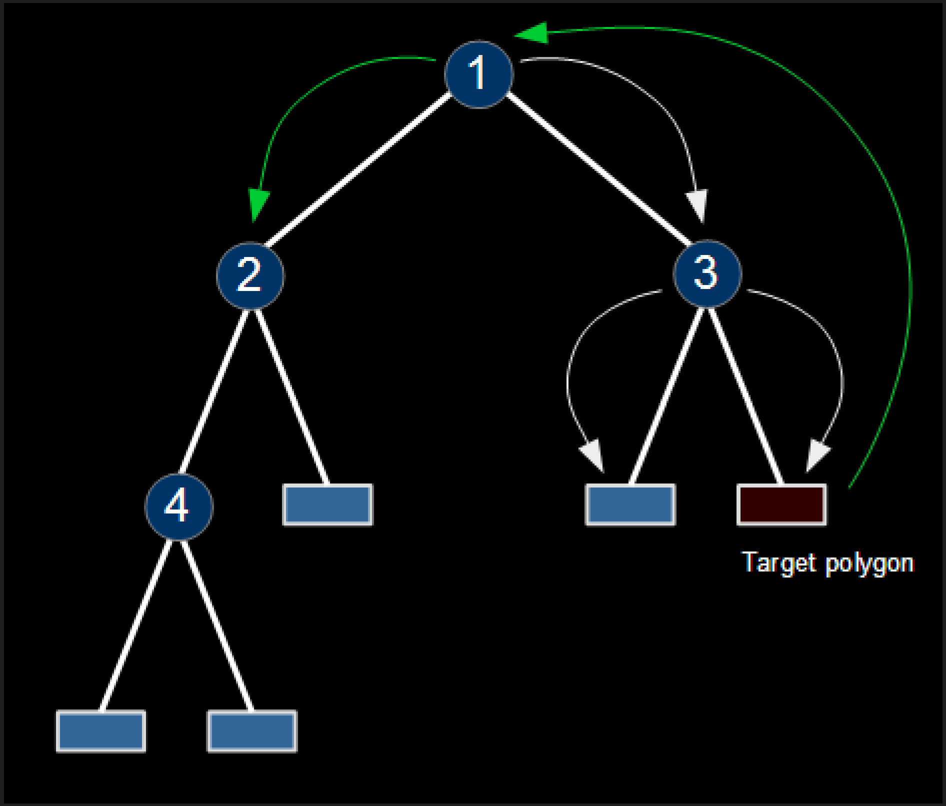


# KD Tree Path Tracer Optimization

Rony Edde

Short-stack algorithm:

Node 2 collides father away and it's the only node on the stack so we are done and can return.



# KD Tree Path Tracer Optimization

Rony Edde

The short-stack improved performance considerably but memory issues started appearing around 3 million vertices.

This was solved by reducing the size of the data structures.  The original KDnode and KDtriangle classes used respectively 136 and 116 Bytes.  This can have a considerable impact with over one million triangles in the scene.

# KD Tree Path Tracer Optimization

Rony Edde

The KDnode and KDtriangle classes can be easily optimized into smaller structs once the tree was built since tree traversal requires index locations only in addition to parent child and bounding information per KDnode.  The KDtriangle structure required vertex and normal information as well as shading information.

The result is 2 data structures: NodeBare and TriBare with the following sizes:

NodeBare: 64 / TriBare: 76

Resulting in using 47% and 65% memory from the initial KDnode and KDtriangle memory respectively.

## KD Tree Path Tracer Optimization

Rony Edde

Results:

The following results show some renders done with high poly count geometry.  These renders would not be possible using a brute force method.

All models are from http://www.turbosquid.com

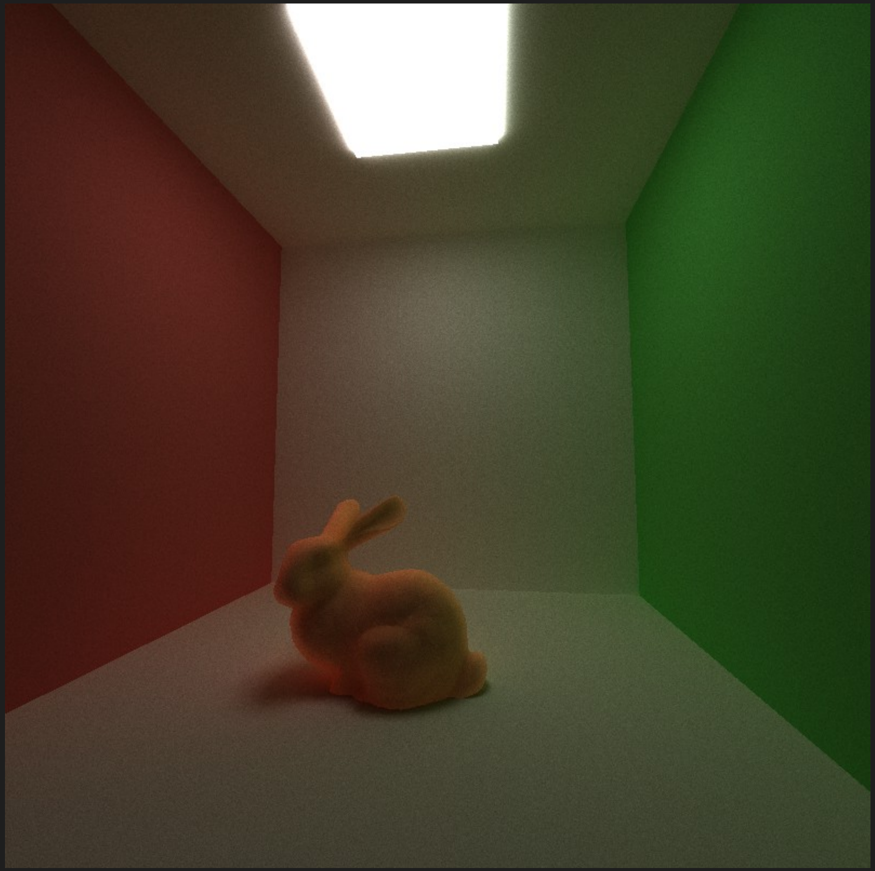
# KD Tree Path Tracer Optimization

Rony Edde

Stanford Bunny 5000 iterations:

70k polys

208k verts.



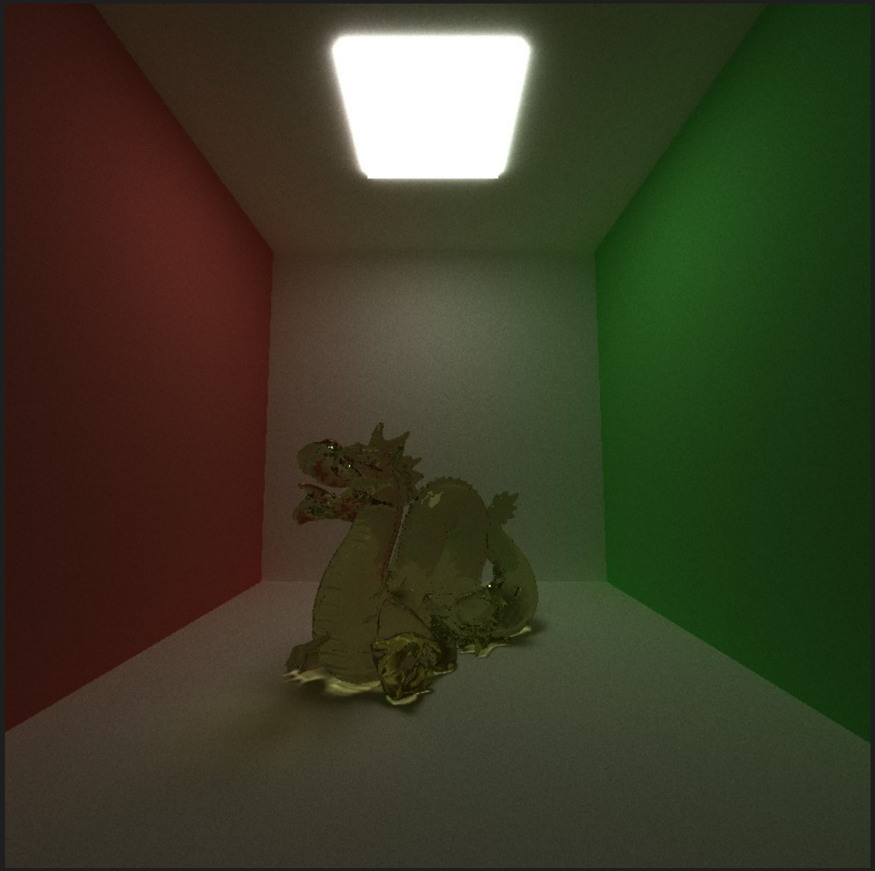
# KD Tree Path Tracer Optimization

Rony Edde

Stanford Dragon 5000 iterations:

100k polys

300k verts.

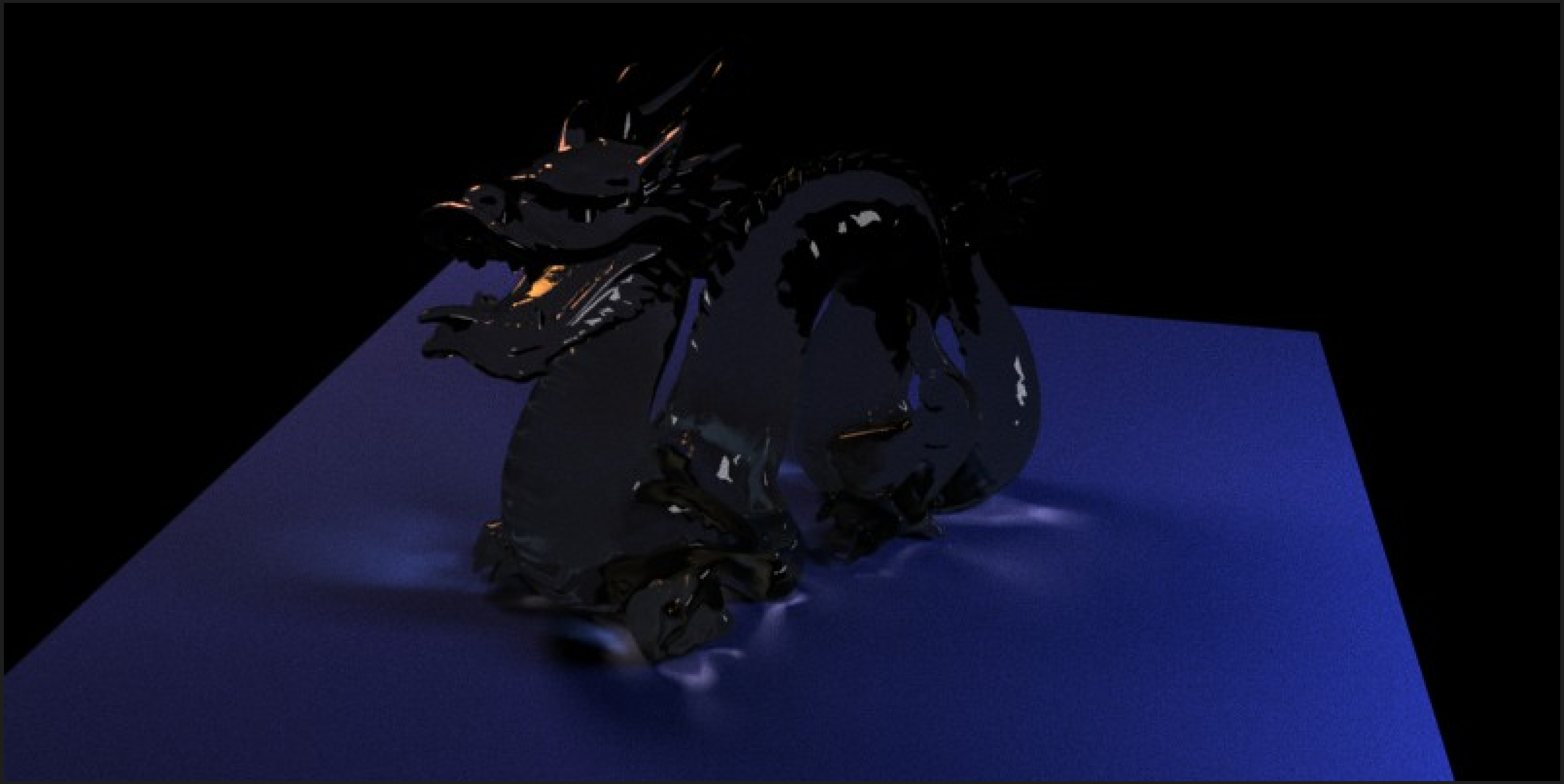


# KD Tree Path Tracer Optimization

Rony Edde

Stanford Dragon scene 2 ~8500 iterations:

100k polys 300k verts.

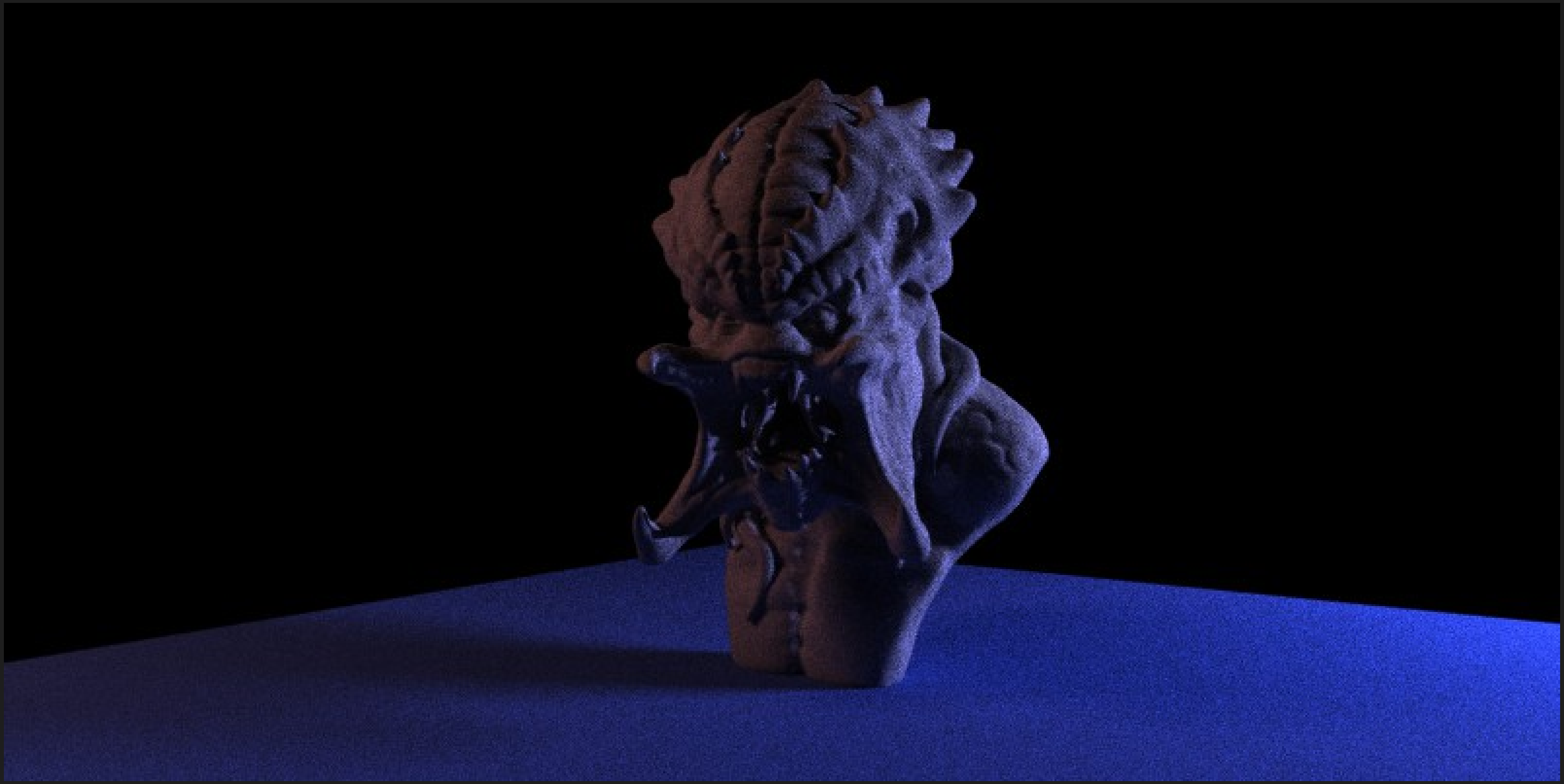


## KD Tree Path Tracer Optimization

Rony Edde

Predator model 5000 iterations:

333k polys 1 million verts.

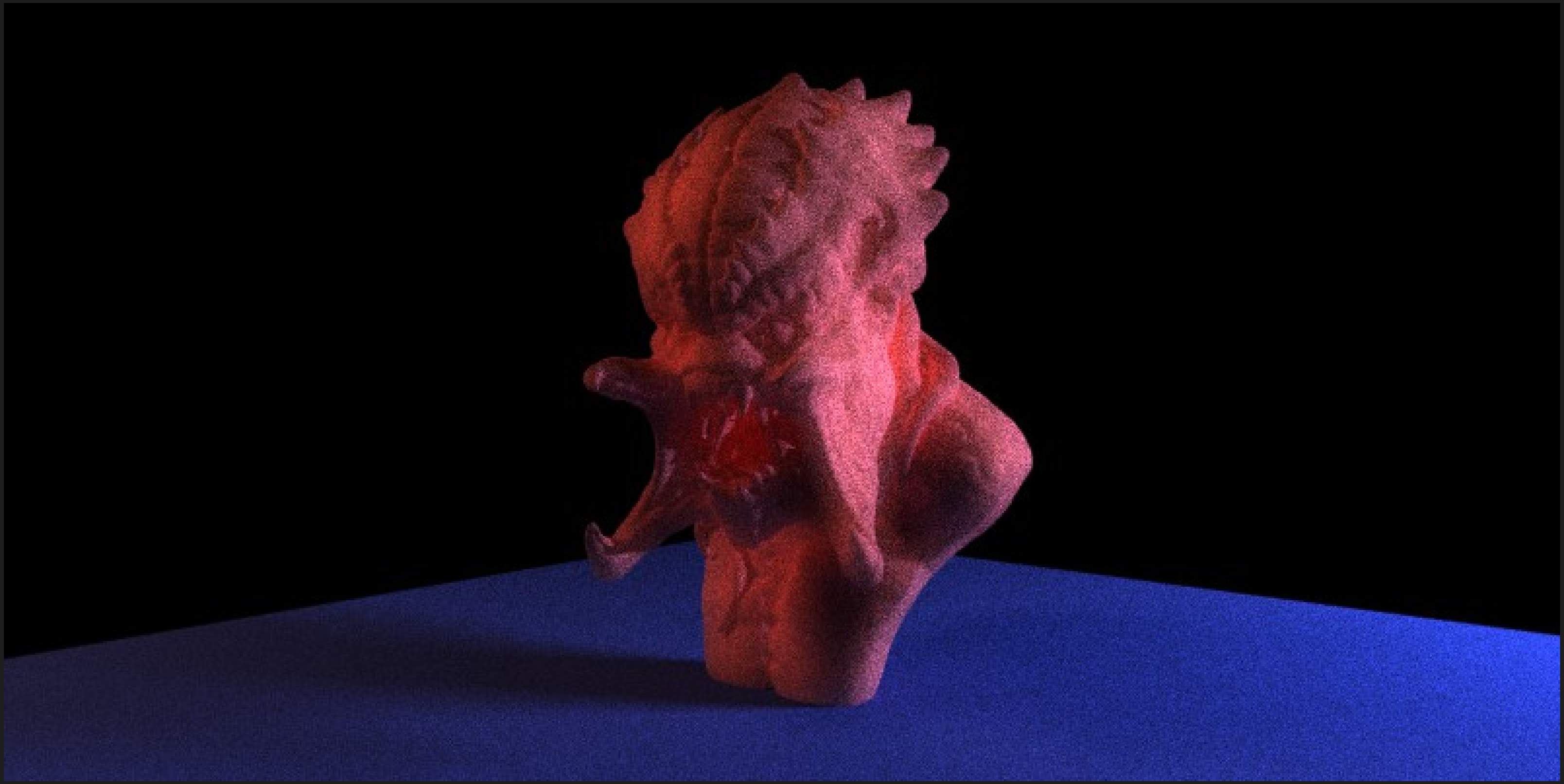


# KD Tree Path Tracer Optimization

Rony Edde

Predator model with sub-surface scatttering:

333k polys 1 million verts.

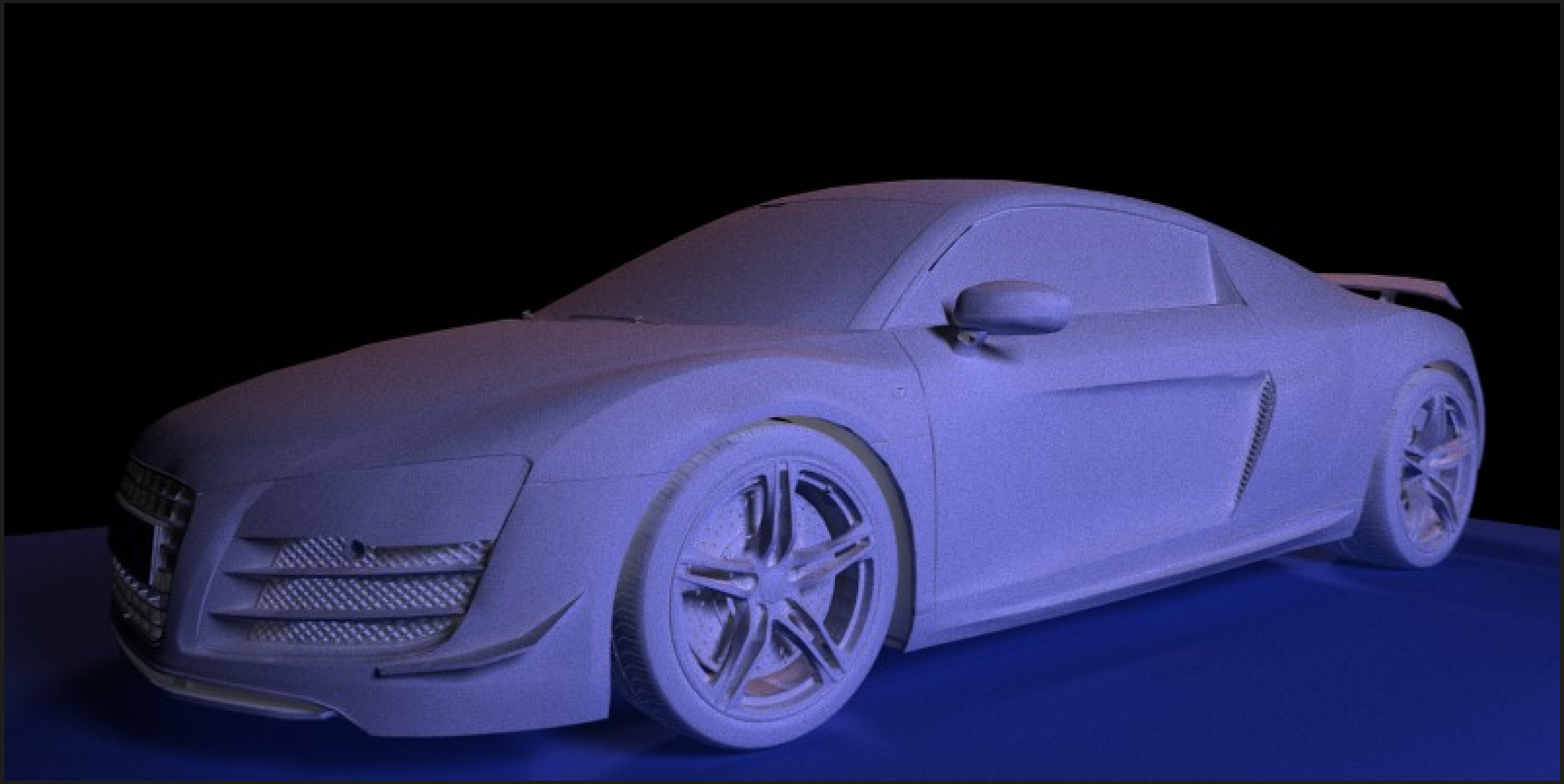


# KD Tree Path Tracer Optimization

Rony Edde

Audi R8 30000 iterations:

563k polys 1.7 million verts.

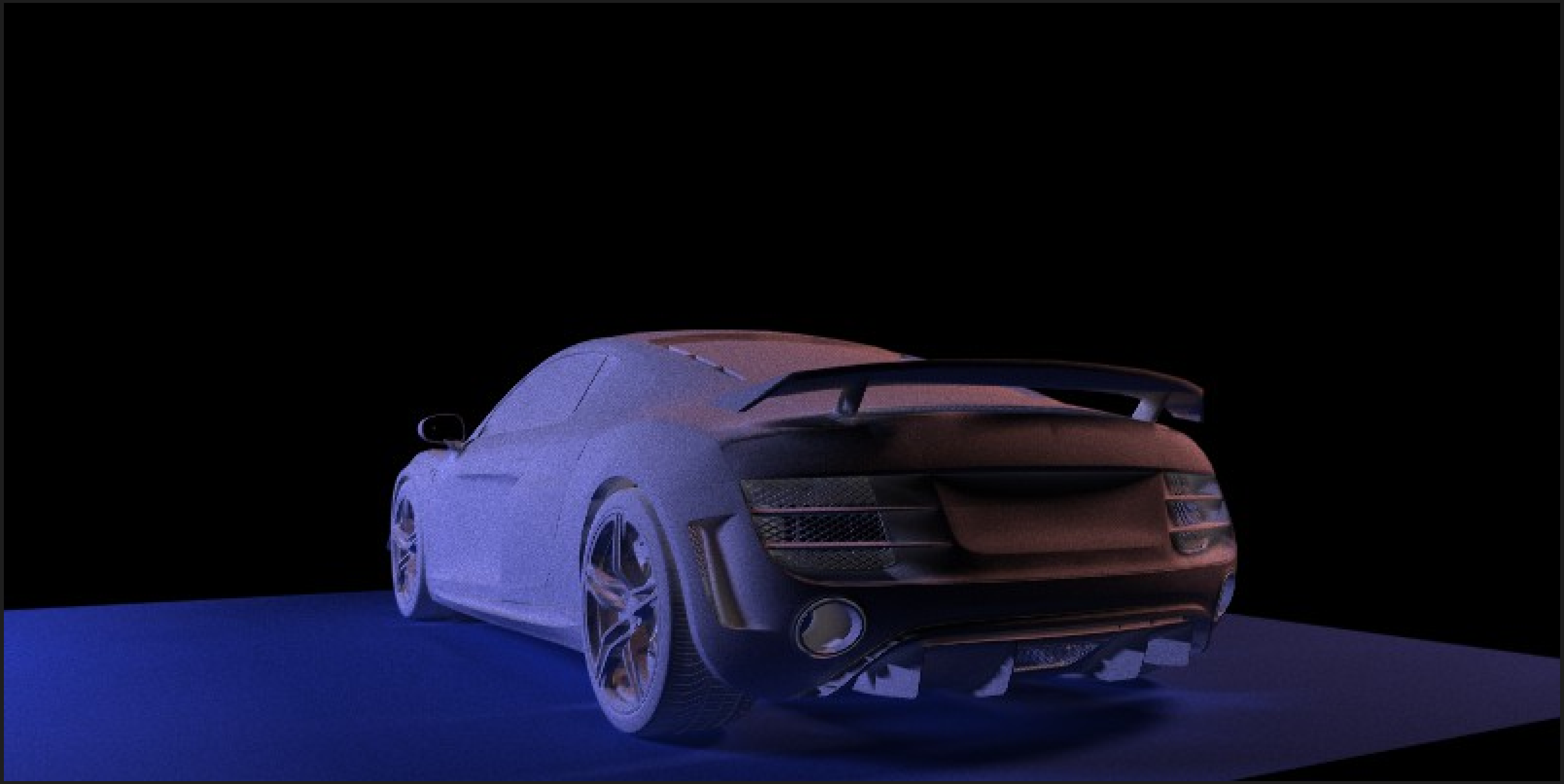


# KD Tree Path Tracer Optimization

Rony Edde

Audi R8 rear-view with 15000 iterations:

563k polys 1.7 million verts.

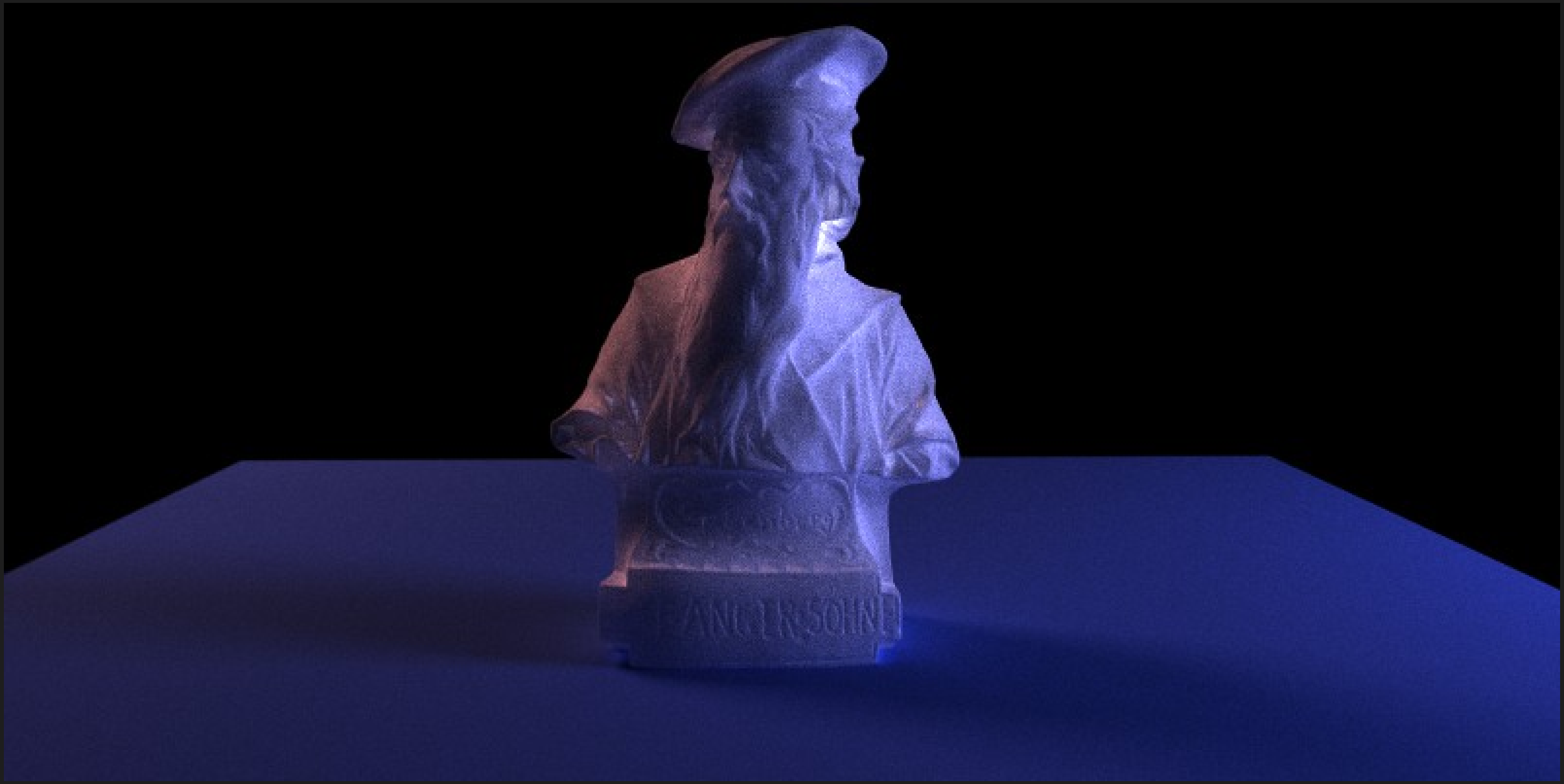


# KD Tree Path Tracer Optimization

Rony Edde

Gutenberg scan 15000 iterations:

~1.06 million polys ~3.2 million verts.



# KD Tree Path Tracer Optimization

Rony Edde

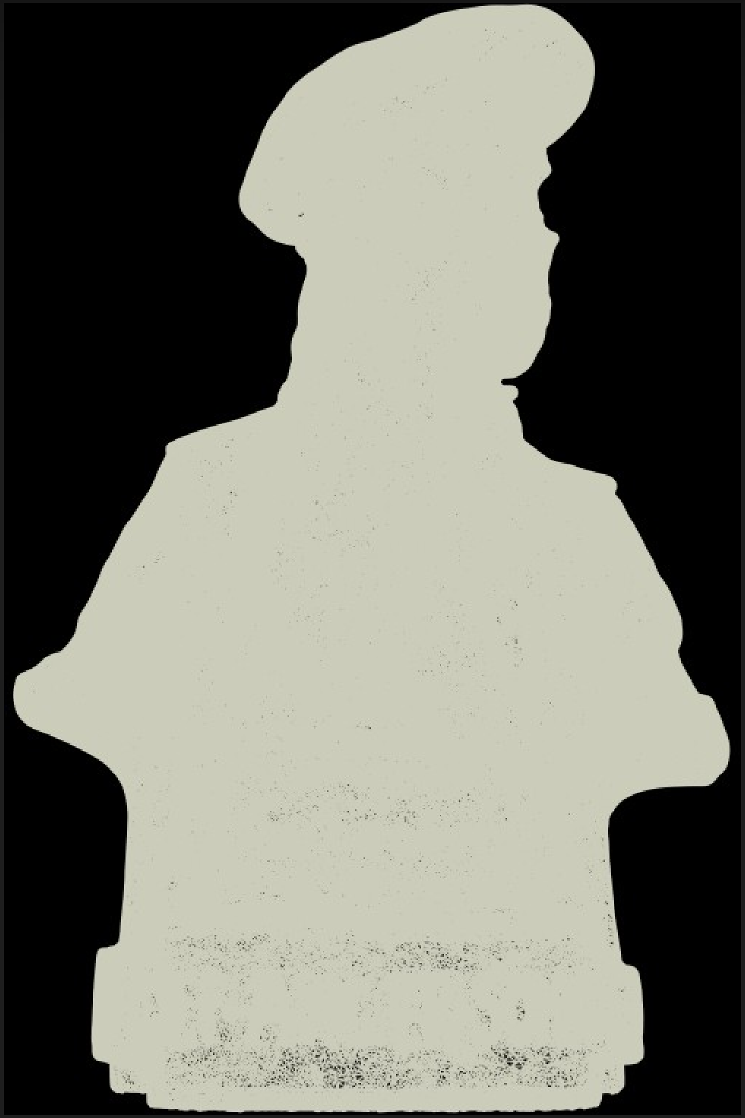
Most of these models fail to load with a standard approach. Using a bounding box check didn't help wither. It was only possible to load these models using the kd-tree.

The Gutenberg model was an interesting test because of the high density scan. The following is a wireframe of that model.

# KD Tree Path Tracer Optimization

Rony Edde

Gutenberg wireframe:



# KD Tree Path Tracer Optimization

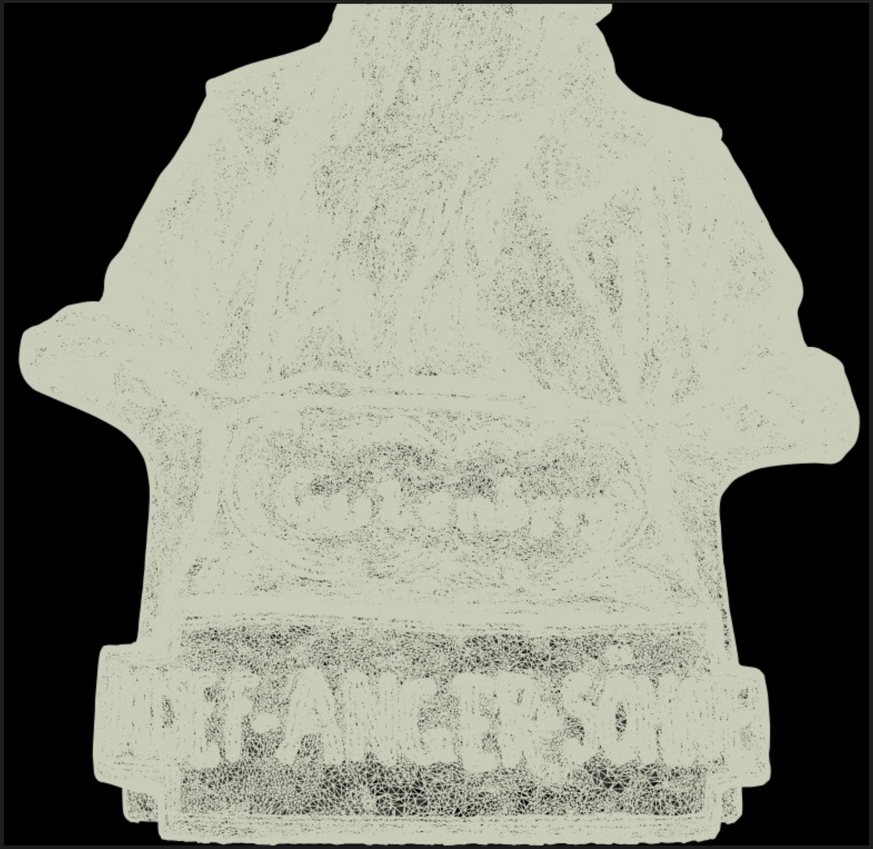
Rony Edde

Gutenberg wireframe head:

# KD Tree Path Tracer Optimization

Rony Edde

Gutenberg wireframe torso:



# KD Tree Path Tracer Optimization

Rony Edde

Visualizing the tree:

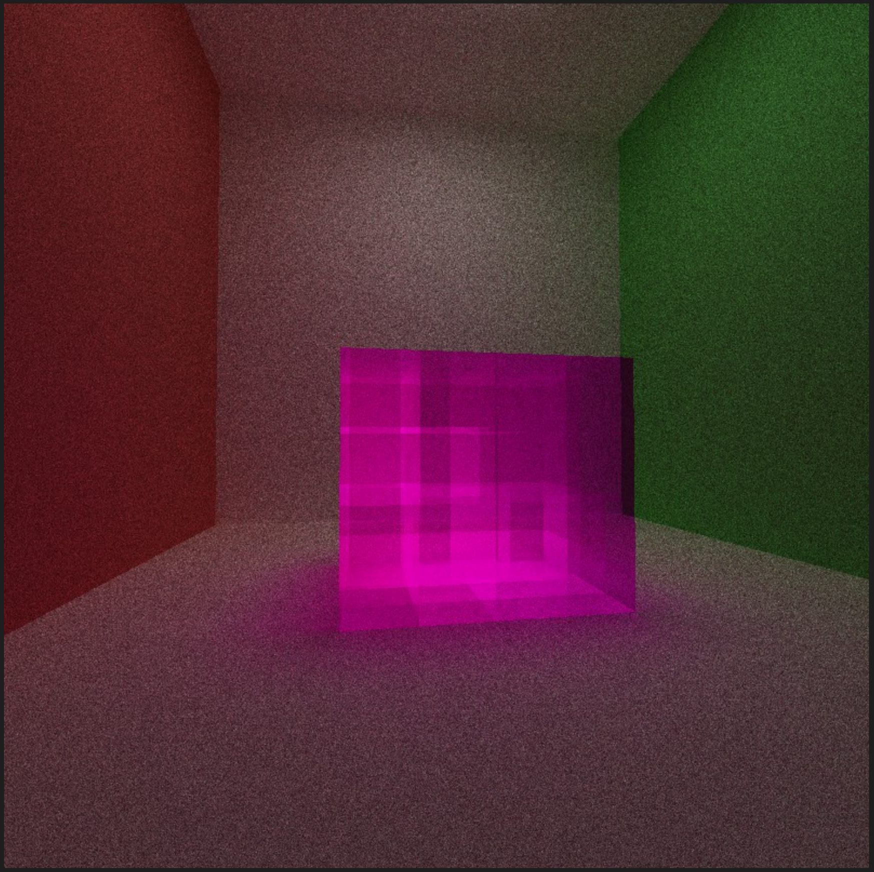
Visualization of the tree is one of the best ways to test if all is working well. The visualization toggle was an important part of making sure the sectioning is working correctly. The following are a few renders that show the structure of the tree. Rather than saving the tree to file and sending the data to a 3rd party software, being able to see the aligned tree structure offers a better debugging framework.

# KD Tree Path Tracer Optimization

Rony Edde

Visualizing the tree:

Short depth bunny
Stanford Bunny:

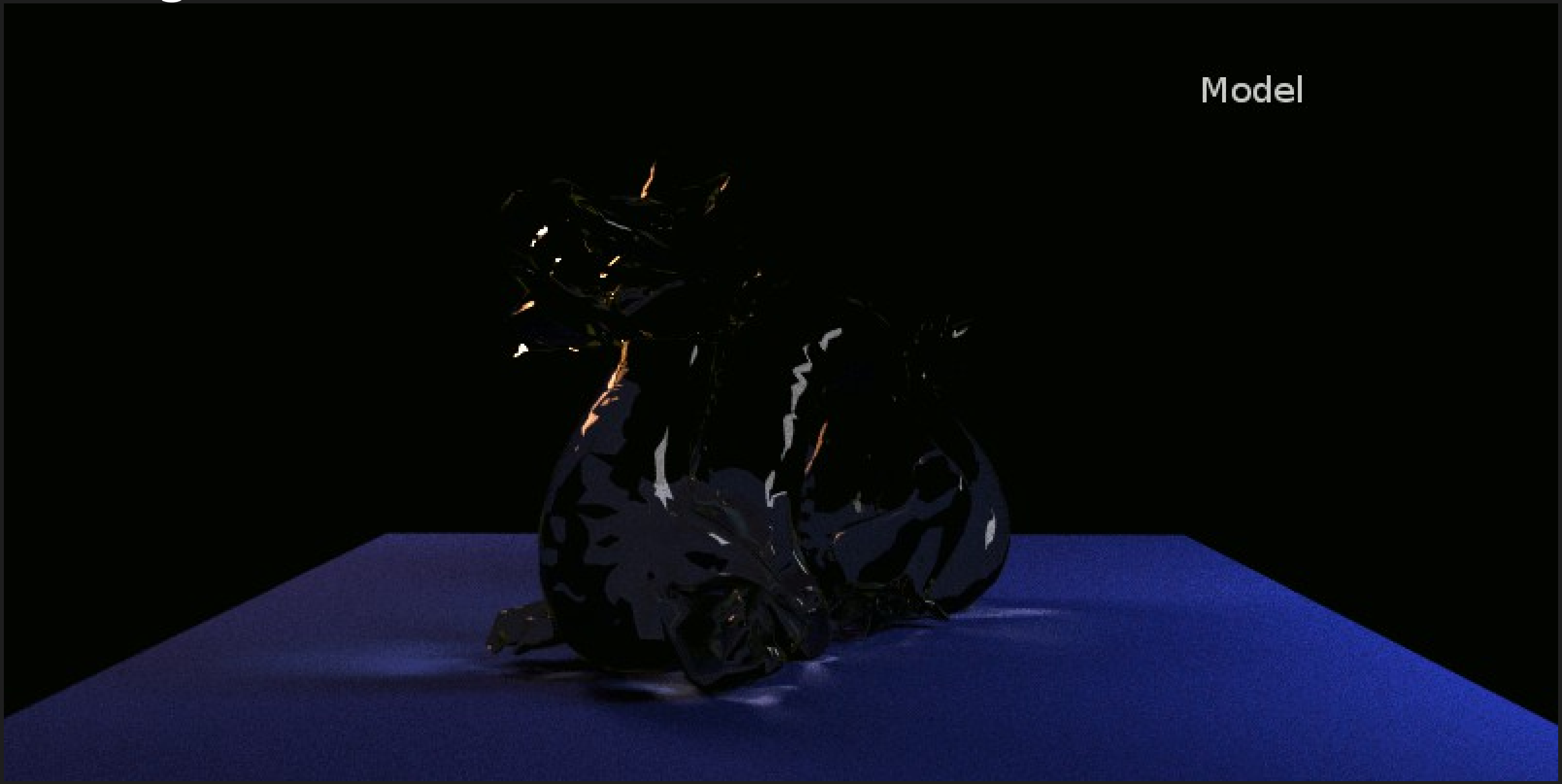


# KD Tree Path Tracer Optimization

Rony Edde

Visualizing the tree:

Stanford Dragon:

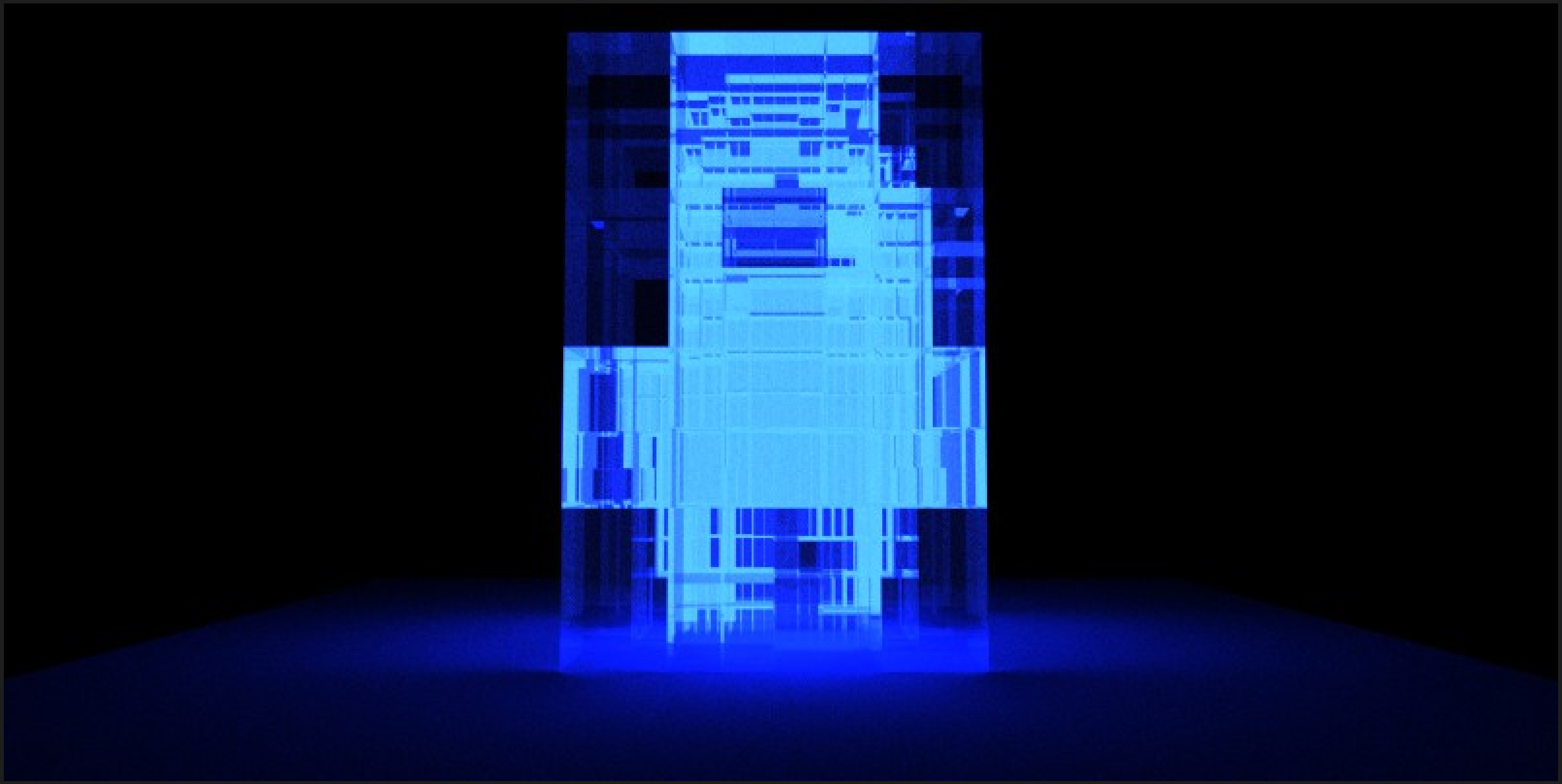


# KD Tree Path Tracer Optimization

Rony Edde

Visualizing the tree:

Predator front:

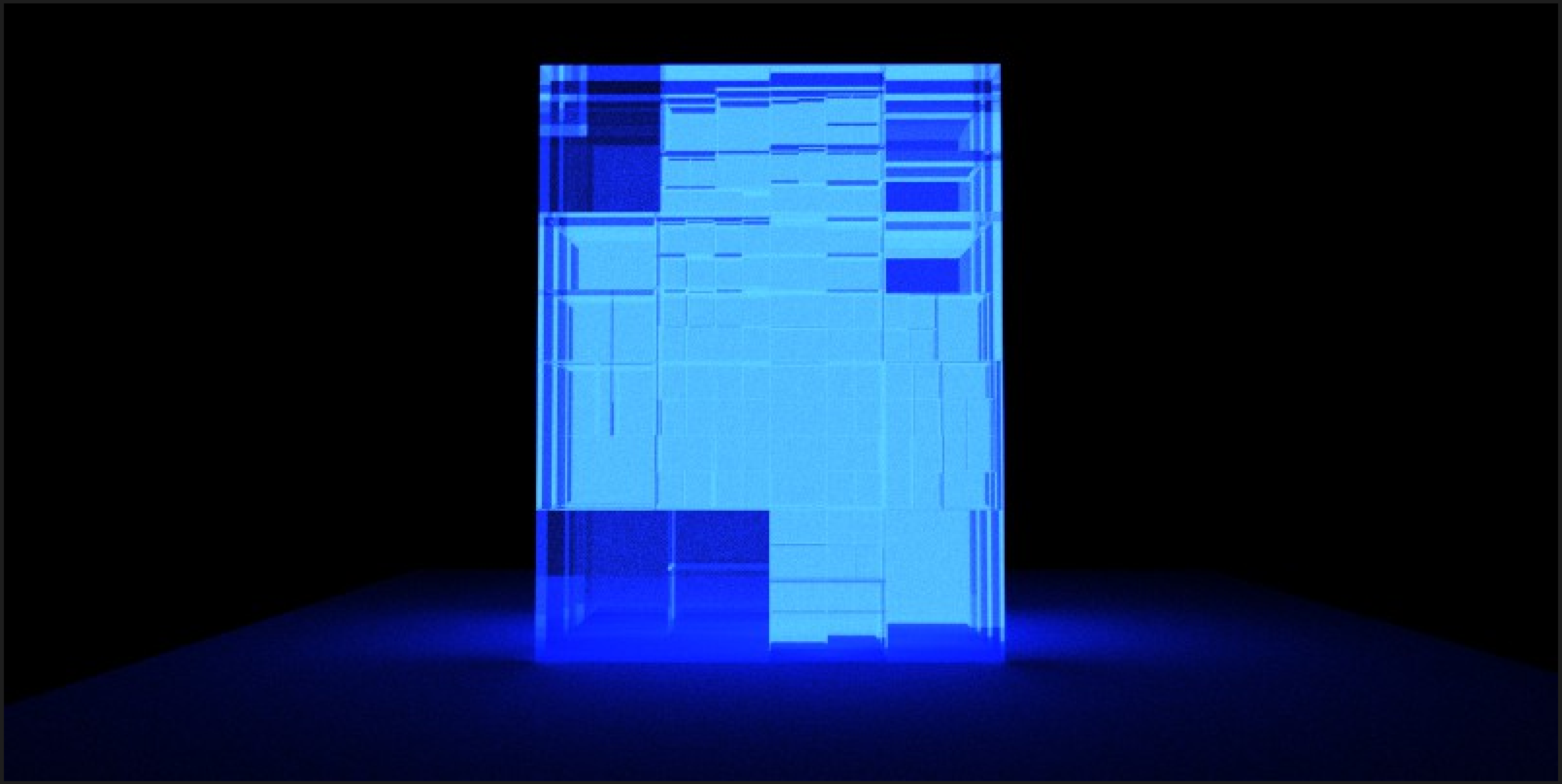


## KD Tree Path Tracer Optimization

Rony Edde

Visualizing the tree:

Predator side:



# KD Tree Path Tracer Optimization

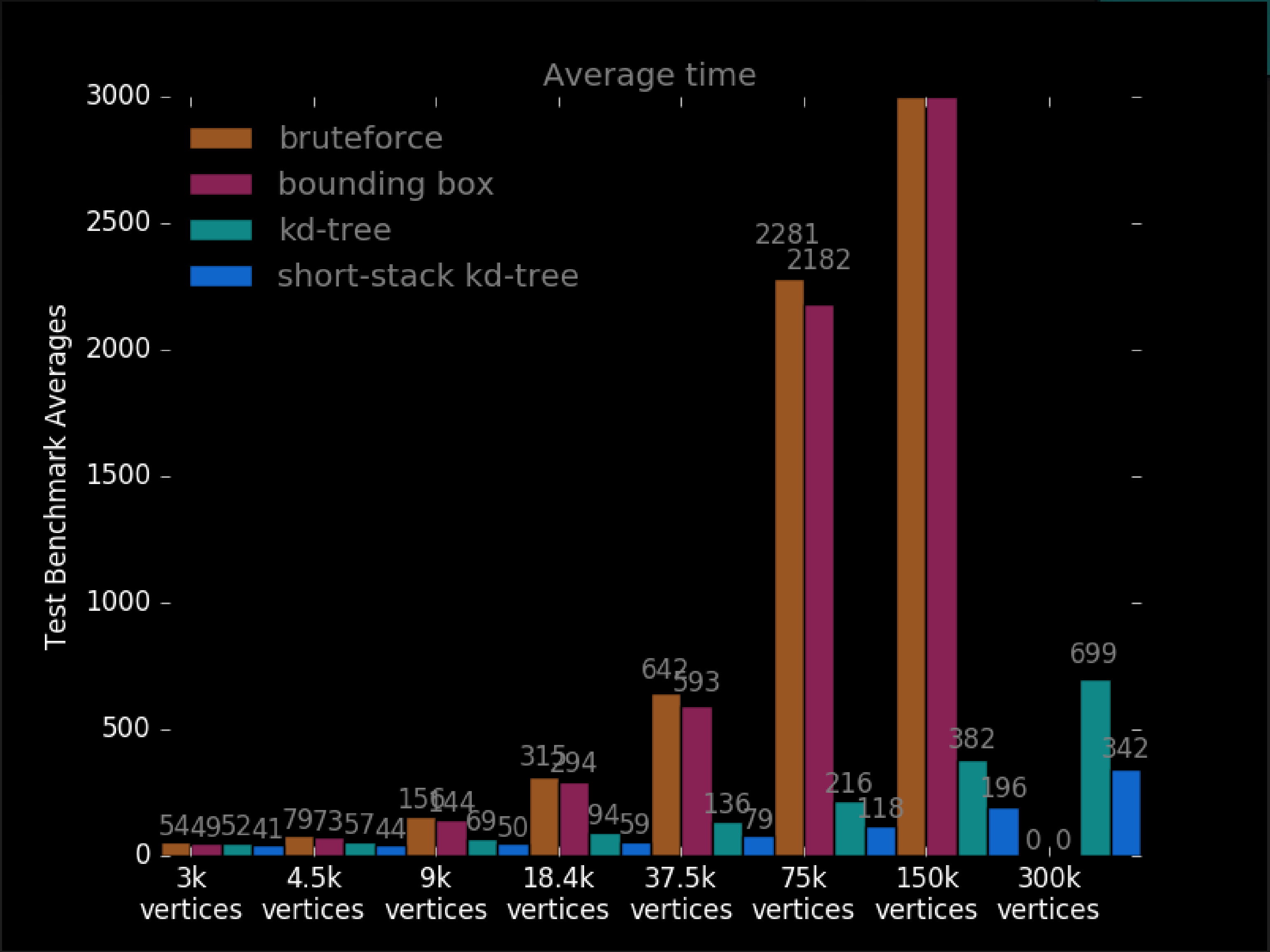
Rony Edde

Performance analysis:

Tests were make on the Stanford Dragon with 8 levels of resolution ranging from 1 (lowest) to 8 (full resolution). The resolutions bench marked were 3k, 4.5k, 9k, 18.4k, 37.5k, 75k, 150k, 300k vertices.

# KD Tree Path Tracer Optimization

Rony Edde



# KD Tree Path Tracer Optimization

Rony Edde

Performance analysis:

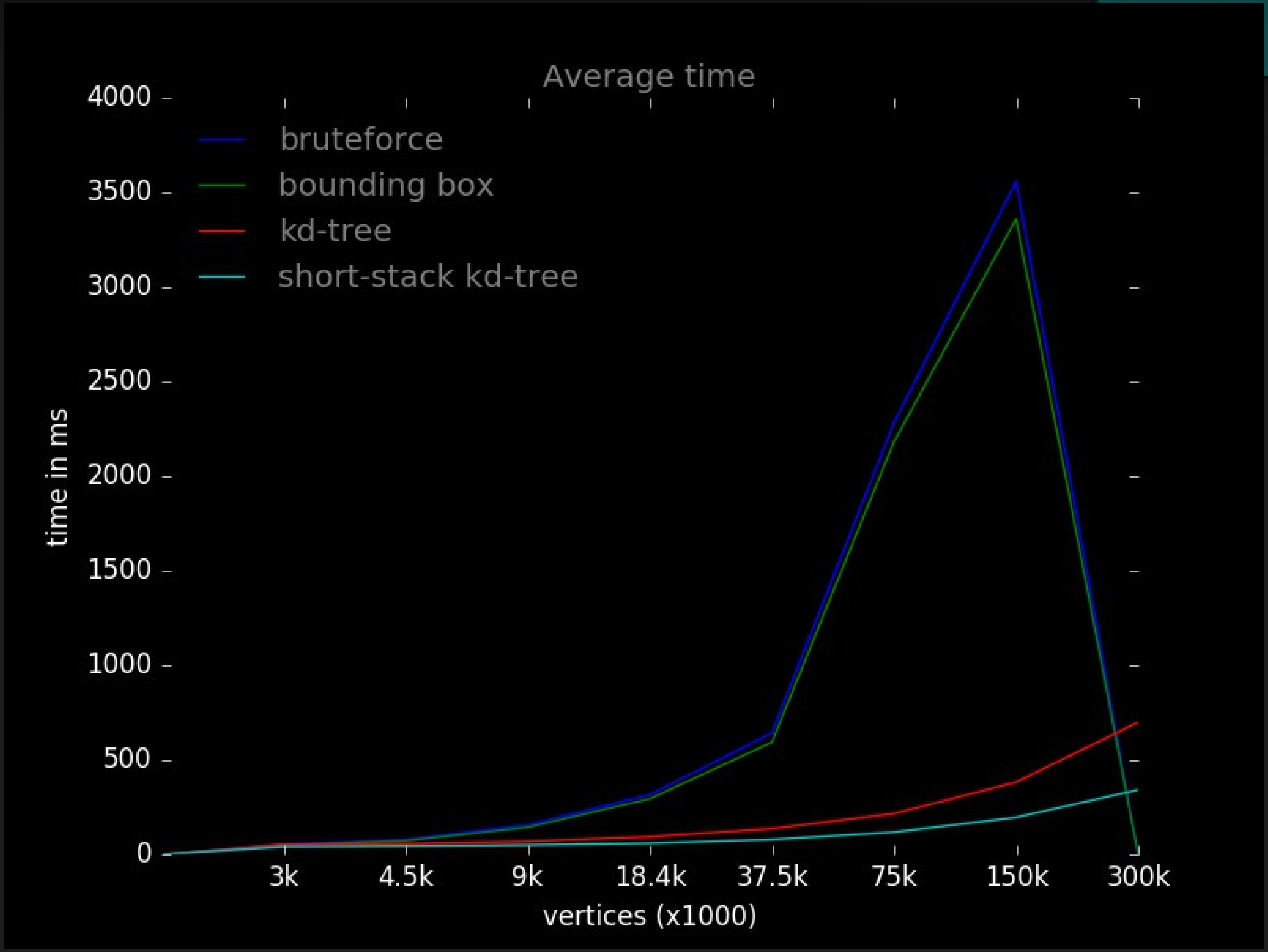
As can be seen, the slowest method is the brute force, followed closely by the bounding box method.  All scale exponentially.

The naive kd traversal offers a 10 fold improvement over both methods while the short-stack method offers a near 20 fold increase in performance.

Notice the last benchmark failed for both the brute force and bounding box resulting in a crash which is is more prominent in the following plot.

# KD Tree Path Tracer Optimization

Rony Edde



# KD Tree Path Tracer Optimization

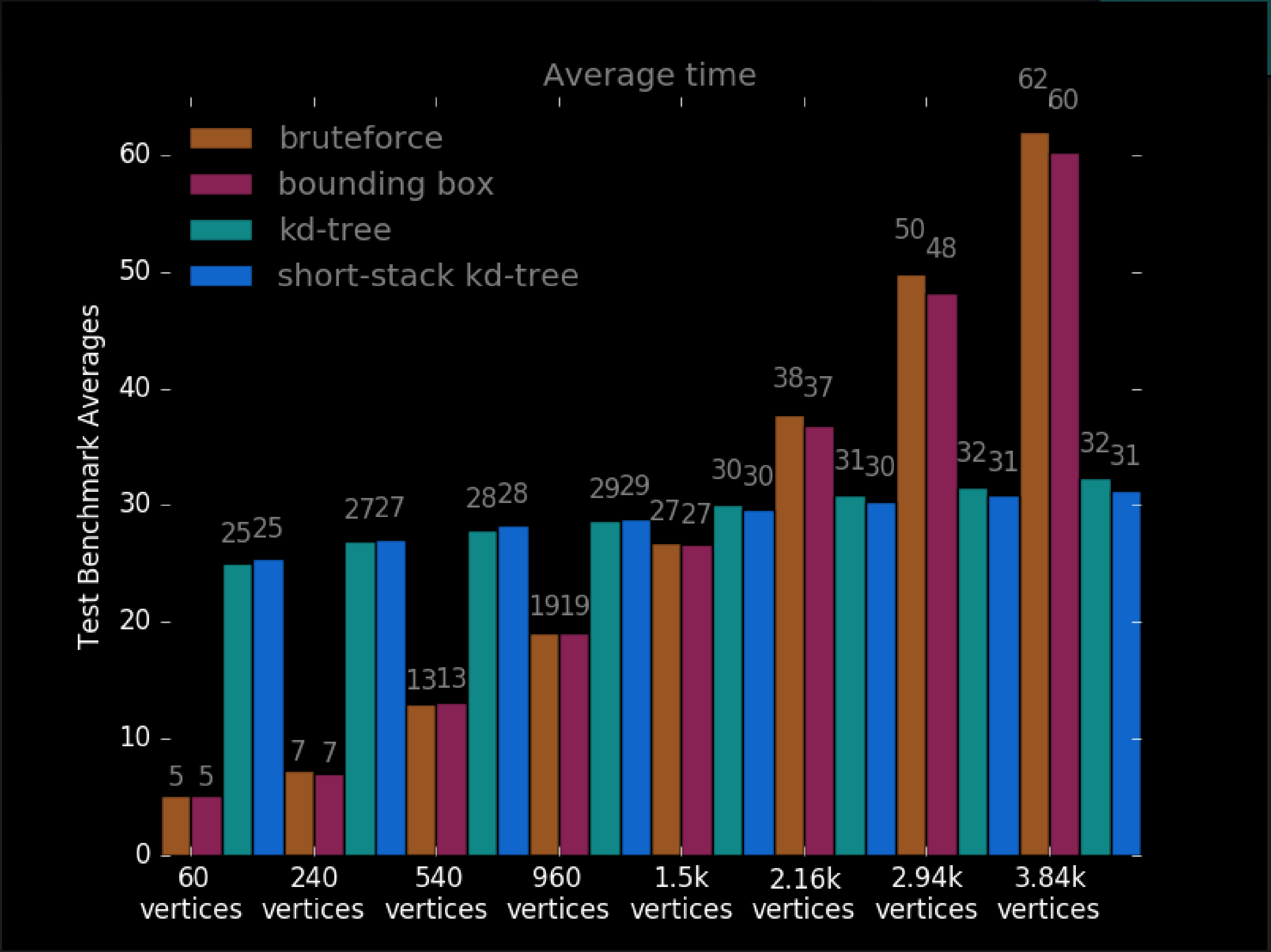
Rony Edde

Performance analysis:

High density geometry shows obvious improvements, however low density geometry should not benefit as much.  Here are results for low resolution geometry

# KD Tree Path Tracer Optimization

Rony Edde



# KD Tree Path Tracer Optimization

Rony Edde

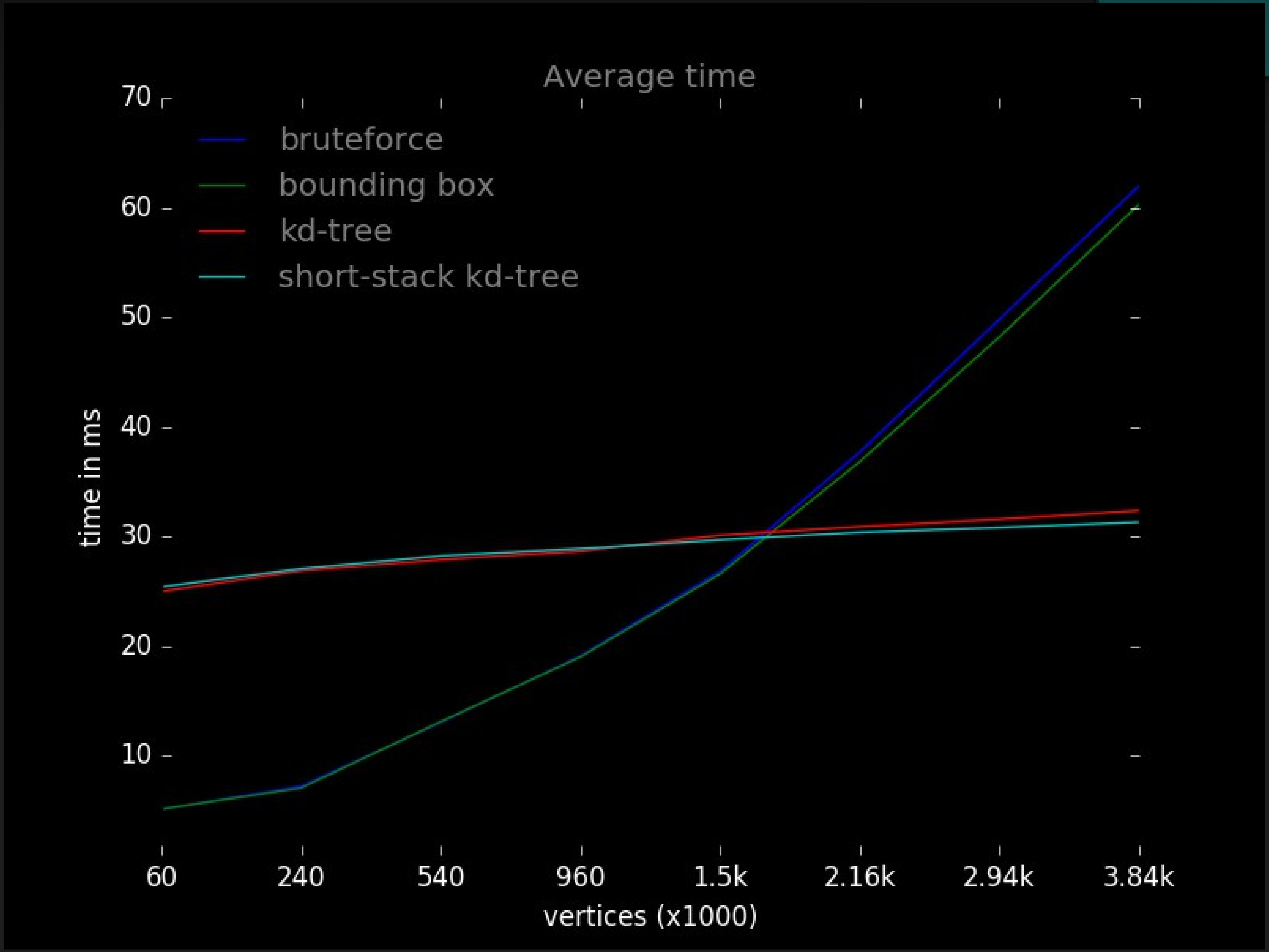
Performance analysis:

Using a kd tree on a low polygon model hinders performance because the additional traversal cost is equivalent to checking all triangle collisions which only adds to the cost.  This is equivalent of a leaf node collision check.

This is due to the minimal cost required to trace the entire geometry.   In these cases, it's best to leave the tree depth to 1 to improve performance.  The limit is around 1.8k polygons.

# KD Tree Path Tracer Optimization

Rony Edde


Average time
bruteforce
bounding box
kd-tree
short-stack kd-tree
time in ms
1.5k
2.16k
2.94k
3.84k
vertices (x1000)