Ray Tracing Optimization

Andrew Dunn, Alec Jackson

https://github.com/csci480-21w/fp-dunn_jackson

Summary

The goal of our project was to incorporate several acceleration methods into the base ray tracer from A2 in order to considerably improve performance with the hope of approaching real-time rendering of scenes. The approaches we used to improve performance were, put succinctly, to render fewer triangles. This would be achieved via two separate mechanisms: firstly, through the use of bounding boxes, and secondly, by way of mesh decimation.

Bounding boxes are, in essence, exactly what they sound like: boxes that encompass an object in a scene. If the bounding box is not intersected by a given ray, then none of the triangles composing the object will need to be checked for ray intersection. This significantly reduces the number of intersections the ray tracer needs to calculate when there are multiple objects in the scene. Computational cost savings are further compounded as the complexity of an object's triangle mesh increases; we were able to render scenes using one sphere, one bunny, and two airships in ~4 seconds while the unoptimized ray tracer took 500+ seconds.

The second approach was mesh decimation. This is an automated process that involves simplifying a mesh when the object is sufficiently small or far from the camera. This can also significantly reduce render times by cutting a large number of triangles out of the scene since the fine detail of a complex mesh will be unseen from a significant "scene distance".

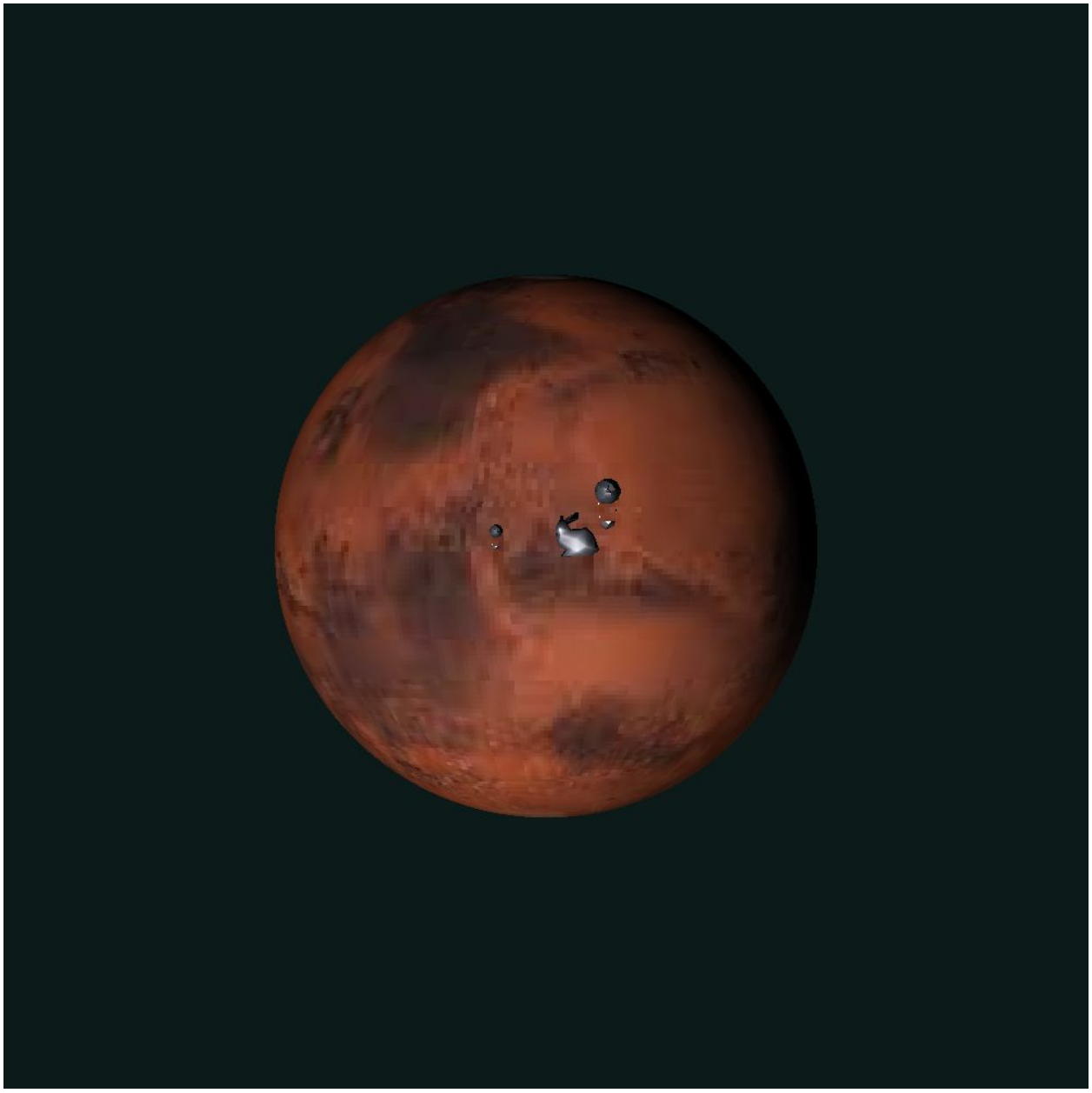
Our final deliverable relies much more heavily on the usage of bounding boxes than mesh decimation to achieve its optimized performance. This approach was chosen for a variety of reasons, not the least of which being that on-the-fly mesh decimation is a computationally intensive algorithm, so preprocessing mesh decimation seems to be preferred for more complex polygon meshes. Additionally, since the base ray tracer doesn't utilize interpolation techniques to smoothly draw lines, the overall appearance of a decimated mesh at distance is entirely indistinguishable from a complete mesh as both look equally pixelated. We also saw more dramatic performance improvements from the Bounding Volume Hierarchy (BVH) implementation, which stands to reason as BVH methods can exclude entire structures from being rendered unnecessarily. Unfortunately, we were unable to achieve real-time rendering of scenes with anything more than a few spheres, but given more time to accumulate other optimization techniques, we remain optimistic that the A2 ray tracer could be made competitive.

Results

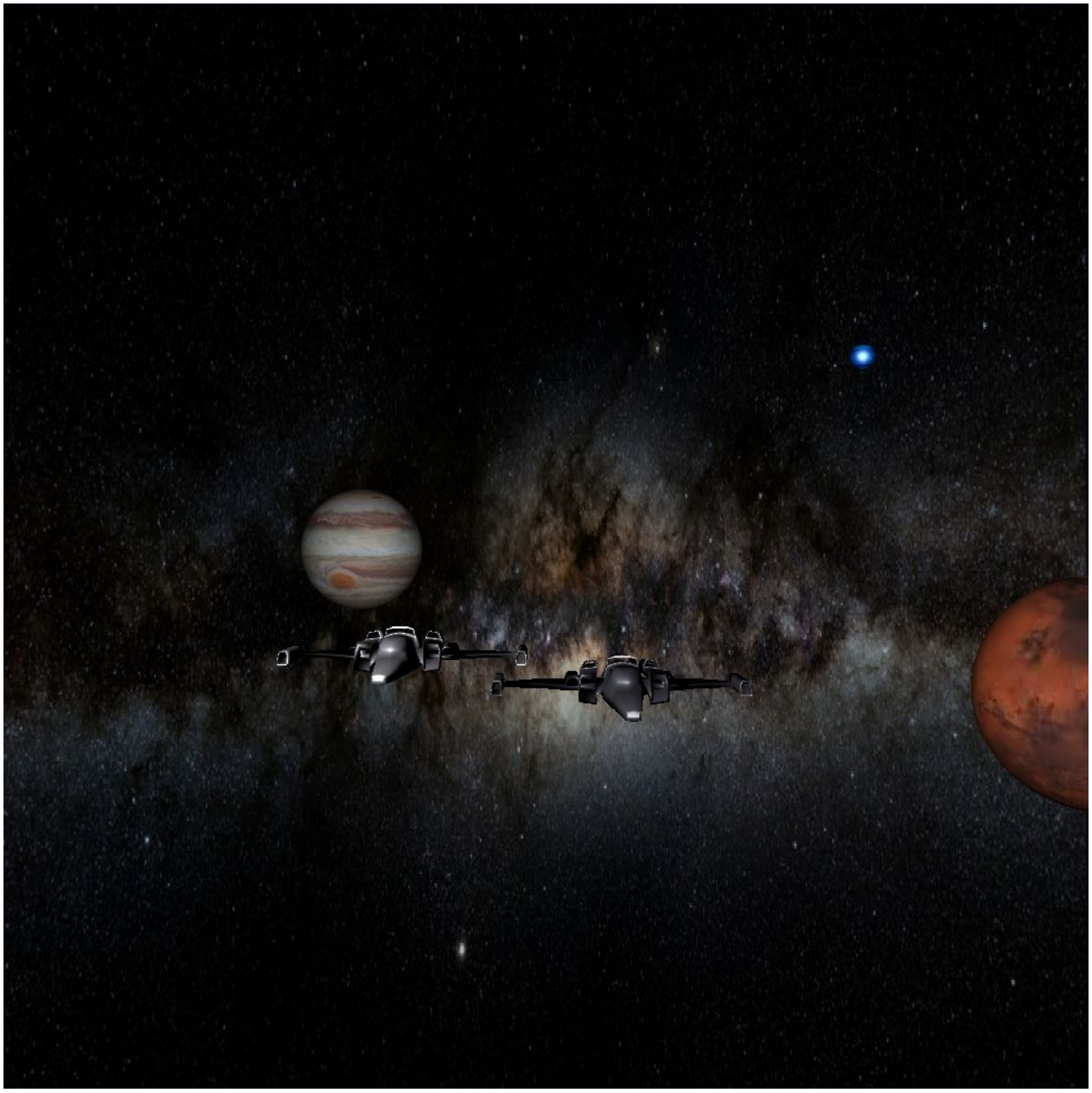
Bounding Box - bvhTest from the Julia REPL

```
julia> WWURay.bvhTest(true)
Running BVH comparison test!
~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~
First Scene- Simple Space Scene
Benchmark:
Scale:    (1-2-3-4-5-6-7-8-9-0)
Progress: (*~*~*~*~*~*~*~*~*~*)
Done!
532.930632 seconds (4.45 G allocations: 265.141 GiB, 5.72% gc time)

BVH Demo:
Hierarchy assembled
Size: 4 objects
Scale:    (1-2-3-4-5-6-7-8-9-0)
Progress: (*~*~*~*~*~*~*~*~*~*)
Done!
  3.800739 seconds (36.65 M allocations: 3.799 GiB, 7.40% gc time)

~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~
Second Scene- Complex Space Scene
Benchmark:
Scale:    (1-2-3-4-5-6-7-8-9-0)
Progress: (*~*~*~*~*~*~*~*~*~*)
Done!
517.323267 seconds (4.90 G allocations: 290.738 GiB, 2.12% gc time)

BVH Demo:
Hierarchy assembled
Size: 6 objects
Scale:    (1-2-3-4-5-6-7-8-9-0)
Progress: (*~*~*~*~*~*~*~*~*~*)
Done!
 32.355619 seconds (404.48 M allocations: 23.836 GiB, 7.43% gc time)

~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~*~
Third Scene- No OBJMeshes (Benchmark should win)
Scale:    (1-2-3-4-5-6-7-8-9-0)
Progress: (*~*~*~*~*~*~*~*~*~*)
Done!
  1.271541 seconds (15.47 M allocations: 736.371 MiB, 7.17% gc time)

BVH Demo:
Hierarchy assembled
Size: 9 objects
Scale:    (1-2-3-4-5-6-7-8-9-0)
Progress: (*~*~*~*~*~*~*~*~*~*)
Done!
  1.401279 seconds (10.22 M allocations: 1.345 GiB, 6.32% gc time)

Test Concluded!
julia>
```

Optimized results from Scene 1:

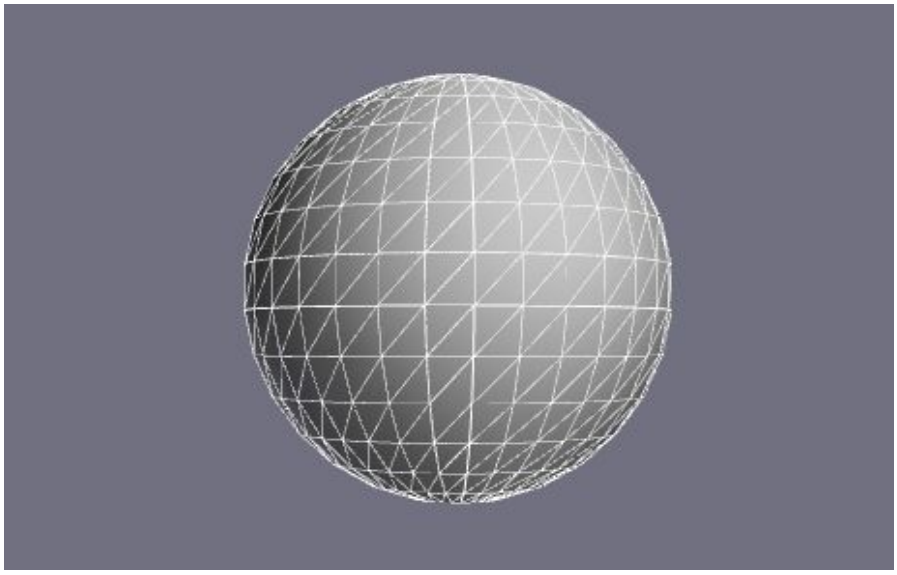


Optimized results from Scene 2:

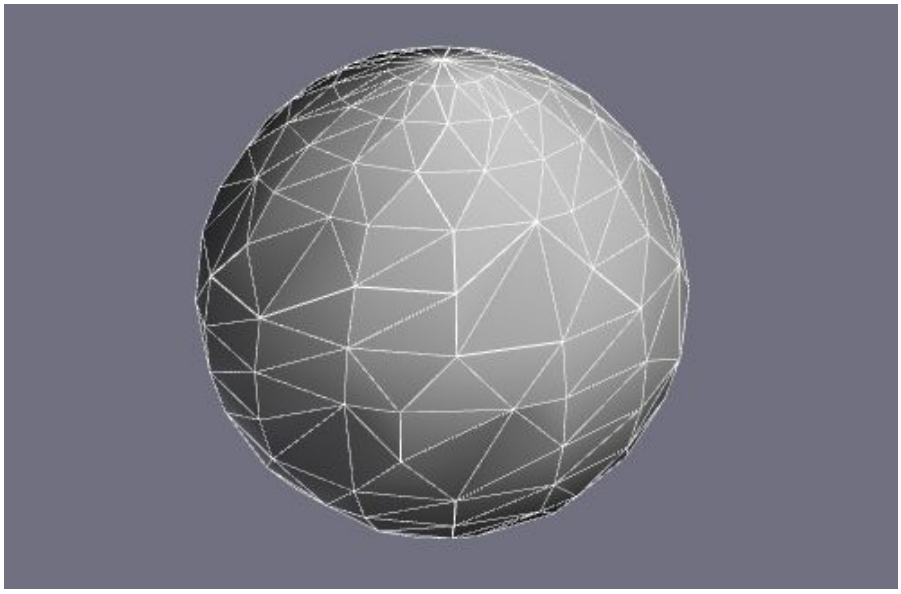


Mesh Decimation
Sphere Mesh Results

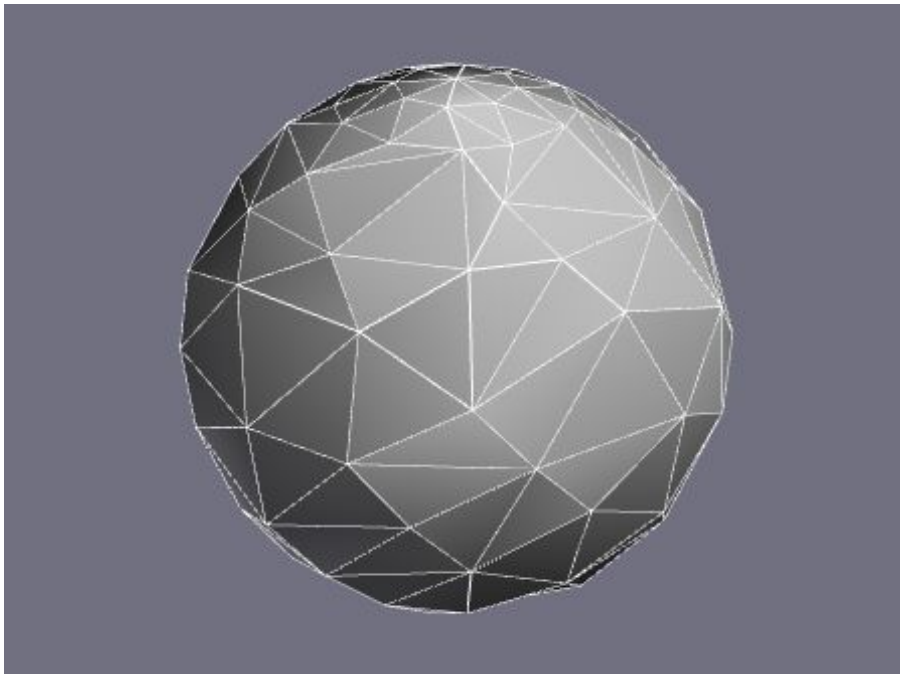
Original - 896 tris



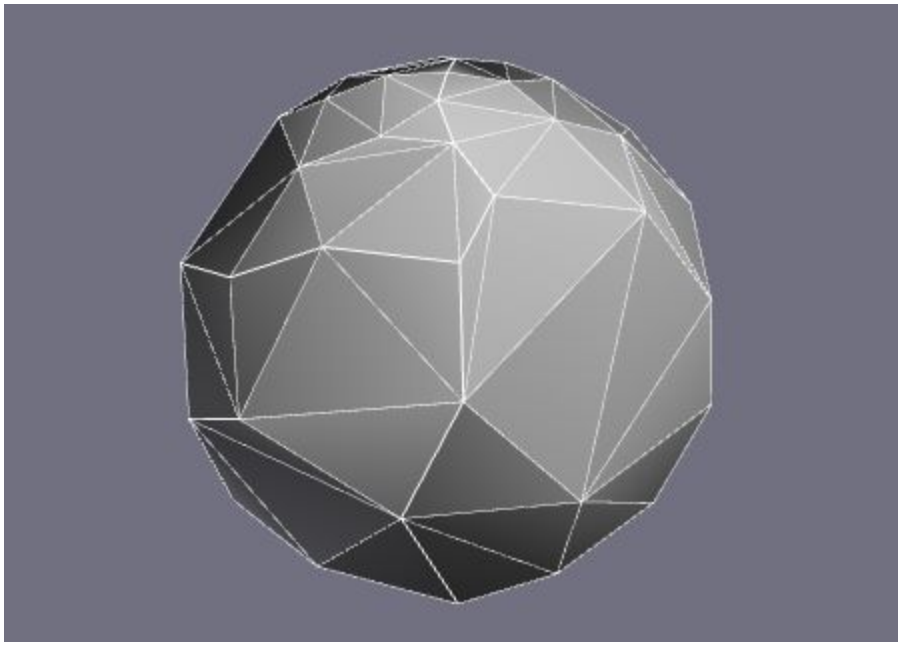
Decimation Round 1 - 464 tris



Decimation Round 2 - 250 tris

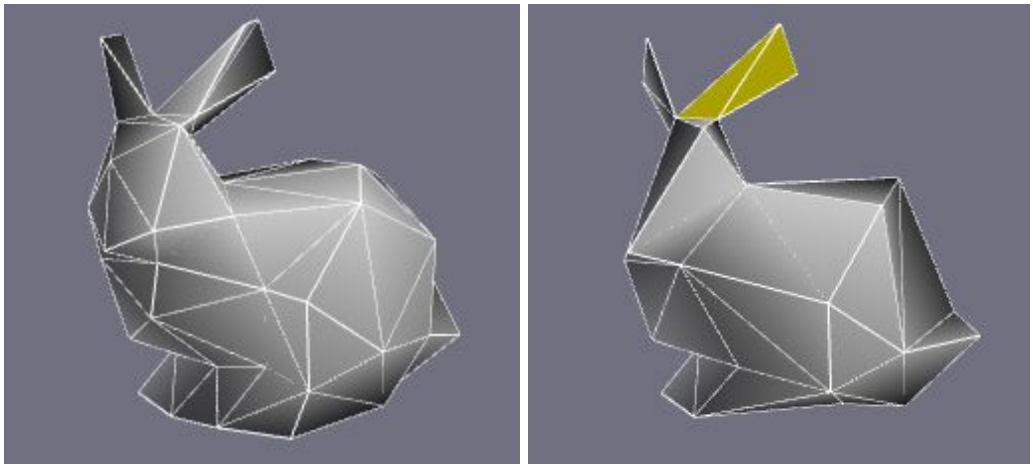


Decimation Round 3 -134 tris

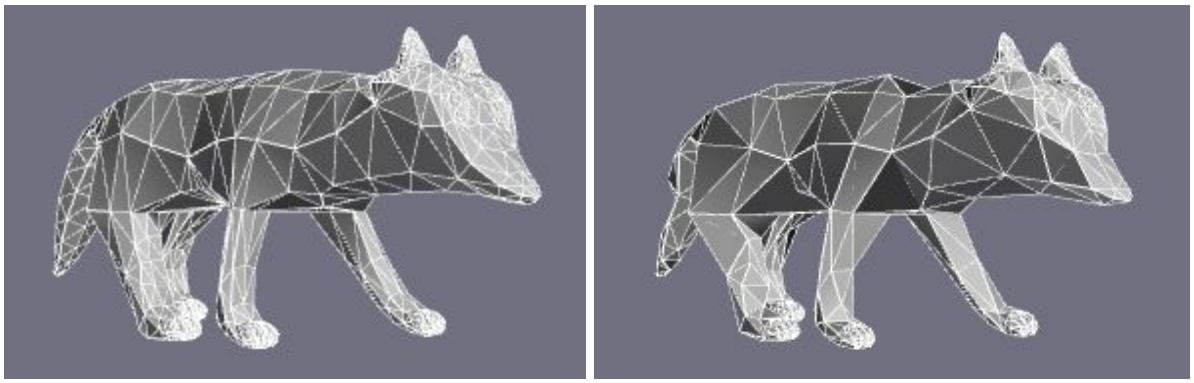


Other Results

Bunny - 114 tris to 58 tris



Wolf - 1928 tris to 1009 tris



<u>Instructions</u>

Bounding Box

The BVH functionality was built into the existing ray tracer in such a way that utilizing the BVH accelerations simply requires passing an additional argument of “true” to any previously recognized call to the WWURay.main() method. An example call using this functionality would be:

WWURay.main(12, 7, 1000, 1000, “results/test.png”, true)

Note that camera_7 is preferred for scene #12 and camera_5 is preferred for scene #13 for best results. Most of the other test scenes from A2 should be callable using BVH, but all remain untested so results are not guaranteed.

For comparing against non-accelerated ray tracing, providing “false” as a final argument will disengage the BVH constructor, and the default argument is set to “false” as well.

Mesh Decimation

The mesh decimation can be called independently of any ray tracing by calling the "decimateMesh()" function. This function takes 2 arguments. The first is a string naming the file name without extension inside of the data directory. For the mesh bunny.obj inside of data you would simply call "bunny". File paths can be added to the beginning of the string to navigate. The second argument is the number of rounds of decimation you want to run on the mesh. One run has a large compressing effect on the mesh, so running multiple iterations will most likely result in a very compressed mesh. This input will default to 1, so can be left blank in most cases.

Code Guide

Bounding Box

The original design schema for our BVH implementation was a bit half-baked in its concoction; there isn't really a good way to implement a BVH structure as a single module without rerouting all of the functionality of WWURay.jl through the BVH module. What we wound up implementing is mostly housed in the Scenes.jl module in order to piggyback on the ray_intersect functionality that's implemented there. The final version of our BVH implementation could actually be completely detached from Bound.jl and rehomed in Scenes.jl, but disentangling all of the snippets and retesting at the last minute seemed ill advised.

The BoundVol data type is defined in Bound.jl, and several helper methods used in determining the maxima and minima of the bounding boxes are also housed there. Much of the code in Bound.jl, as mentioned above, could be deprecated and moved elsewhere, though.

Scenes.jl does the vast majority of the heavy lifting to implement the BVH: build_hierarchy constructs a hierarchy of BoundVols from the Scene object that is passed in; the overloaded bound_object method handles the brunt of constructing a nice BoundVol to snugly fit the object passed in; bound_builder takes a min Vec3 and a max Vec3 and builds a cube out of the vector between the points; hit_box is the method behind the magic since it encompasses the rewrap of ray_intersect to deal with BoundVols, and is therefore quite a behemoth (with two helper methods of its own, no less).

Other minor modifications were made to WWUMeshes.jl to add fields to the OBJMesh data structure as well as adding handling of those fields to the different constructor methods in the module. A couple of methods for showcasing the BVH accelerations were added to WWURay.jl, and minor changes to allow for optionally passing an additional boolean argument to the main method were also added. TestScenes.jl had two additional cameras added as well as three additional scenes to test and showcase the accelerations.

Mesh Decimation

All mesh decimation is handled inside Decimate.jl. The process starts by loading a mesh, this mesh then gets passed to find edges. This will find all unique edges in the mesh. We'll now take all the edges in the mesh and find the respective euclidean distances of them. Then a sort is performed to sort our array of edges by their distances. Now that we have a sorted array of edges the next step is to remove any edges that reference a vertex that has already been referenced. This is done after sorting so that we'll remove closer points first generally maintaining the shape of the object. The final step is to go through our list of viable removal points and change the value of one or our original points to the new point value. Then we'll need to change the index section following 2 rules. And reference to the second index is changed to the first. And triangles that reference both points are deleted.