M2 Progress

- Edge detection for shadows and reflections
- Edge detection has a locality parameter to allow for thicker lines. Default is 0.0 and anything above 2.0 is too thick to be useful
- Convolution filtering can be applied
- Josh does not want to write the Oren Nayer diffusion model. Instead, Josh wrote convolutional filtering. Is this acceptable?
- Area lights have been implemented for Lambertian surfaces. They are defined by an intensity, a position, and two vectors. Sampling rate is currently hard-coded to 5, which provides nice soft shadows at the scale we are working at.
- Bounding Volume Hierarchies implemented for improved test scene render times.

Run Commands

Rays.main(7, 1, 300, 300, "results/bunny.png")

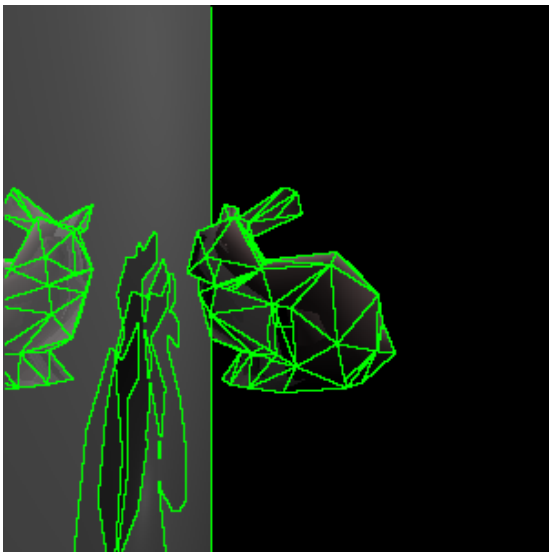
Rays.blur("conv_imgs/treebranch_source.jpg", "conv_imgs/treebranch_convolution.jpg", "box_blur_33")

Rays.aa_run(“results/bunny_stratifedAA_2x.png”, “s”, 2, false)

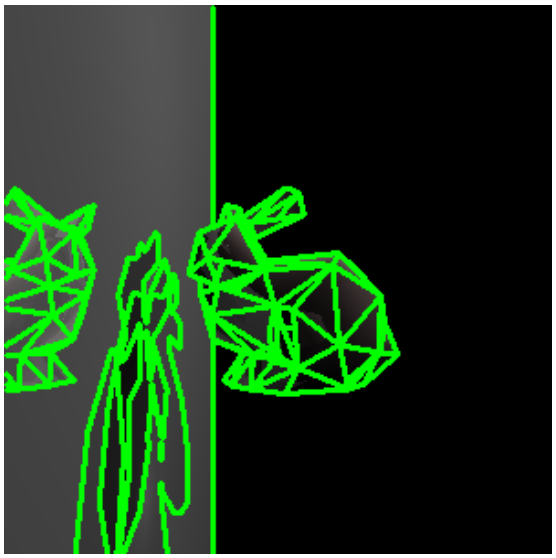
Rays.main(11, 1, 300, 300, "results/sphereRefraction1.png", 1, 16)

Rays.main(5, 1, 300, 300, "results/sphereRefraction2.png", 1, 16)

Visual Output



Edge Detection for Reflection and Shadows



Chunky edge detect on Bunny locality = 1.0

Edge Detection with Convolutions

Box Blurring Convolution

Detailed Description

The locality parameter was implemented by inscribing a circle onto my boolean mask. The edge detection algorithm is run as usual, but this time

mask[i, j] = true was replaced with a call to draw_circle!(). This draws a pixelated circle with radius = locality; center = (i, j).

To implement edge detection for shadows and reflections I created a custom object. This stores the object id of each object that is hit during the recursive reflection method into a Vector{Any}. Shadows are dealt with by storing the number of shadows at each position in the scene. Each light source generates its own shadow. When we implement area lights, there will be a large number of light sources affecting each pixel and this algorithm will detect shadow edges everywhere. However, the transition from light to dark will be smooth already, eliminating the need for anti-aliasing, and thus no motivation for shadow edge detection. I have already added a toggle to turn shadow edge detection on and off.

Convolutional filtering is implemented in two parts. I have a method to apply any sized convolution to an image. Then I have a method which generates any sized box blur matrix or gaussian matrix. Two 3x3 edge detect matrices are also included. Run convolutional filtering on my tree branch image by running the command:

Rays.blur("conv_imgs/treebranch_source.jpg",
"conv_imgs/treebranch_convolution.jpg", "box_blur_33")

Acceptable Convolution Type Inputs

- box_blur_{odd, positive integer}
- gaussian_blur_{odd, positive integer}
- edge_detect_1
- edge_detect_2

Bounding Volume Hierarchies (BVH)

The primary purpose of the BVH implementation in this project is to improve the performance of rendering scenes in TestScenes.jl. It does this by organizing all geometric objects in the scene into a structure similar to a binary

tree, with each node in the tree representing a “bounding volume” that contains some of the geometric objects in the scene.

To implement a BVH, the algorithm first implements an “Axis-Aligned Bounding Box” (AABB) which encloses a given set of geometric objects in the scene. The dimensions for these boxes are usually defined by the coordinates of objects on the farthest end of each box. These AABBs’ represent a single node in the BVH tree. The algorithm generally knows how to partition geometric objects into AABBs’ by sorting them (on the x-axis in this case) relative to the bounding boxes “centroid” (the center of the bounding box). Objects that are located to the left of the center of a box are sorted into nodes on the left side of the tree, and objects that are located to the right of the centroid are sorted into nodes on the right side of the tree.

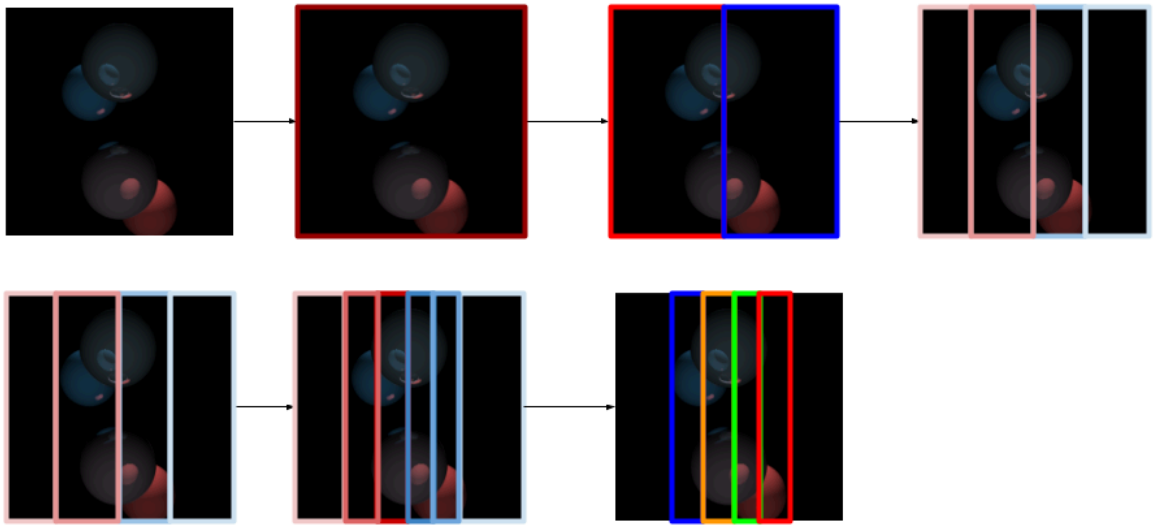


Figure A: Example of AABB bounding boxes on an image of test scene 11. The solid blue and red spheres would be sorted into the blue and red boxes respectively, while the two refractive spheres would be sorted into either the orange or green boxes, depending on how close each sphere is to the center of each box. If one object is in the exact middle of the two boxes, the object would usually be sorted into the box on the left (orange). Otherwise, if multiple objects are located in the exact middle between two boxes, the algorithm will attempt to evenly sort the objects into boxes.

In this case, the BVH algorithm starts with one large bounding box (which we'll call *a* and which we'll assume has a size of *n*) composed of all the objects in a test scene. This box *a* is then split into two smaller boxes (*a1* and *a2* with each box having a size of $\frac{n}{2}$), and the objects on each side of the previous bounding box are then further sorted into smaller bounding boxes, depending on where they are relative to the centroid of the smaller bounding box. This continues on until each object is ideally located in its own bounding box. (Things may become complicated if two objects are significantly overlapping). In this state, each node will either be a "parent node" (a node that stores an AABB as well as pointers to its two left and right child nodes) or a "child node" (a node that stores an AABB that also does not contain any left/right child nodes to point to).

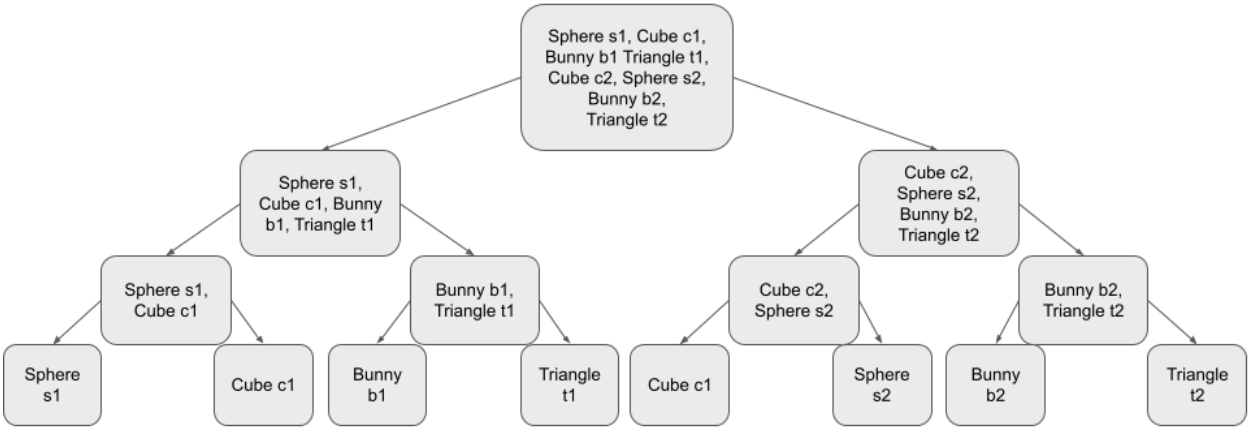


Figure B: Example depiction of a BVH in tree form. Assuming there are 8 objects in the scene and each object is sorted into an array where Sphere s1 is the object closest to the farthest left edge of the scene and triangle t2 is the object closest to the farthest right edge of the scene, the process of separating each object into its own AABB goes as follows: *Layer 1: 8 objects in one AABB (eight objects per AABB) → Layer 2: 8 objects in two AABBs (four objects per AABB) → Layer 3: 8 objects in four AABBs (two objects per AABB) → Layer 4: 8 objects in eight AABBs (one object per AABB)*.

The BVH can then recursively traverse through the hierarchy and test individual AABBs to determine if a ray intersects an AABB. Once the closest intersection point between an AABB and a ray is found, the algorithm terminates traveling along that branch of the BVH to reduce the number of redundant ray-intersection computations, thereby potentially saving a considerable amount of time.