Final Project Report

Josh

1. Edge detection using object IDs from our raytracer. Parameters: thickness and a toggle to include shadow edges.
2. Convolutional filtering– any odd sized matrix for box blur and gaussian blur. Two versions of edge detection convolution filtering.
3. Gaussian splats. I use stochastic gradient descent to learn scale_x, scale_y, and luminance.
4. Nearest neighbor splatting. I find the nearest neighbor using Fortune's algorithm, an O(n log n) tiling for Voronoi Diagrams. I pulled the algorithm from https://github.com/arthursantana/mac0499

Shahu

5. Refraction– using Snell's Law
6. Bounding Volume Hierarchy– Algorithmic technique which uses a Binary Tree structure to recursively define bounding boxes.

Jackson

7. Anti-aliasing using 3 sampling techniques: uniform, random, and stratified.
8. Pipelining between edge detection and anti-aliasing systems to allow for edge-only anti-aliasing of variable thickness.
9. Area lights using stratified sampling from a parallelogram, and resultant soft shadows.

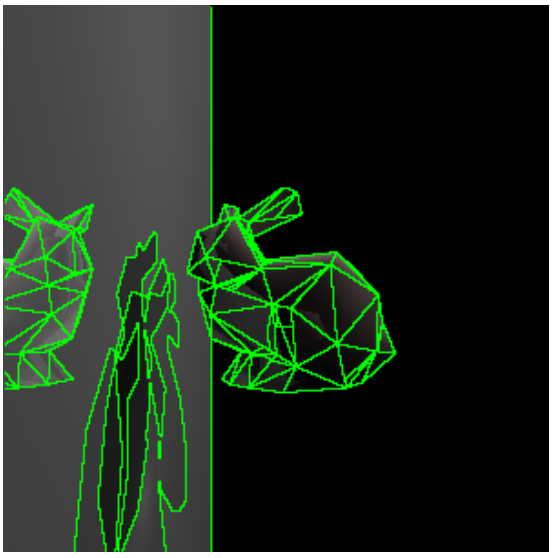
Edge Detection

I update traceray to store object ids of each object encountered along the ray's path. A pixel is considered an edge if the coordinate above, left, or above-left contains a different object id. The locality parameter works by drawing a circle of radius L at each edge coordinate. The output of this function is a boolean matrix, where true is an edge location. I've adjusted this method to also draw green on edges to create the visual output below.

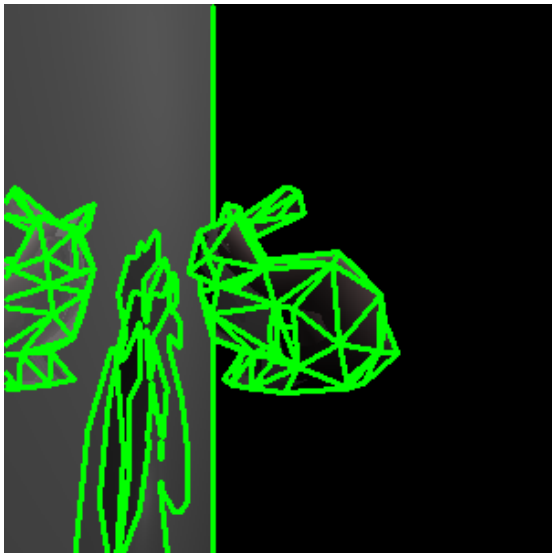
Edge detection works with our Anti-Aliasing filters to apply AA only at the edges of our rendered images.

Run Command: Rays.main(300, 300, “results/Anti_Aliasing_Results”, “edge_detect”, “uniform”, 4, true)

Image Location– ./Rays/results/Anti_Aliasing_Results



Edge Detection for Reflection and Shadows



Chunky edge detect on Bunny locality = 1.0

Convolutional Filters

Apply convolutional filters to a source image and save to an outfile location. I implement box blur and Gaussian blur for any odd sized matrix. I also have two edge detection convolutional filters. Finally, I wrote capabilities to multiply by a scalar (brighten).

Run Command– Convolve.blur("conv_imgs/treebranch_source.jpg", "conv_imgs/treebranch_convolution.jpg", "box_blur_33")

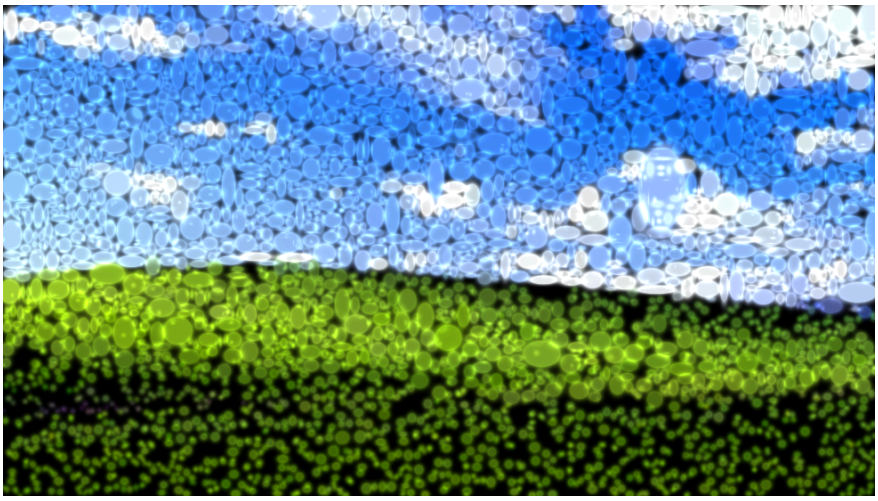
Image Location– ./Rays/results/Convolutions

Allowed Convolutions

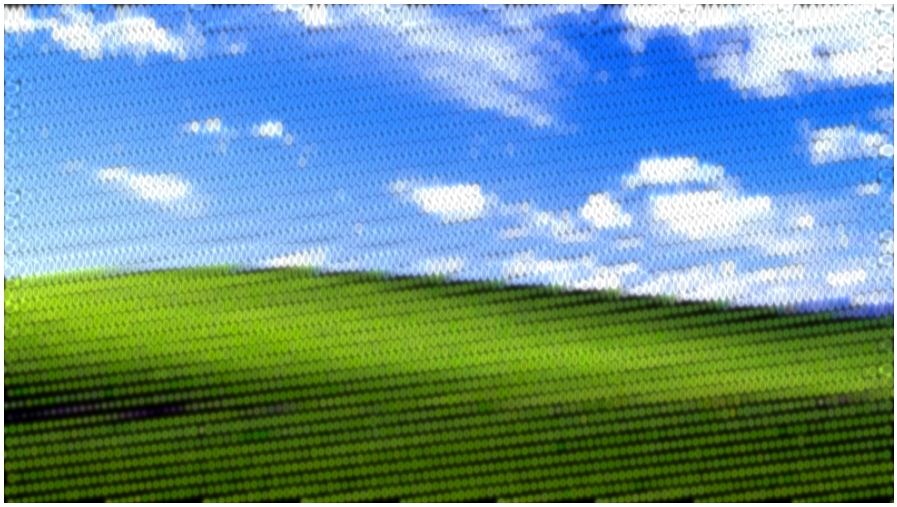
- brighten_{Float}
- box_blur_{odd, positive int}
- gauss_blur_{odd, positive int}
- edge_detect_1
- edge_detect_2

Splatting– Implementation Details

I tried to replicate a source image using SGD and Gaussian splats. I began by sampling colors at uniform points from a source image. Then I iterated Stochastic Gradient Descent, allowing for x-scale and y-scale to be learned. See Appendix for SGD Equations.



7,000 epochs. Random Initialization of 5000 blobs.
Learning scale x, scale y, luminance



14,000 epochs. Uniform Initialization of 5000 blobs.
Learning scale x, scale y, luminance

The biggest issue with Gaussian splats is the background color which peeks through around the borders. I was hoping I would be able to get a blending effect between Gaussians at the borders, but I haven't been able to train this behavior yet. This introduced the idea that I could interpolate an image by assigning a color based on the nearest splat. This is actually a popular mathematical problem: Tessellation of Voronoi Diagrams. Arthur Santana has a nice Julia implementation on Github which utilizes Fortune's algorithm to perform Voronoi Diagram tessellation in O(n log n) time.

Run Commands:

cd Splatting
include("splat_gauss.jl")
splat_gauss.control_panel()

cd Splatting
include("splat_neighborhood.jl")
splat_neighborhood.control_panel()

– All model parameters are available to edit within control_panel()

Image Location– ./Splatting/imgs/

Nearest Neighbor Splatting

These images take a very long time to render. I was not able to utilize the speed of Fortune's algorithm because the mac0499 repo utilizes this strange polygon data type to represent a region. I had to use matplotlib as an intermediary with ax.fill() then save as a .png then load the image into Julia as a Matrix{RGB{Float32}}. These data conversion steps slow down my code.

I wrote a baseline Voronoi Tessellation algorithm which iterates through every pixel, then iterates through every splat to find the nearest splat. This naive implementation is slow and is used for baselining purposes.

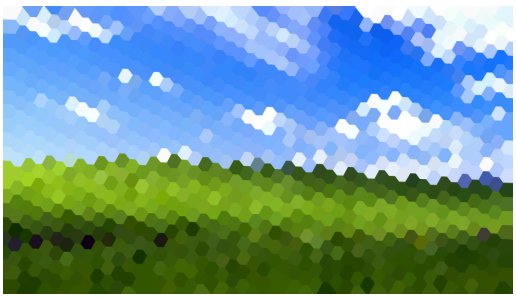
Fortune's algorithm tessellates Voronoi Diagrams much faster than baseline, but nowhere near quick enough to train using Stochastic Gradient Descent. Additionally, I wouldn't be able to take partial derivatives of this function

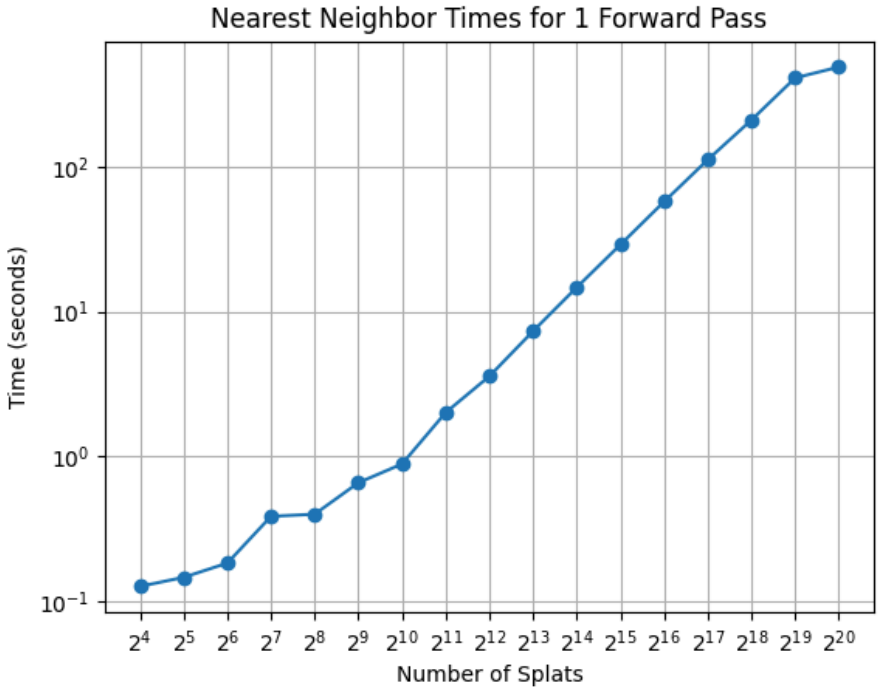
because I don't know how to take the derivative of "find the nearest splat for this pixel."

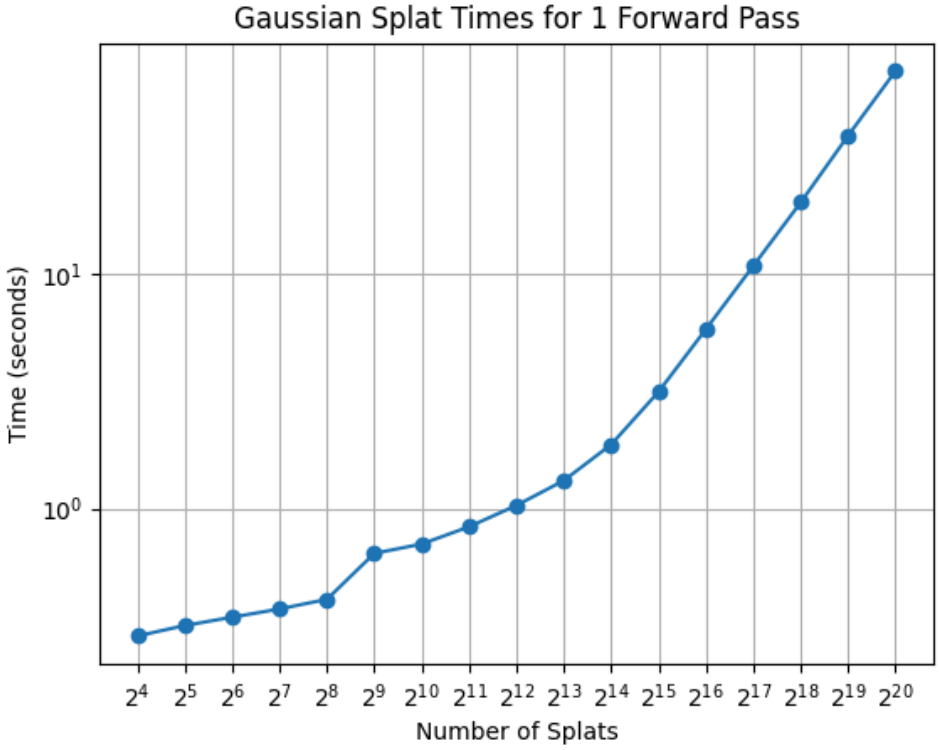
Gaussian Splats performs four powers of 2 faster than Nearest Neighbor interpolation using the Mac0499 repo.

	Gaussian Splats	Mac0499 nearest neighbor	Naive nearest neighbor
Seconds to complete 1 forward pass with 5,000 splats	0.05 seconds	9 seconds	592 seconds

	Random Initialization	Uniform Initialization
50 splats		
250 splats		

1,000 splats		
10,000 splats		
1 Million		


Nearest Neighbor Times for 1 Forward Pass
Time (seconds)
10^2
10^1
10^0
10^{-1}
2^4 2^5 2^6 2^7 2^8 2^9 2^{10} 2^{11} 2^{12} 2^{13} 2^{14} 2^{15} 2^{16} 2^{17} 2^{18} 2^{19} 2^{20}
Number of Splats


Gaussian Splat Times for 1 Forward Pass
Time (seconds)
10^1
10^0
2^4 2^5 2^6 2^7 2^8 2^9 2^{10} 2^{11} 2^{12} 2^{13} 2^{14} 2^{15} 2^{16} 2^{17} 2^{18} 2^{19} 2^{20}
Number of Splats

Splatting Future Expansions

Given more time I would implement the following features for my splatting:

- Cloning and Splicing of Gaussian blobs
- Hyperparameter sweep to determine best learning rate and batch size. Currently, I tested some arbitrary values until I found something sensible.
- Create and test new loss functions. My loss function is simply the difference between ground truth and my model forward pass. I sum across the RGB channels, so I'm really optimizing for brightness. I'm curious if I could make a multispectral loss equation to optimize towards the correct wavelength of light.

Refraction

Refraction is generally defined as the bending of light rays as it passes from one medium into another medium that possesses a different *Index of Refraction* (IOR). Notable examples of this effect include light rays passing from air into water, or from air through glass. In both cases, the light rays "bend" since the IOR of air (1.0003) is different from the IOR of glass (1.5), and the IOR of water (1.3333).

To calculate refraction, we first need to detect when a light ray intersects with an object in the scene. At the point of an intersection, we take the intersection point of the ray and the object as well as the surface normal at the intersection point. We'll also get the material at the point of intersection and make sure it is translucent, as solid (opaque) objects are generally not known for refracting light rays.

Assuming the point of intersection has a translucent material, there are two possible outcomes for the light ray. If the light ray hits the object at a "shallow" angle, we can calculate the new direction of the ray using *Snell's Law.* Snell's Law is calculated with the formula $n_1 sin(\theta_1) = n_2 sin(\theta_2)$, where

n_1, n_2 are the refractive indexes of both mediums, θ_1 is the angle of incidence (the angle at which the light ray enters the object), and θ_2 is the angle of refraction (the angle the light bends at through the refractive object).

Otherwise, if the light ray hits the object at a "steep" angle, the ray will undergo a process known as *Total Internal Reflection* (TIR), in which the light ray will be completely reflected away from the object instead of passing through the object at any point. One would calculate TIR with the formula $\theta_c = sin^{-1}(\frac{n_2}{n_1})$, where n_1, n_2 are the refractive indexes of both mediums, and θ_c is the critical angle between the light ray and the point of intersection.

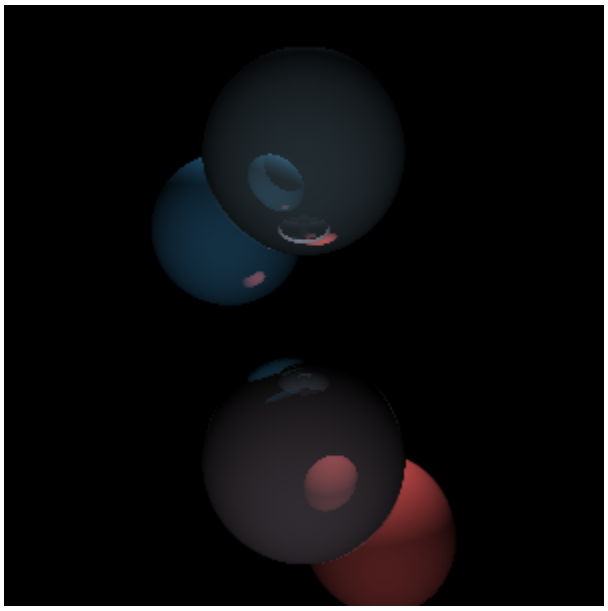


Figure A: An example of the refractive effect occurring. In this situation, two solid colored spheres (transparency = 0.0, IOR = 1.0003), are positioned at an offset angle relative to two glass spheres (transparency = 0.5, IOR = 1.5) such that part of each solid sphere is located just behind part of each glass sphere. As light travels through each sphere, the light rays are "refracted" such that the parts of the solid spheres located behind each glass sphere appear to be distorted.

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.

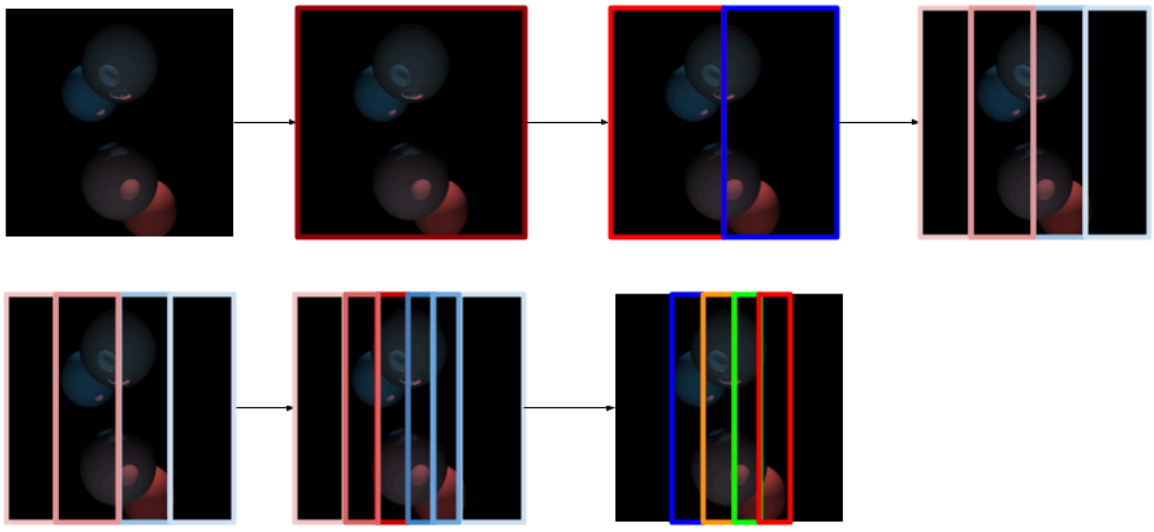


Figure A: Example of AABB bounding boxes on an image of test scene 11. The solid blue and red spheres would be sorted into the blue and red boxes respectively, while the two refractive spheres

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).

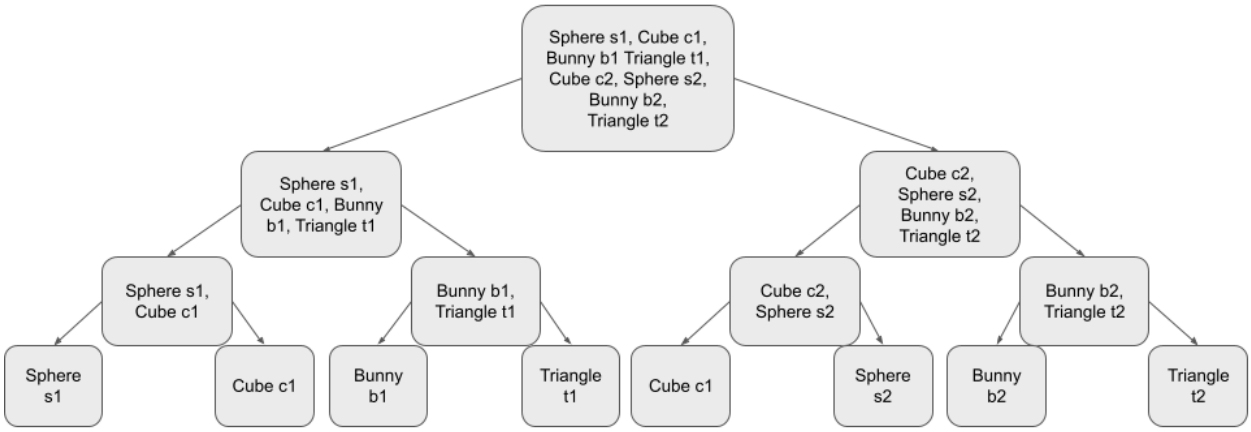


Figure B: Example depiction of a BVH in tree form. Assuming there are 8 objects in the scene and each object is sorted into an array where Sphere s1 is the object closest to the farthest left edge of the scene and triangle t2 is the object closest to the farthest right edge of the scene, the process 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.

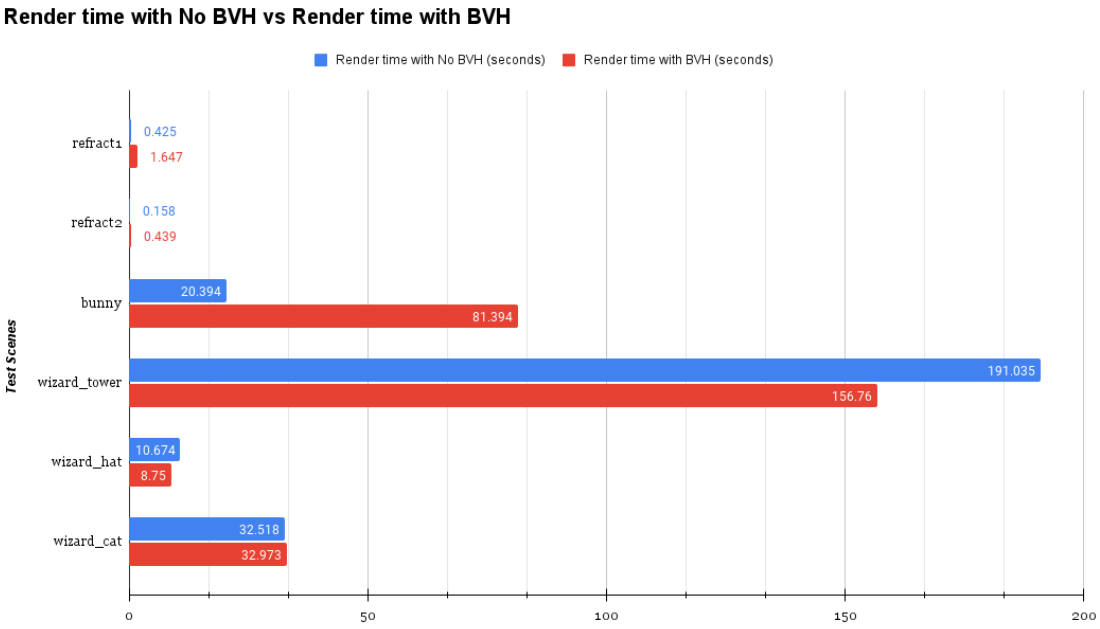


Figure C: Render times of six scenes using BVH (red) compared against the same six scenes without using BVH (blue). The smaller the bar is, the lower the render time is.

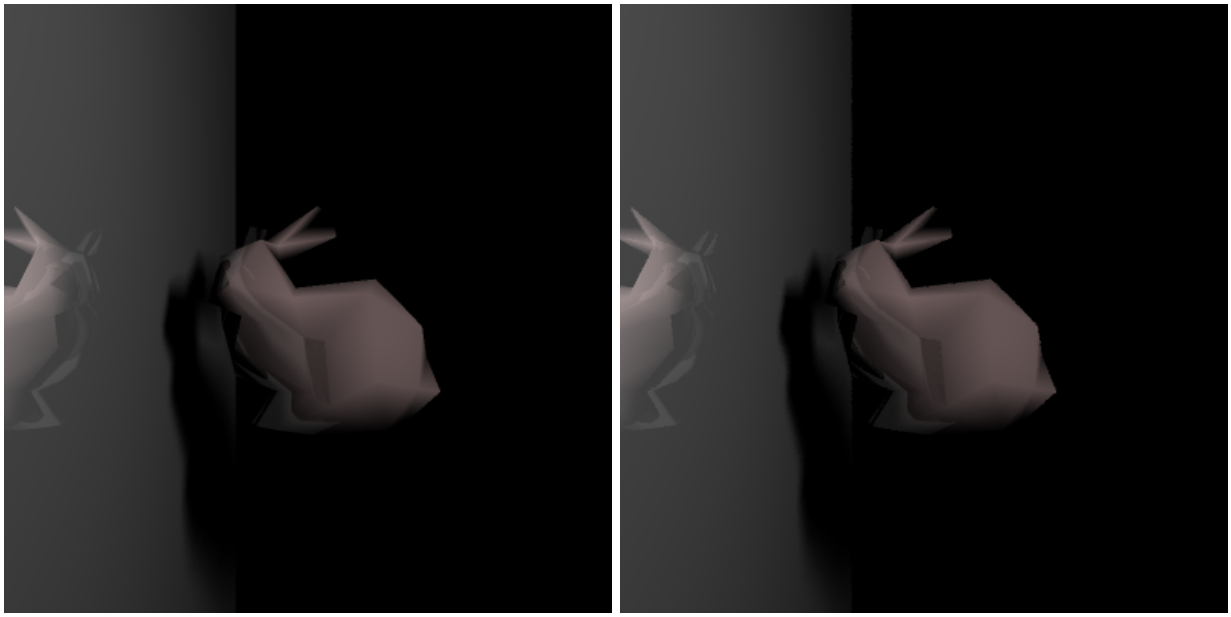
Anti-aliasing

The purpose of anti-aliasing is to smooth rough edges in an output image of a scene, allowing for a reasonable rasterized approximation of the continuously-defined scene with lower image resolution than would otherwise be necessary. This is accomplished by taking multiple samples per pixel ("supersampling") by casting more than one ray for each pixel on the screen. Every sample for a given pixel is then averaged together to produce the output pixel color.

The implementation of anti-aliasing is straightforward. Rather than sampling once in the center of each pixel to determine its color, a sampling rate chosen by the user determines how many samples will be taken for each pixel. The number of samples is equal to the square of the chosen sample rate (e.g. a sample rate of 2 results in 4 samples per pixel). This is so that sampling is evenly spaced across the pixel, giving the most even subpixel distribution for the multiple samples. Our implementation provides three methods for sampling: uniform, random, and stratified. Uniform sampling samples from consistent, evenly spaced positions within a given pixel. These positions are given by the ordered pair $(i + \frac{p + 0.5}{n} - 0.5, j + \frac{q + 0.5}{n} - 0.5)$, where i and j are the pixel coordinates, p and q are the vertical and horizontal subpixel iterators respectively, and n is the sample rate. This method is the simplest to implement, but can lead to artifacting due to the consistency of the subpixel sample positions, which may barely hit or miss edges to produce irregular results. This problem is solved by the second sampling method, random sampling.

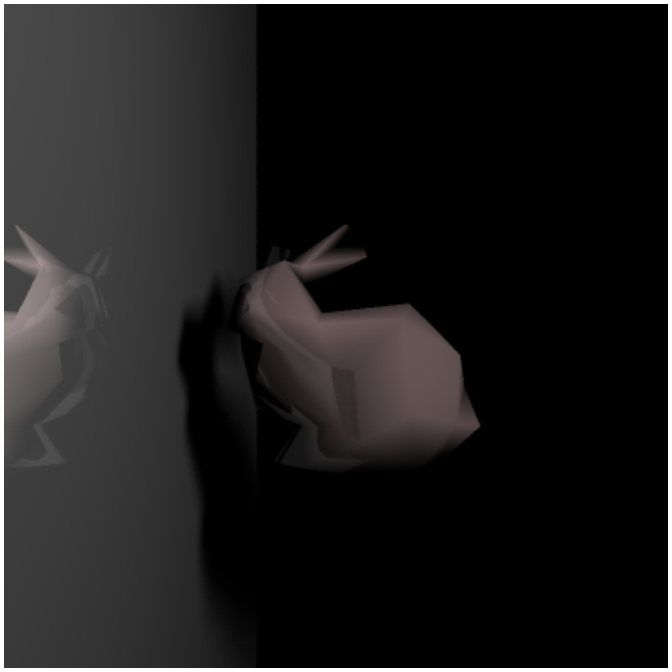
With random sampling, rather than sampling in a consistent grid within the pixel, a number of samples is taken at random positions within the pixel. A total of n^2 samples are taken at each pixel, with sample positions determined by the ordered pair $(i - \xi + 0.5, j - \xi + 0.5)$, where i and j are the pixel coordinates and ξ is a uniform random variable with a range of [0, 1]. With this method, the problem of near misses and near hits produced by the grid-like uniform sampling is eliminated, and the sample positions may be anywhere within the

pixel. However, this method can lead to objectionable levels of noise in the image due to random clustering of sample positions, particularly with low sample rates.



Renders of a refractive bunny looking at a mirrored surface. Both images use full-screen anti-aliasing with 2x supersampling; the left image uses uniform AA sampling, while the right image uses random AA sampling. Note the persistence of some rough edges with uniform sampling and the appearance of noise along edges with random sampling.

To decrease noise in the images while still avoiding the issues caused by grid-like sampling, features of uniform and random sampling can be combined to implement what is known as stratified sampling. With stratified sampling, rays are cast from random positions constrained by square subsections of the pixel. These positions are given by the ordered pair $(i + \frac{p - \frac{\xi+0.5}{2}}{n} - 0.5, j + \frac{q - \frac{\xi+0.5}{2}}{n} - 0.5)$, where all variables serve the same purpose as in above methods. By varying sample positions within subsections of the pixel rather than within the entire pixel, a reasonable compromise can be made between rigidity of sampling and output noise.



A render of the same scene as above, this time using stratified sampling for anti-aliasing. Note the edges on the bunny, which are smoother here than with uniform sampling and cleaner than with random sampling.

By using the edge detection system mentioned earlier, we can apply anti-aliasing to only the edges in an image. If the anti-aliasing mode is set to edge_detect, our implementation iterates over the edge mask, resampling the scene using the provided sample mode and rate at all positions marked as an edge in the edge matrix. This system is capable of substantially speeding up rendering, particularly in scenes with refractive objects.

```
julia> Rays.main(400, 400, "results/Anti_Aliasing_Results/", "full", "stratified
", 6, 1.0)
results/Anti_Aliasing_Results//wizard_cat/full_AA-stratified-N=6.png
3700.366148 seconds (75.85 G allocations: 4.393 TiB, 22.14% gc time)
```

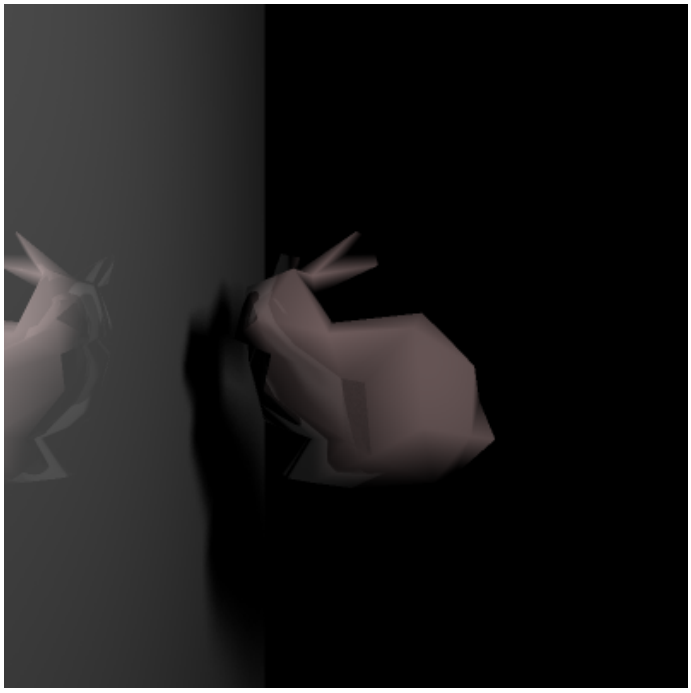
```
julia> Rays.main(400, 400, "results/Anti_Aliasing_Results/", "edge_detect", "str
atified", 6, 1.0)
results/Anti_Aliasing_Results//wizard_cat/edge_detect_AA-stratified-N=6-THICK=1.
0-shadows=true.png
1128.870670 seconds (25.89 G allocations: 1.494 TiB, 20.35% gc time)
```

Rendering a scene with a low-poly triangle mesh, a refractive Blinn-Phong sphere, and an area light takes around 3700 seconds with full-screen stratified anti-aliasing with a sample rate of 6, while rendering the same scene with edge-specific stratified anti-aliasing with an edge thickness of 1.0 and

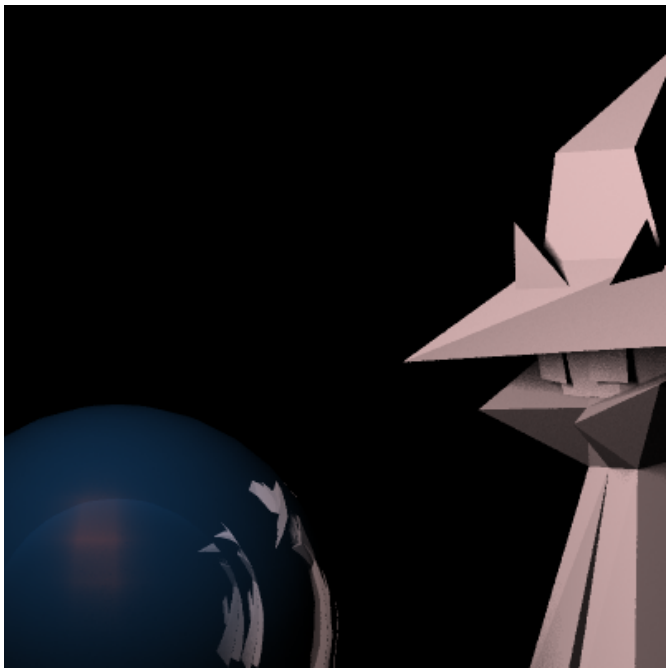
the same sample rate only takes around 1129 seconds. This illustrates the marked performance that comes with only filtering the edges within an image.

Area Lights

Area lights simulate lighting from sources with non-zero area. While in reality this process involves a surface receiving some amount of light from some fraction of the infinitely many points on the light's surface, this would be computationally impossible to simulate. Instead, we can sample rays from a finite number of random positions on the light, which provides convincing penumbra when iterated over many pixels. Building on knowledge from the anti-aliasing implementation, we use stratified sampling points randomly within different sections of the light. The area light is a parallelogram defined by a point and two side vectors, so subsections of the light are also parallelograms. A sample from the area light is a ray to the point $(x, y, z) + \frac{p+\xi}{n}\overrightarrow{a} + \frac{q+\xi}{n}\overrightarrow{b}$, where x, y, and z are the position coordinates of the light, $\overrightarrow{a}, \overrightarrow{b}$ are the side vectors of the light, p and q are iterator variables, n is the number of samples to take in each direction, and ξ is a uniform random variable with range [0, 1]. In our project, n is hard-coded to 5, as we found this value to produce decent penumbra with only moderate performance cost. The light is iterated upon by the shading function, with n samples taken in each dimension for a total of n^2 samples. Our area lights support both Lambertian and Blinn-Phong shading, and calculating the brightness of a given pixel is simply a matter of summing the contributions of each ray cast to the light. For each ray that is unobstructed, the result of the given shading equation is added to the total color with an intensity of $\frac{I}{n^2}$, where I is the intensity of the light. This process, when combined with the jittering within the stratified point sampling, produces highly convincing penumbra without significant noise or artifacts.



A scene using one area light and full-screen uniform anti-aliasing with sample rate 2. Note the soft shadow cast by the bunny.



Note the orange square specular highlight on the refractive sphere, which shows area lighting with Blinn-Phong shading.

Ray Tracer Run Commands

Rays.main(scene_name, camera_name, AA_type, sample_type, [AA_samples], [thickness], [detect_shadows]; [bvh_toggle=true/false])

Scene names

- “refract1”
- “bunny”
- “refract2”
- “wizard _tower”
- “wizard_hat”
- “wizard_cat”

Camera names: integer between 1 and 5

- 1 - canonical
- 2 - recommended for wizard_cat
- 4 - recommended for wizard_hat
- 5 - recommended for wizard_tower

AA types

- “none”
- “edge_detect”
- “full”

Sample types

- “uniform”
- “random”
- “stratified”

AA samples (optional): integer > 0 (default: 1)

Thickness (optional): real >= 0 (default: 1.0)

Detect shadows (optional): Boolean (default: true)

BHV toggle (optional): bvh_toggle=[Boolean] (default: false)