Ray Tracker

by Nate Morrical (u1137457), & Justin Jensen (u1140481)

Overview and Motivation

In a typical ray tracer implementation, a series of rays are tracked and combined for each pixel on the screen. Multiple effects, like anti-aliasing and depth of field can be created by altering how many and where these rays are generated within a pixel. As each of these ray bounce off objects, they emit a color based on a set of light and material properties. Each of these bounces can generate additional reflective and refractive secondary rays, which can in turn bounce off additional objects, collecting color as they go.

As rays travel through a scene, they must test for intersection with a variety of primitives, which can be expensive. As a result, optimizing the ray tracing process is a fairly heavily researched topic. A potential way to learn optimization insights is to study the intermediate data created during the ray traversal process. Unfortunately, the amount of data created by ray tracers is very large, and can be hard to manage and learn from.

However, if patterns can be spotted within this data, it may be possible to improve the performance of ray tracing. Rays intersecting the same geometry can be processed together for better memory coalescing. Rays that are found to have insignificant impact can be terminated early. Photon mapping has the potential to converge faster if light to camera ray paths are processed first. And scene graphs could be filtered for improved primitive intersection times.

Additionally, providing an intuitive and interactive visualization for the ray tracing process may help improve the understanding of ray tracing. As raster based graphics has improved, people may be wondering what the advantages are in moving to ray tracing. Some material effects, like refraction of light, can seem counterintuitive, but could be explained with this tool. As rays bounce, subtle changes in the image could be highlighted.

All of these factors motivated us to create "Ray Tracker", a tool which allows for interactive exploration of ray tracer generated data, for correlation analysis as well as a more general understanding of ray tracing.

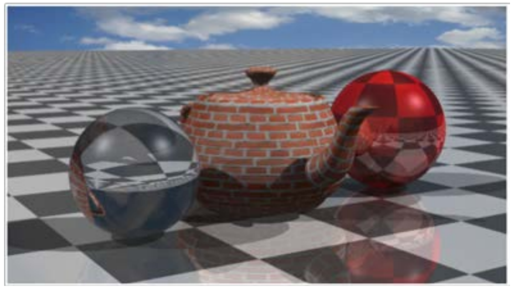


Fig 1. Anti-aliased Teapot

The above image is from Cem's 7th project, on adaptive antialiasing. When color variance is high within a pixel, aliasing can occur which makes an image look blocky. Taking more samples improves quality, but has a time tradeoff. For more information, visit the following page: http://www.cs.utah.edu/~natevm/courses/cs6620/prj7.html

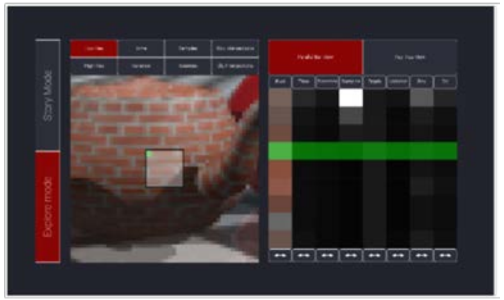


Fig 2. Ray Tracker Final UI

The final UI of the project, showing the same teapot scene, combined with the parallel bar chart displaying per pixel information on the right.

Questions being asked…

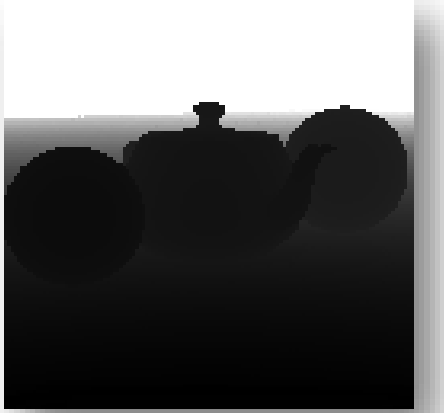
Below are some of the primary questions we tried to answer through our project:

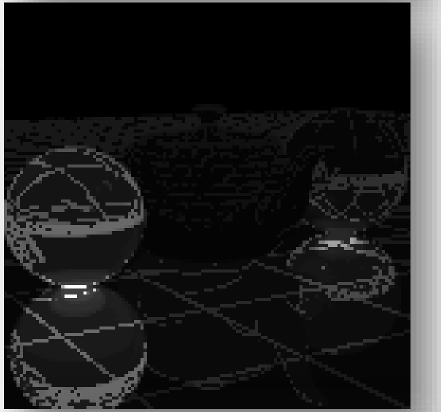
- What rays are intersecting what geometry?
- What rays are creating bottlenecks?
- Which rays are most important to the final image?
- Performance Cost vs Image Quality metrics?
- How does texture filtering affect an image?
- How does anti aliasing affect an image?

To answer these questions, we had to build a tool would allow for per pixel exploration through high quality interaction.

We hope that by using this tool, we can get a better understanding of the ray tracing process
We hope that this tool can also be used to teach the inner workings of a simple ray tracer to students just starting out in computer graphics.

Data





The primary data that we used for our project came from our ray tracer implementations from Cem Yuksel's Introduction to Ray Tracing class: http://graphics.cs.utah.edu/courses/cs6620/fall2017/

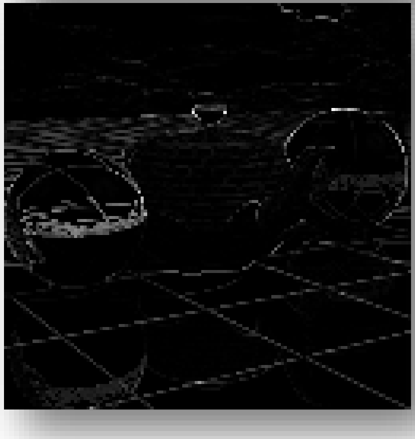
As scenes are generated, we export intermediate results for visualization in a json format. For each pixel, we generate a ray tree structure, which contains information like color contribution, total ray box intersections, total ray triangle intersections, as well as position and direction information.

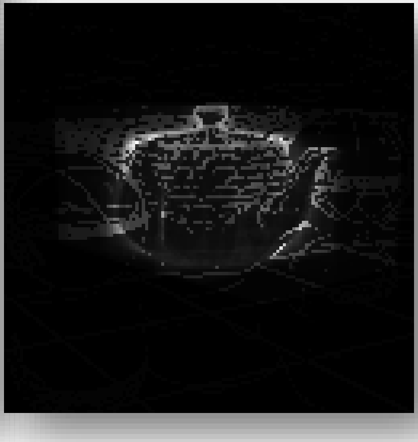
Rays additionally include a type, which can be either camera, shadow, reflection, or refraction.

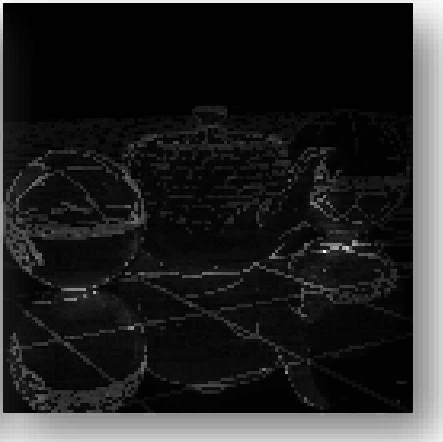
Data Processing

We generate all of the data with our ray tracer, so there isn't much substantial data cleanup involved. We can generate exactly what data we need and organize it how we need it.

Some data preprocessing was required to more effectively utilize d3's tree function. Since we were generating json, the data was inherently hierarchical, so we didn't need to use d3's stratify function.



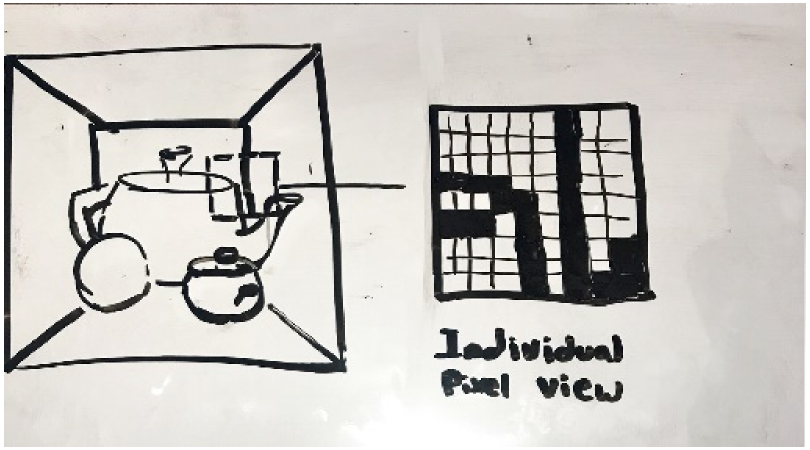


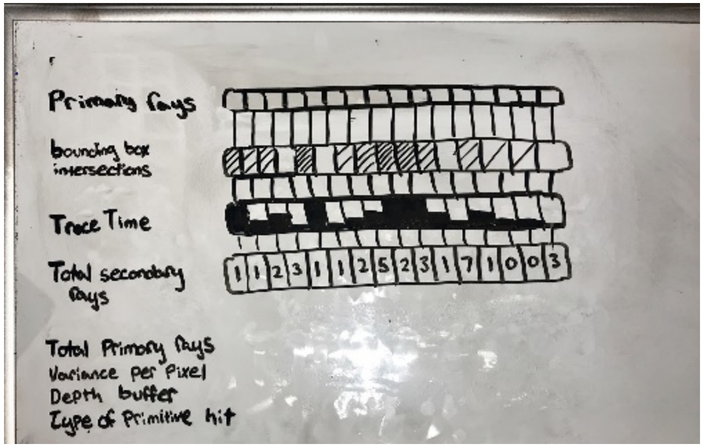


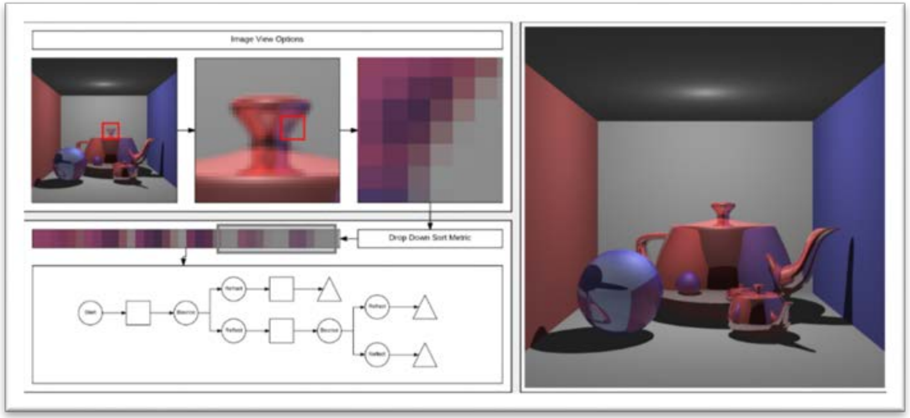
Exploratory Data Analysis

Through the combination of ray data and image data, we discovered a couple interesting things. We discovered that ray triangle intersections are most expensive on the edges of dense meshes, (left middle image), and we discovered that ray branching also correlates with object edges, due to high color variance. As a result, edges of object, especially refractive ones, tend to be more expensive to compute.

Early 2D Prototypes



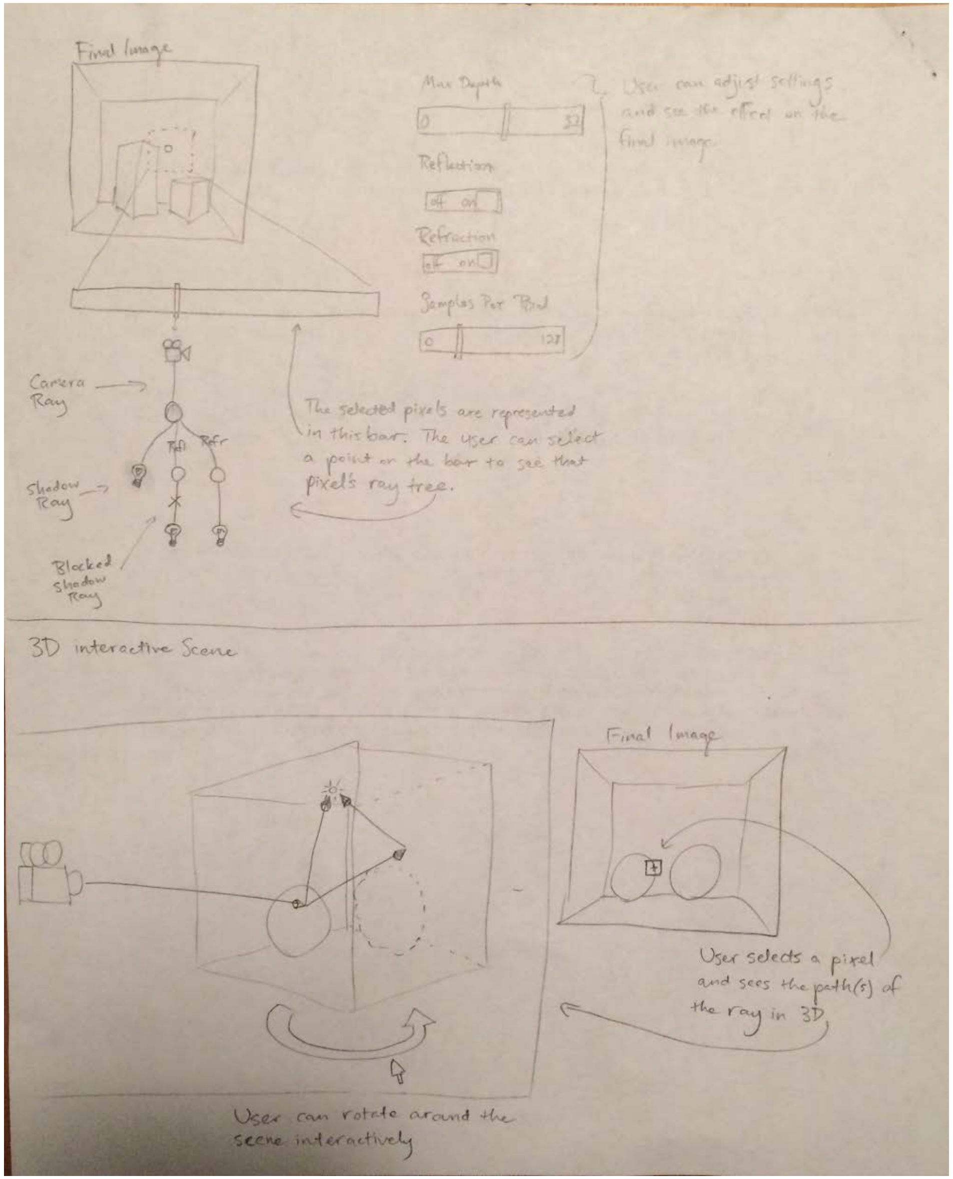




These are some of the first prototypes of raytracker. On the left, we were considering using a rectangular brush to select a region of interest on a given ray traced image. That region would be shown in another view, zooming in on a selection of pixels.

On the right, the individually selected pixels were to be shown as a set of equal length arrays. We were toying with different ways to visually encode different variables, debating whether to use value, position, or just display a number.

3D Prototypes


Final Image
Max Depth
0
32
User can adjust settings and see the effect on the final image
Reflection
off on
Refraction
off on
Samples Per Pixel
0
127
Camera Ray
Refl
Refr
Shadow Ray
Blocked Shadow Ray
The selected pixels are represented in this bar. The user can select a point on the bar to see that pixel's ray tree.
3D interactive Scene
Final Image
User selects a pixel and sees the path(s) of the ray in 3D
User can rotate around the scene interactively

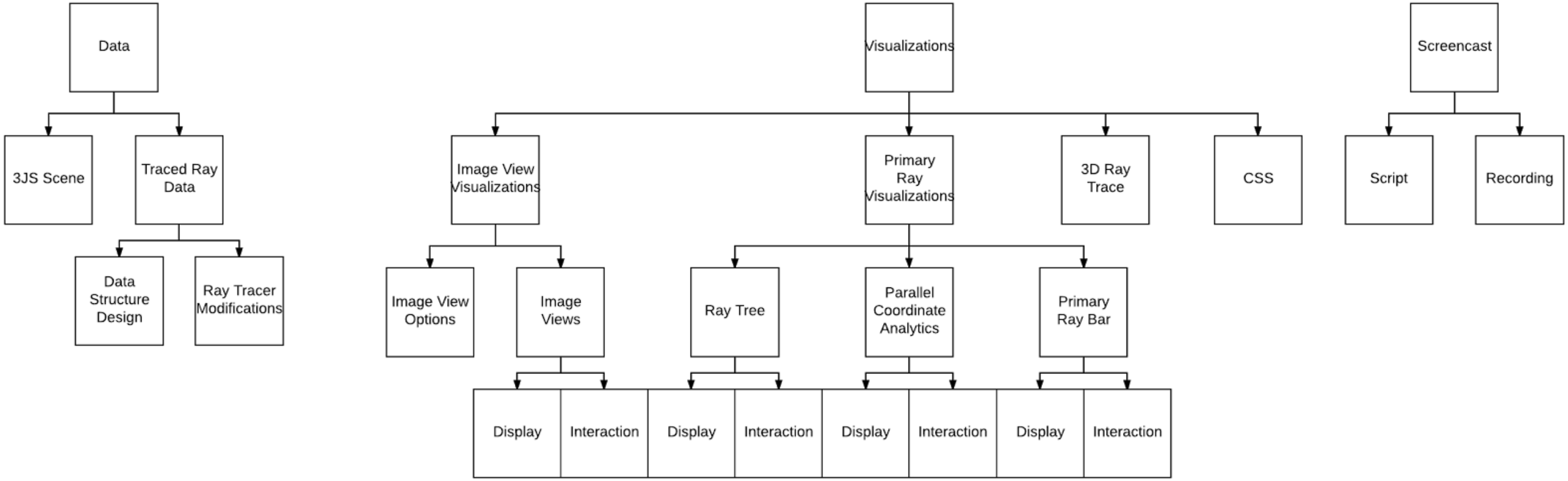
3D Viewport

At the beginning of the project, we intended on incorporating three.js into our project, since threejs is good at quickly spinning up a 3D environment. In that 3D environment, we'd show the rendered scene, but without doing any of the lighting calculations. In addition, we'd show a camera, and the rays generated for the selected pixel in the final image.

Due to time constraints, this evolved into the ray tree view, which is what ended up in the final project

Project Management

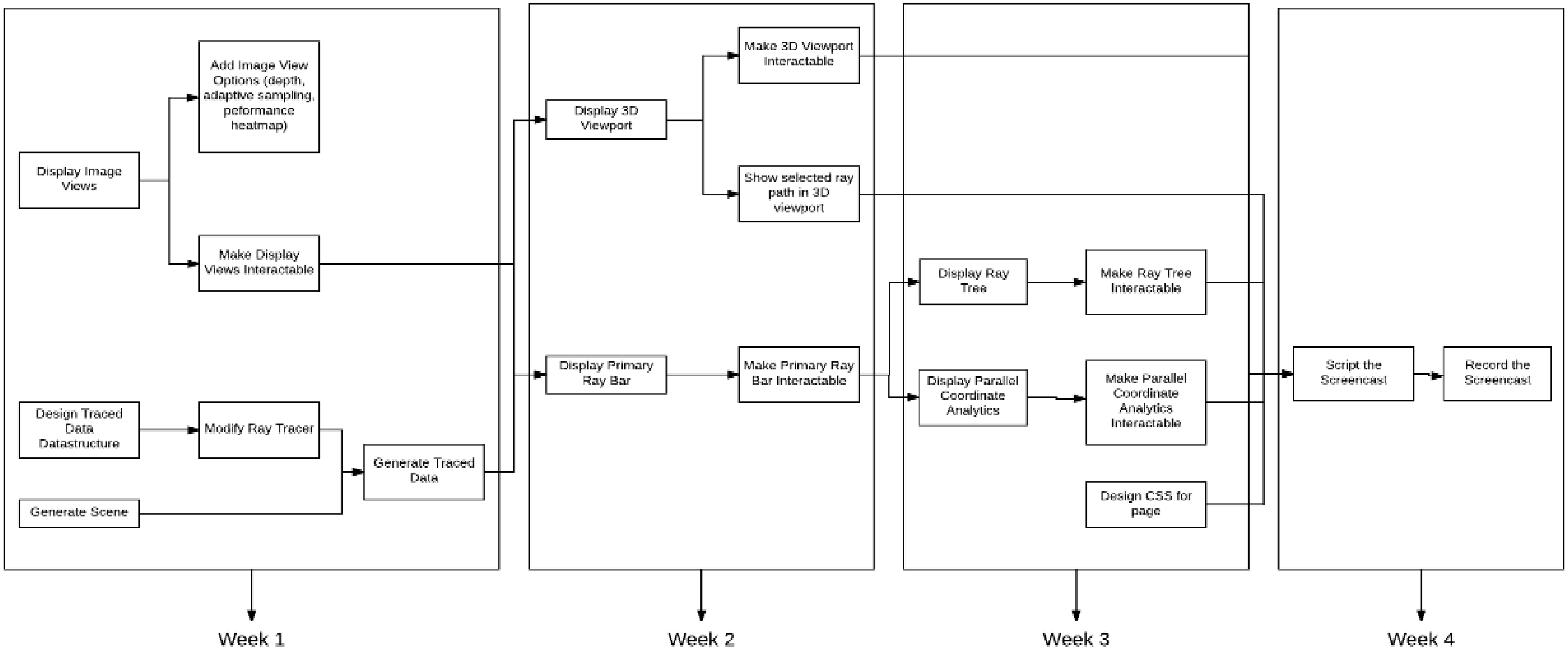
Work Breakdown Structure



We decided at the beginning of the project to incorporate some popular project management techniques to help us stay on track as the semester progresses. First, we broke down our data requirements, determining that we'd need to generate both the data as well as the data structure for the final visualization. Then we broke the visualization workload down into three components, the "Image View", the "Ray Visualization", the "3D Ray Trace", and finally CSS. This ended up changing to "Image View", "Parallel Bar View", and "Ray Tree View".

Project Management

Project Network Diagram/ Schedule



Display Image Views
Add Image View Options (depth, adaptive sampling, peformance heatmap)
Make Display Views Interactable
Design Traced Data Datastructure
Modify Ray Tracer
Generate Scene
Generate Traced Data
Display 3D Viewport
Make 3D Viewport Interactable
Show selected ray path in 3D viewport
Display Primary Ray Bar
Make Primary Ray Bar Interactable
Display Ray Tree
Make Ray Tree Interactable
Display Parallel Coordinate Analytics
Make Parallel Coordinate Analytics Interactable
Design CSS for page
Script the Screencast
Record the Screencast
Week 1
Week 2
Week 3
Week 4

These tasks were then graphed by dependency order, and then were scheduled on a week by week basis. Our initial schedule was pretty heavily scheduled towards the front, since we wanted to get a lot done. This ended up playing to our advantage later on, since we found we didn’t have enough time to implement everything in week 2 before week 3.

Data Design

Data Design

Ray data to collect:

For the scene:

- Camera info:
  - camera position
  - camera target
  - up vector
  - field of view
  - Image dimensions
- Scene info:
  - For each object
    - transformation
    - type (sphere, plane, mesh)
    - if mesh, filename
    - name
  - Scene coordinate system
    - What is up?
    - What is right?
    - What is forward?

For each pixel:

- Final pixel color
- Time taken to render
- (derived) Total number of secondary rays
- How many samples were taken
- Depth buffer
- Variance of the samples

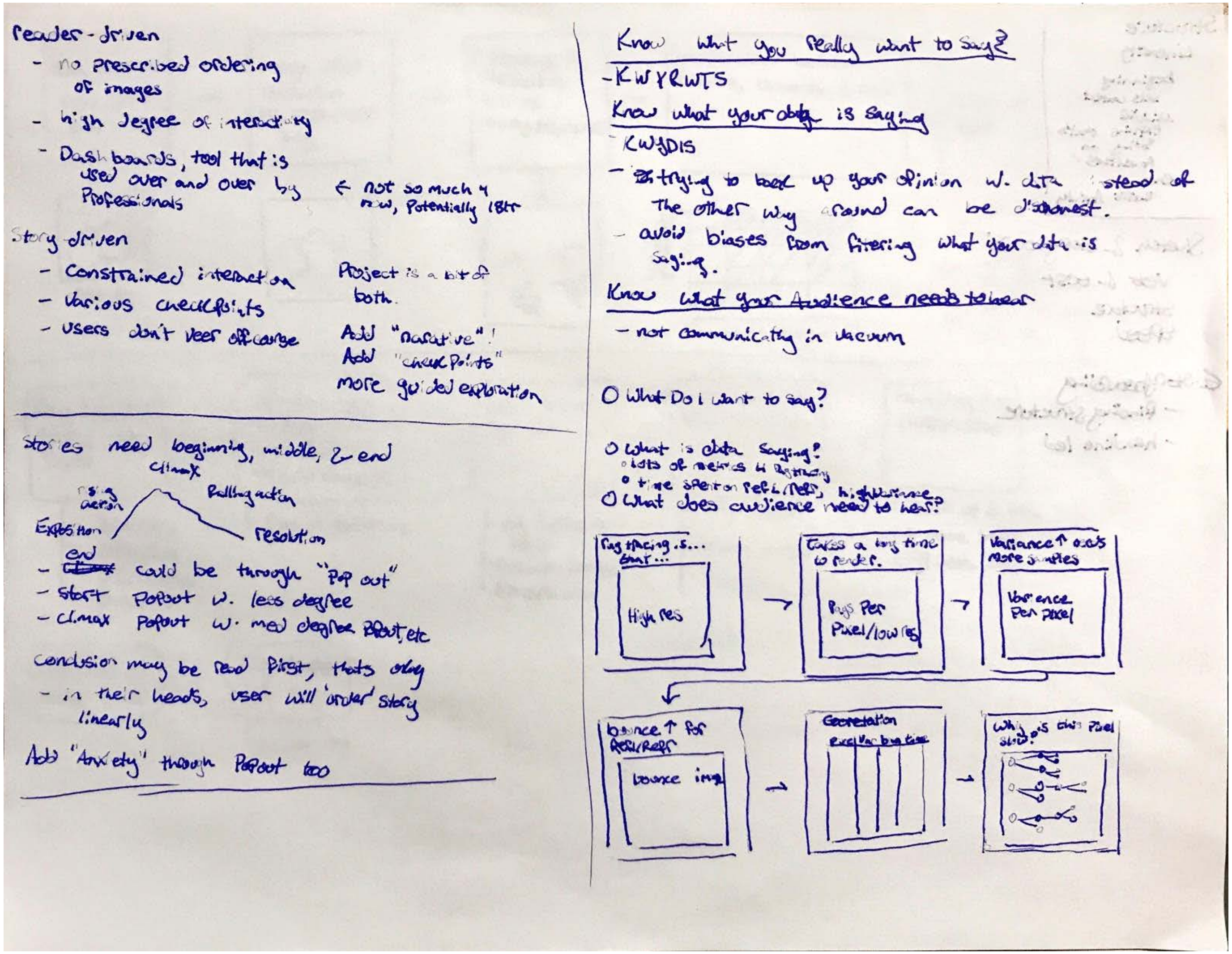
For each sample:

- Time taken to render
- (derived) Total number of secondary rays

For each ray (which includes the traversal through the scene):

- Origin point
- Direction
- Which object we hit
- Number of ray-box intersection tests
- Number of ray-primitive intersection tests
- Type (camera, reflection, refraction, shadow)
- Bounce color contribution
- A list of secondary rays created by this ray

Storytelling Study



reader-driven
- no prescribed ordering of images
- high degree of interactivity
- Dashboards, tool that is used over and over by Professionals

← not so much 4 now, Potentially 18tr

Story-driven
- Constrained interaction
- Various checkpoints
- users don't veer off course

Project is a bit of both.

Add "narrative"!
Add "checkPoints"
more guided exploration

Stories need beginning, middle, & end

- ~~[illegible]~~ end could be through "Pop out"
- start Popout w. less degree
- climax Popout w. med degree Popout, etc

conclusion may be read First, thats okay
- in their heads, user will 'order' story linearly

Add "Anxiety" through Popout too

Know what you really want to say?
-KWYRWTS

Know what your data is saying
-KWYDIS
- ~~[illegible]~~ trying to back up your opinion w. data instead of The other way around can be dishonest.
- avoid biases from filtering what your data is saying.

Know what your Audience needs to hear
- not communicating in vacuum

○ What Do I want to say?

○ What is data saying?
- lots of metrics 4 Raytracing
- time spent on refl/refr, high variance?

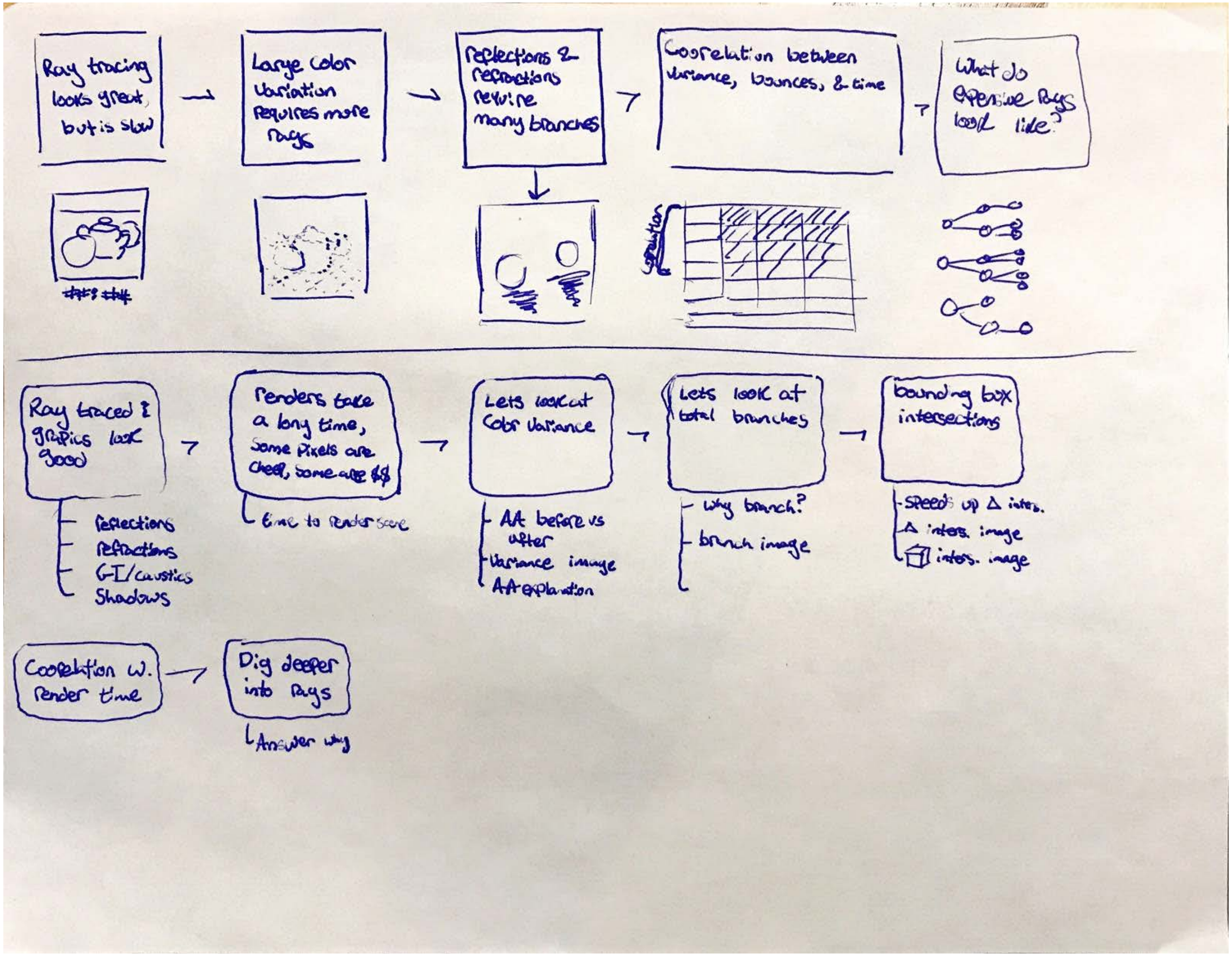
○ What does audience need to hear?

As we designed the interface, we tried to consider how we could strike a balance between creating a reader driven visualization, and a story driven visualization. We ended up creating both a story mode and an explore mode. Story mode is more story driven and explore mode is more reader driven.

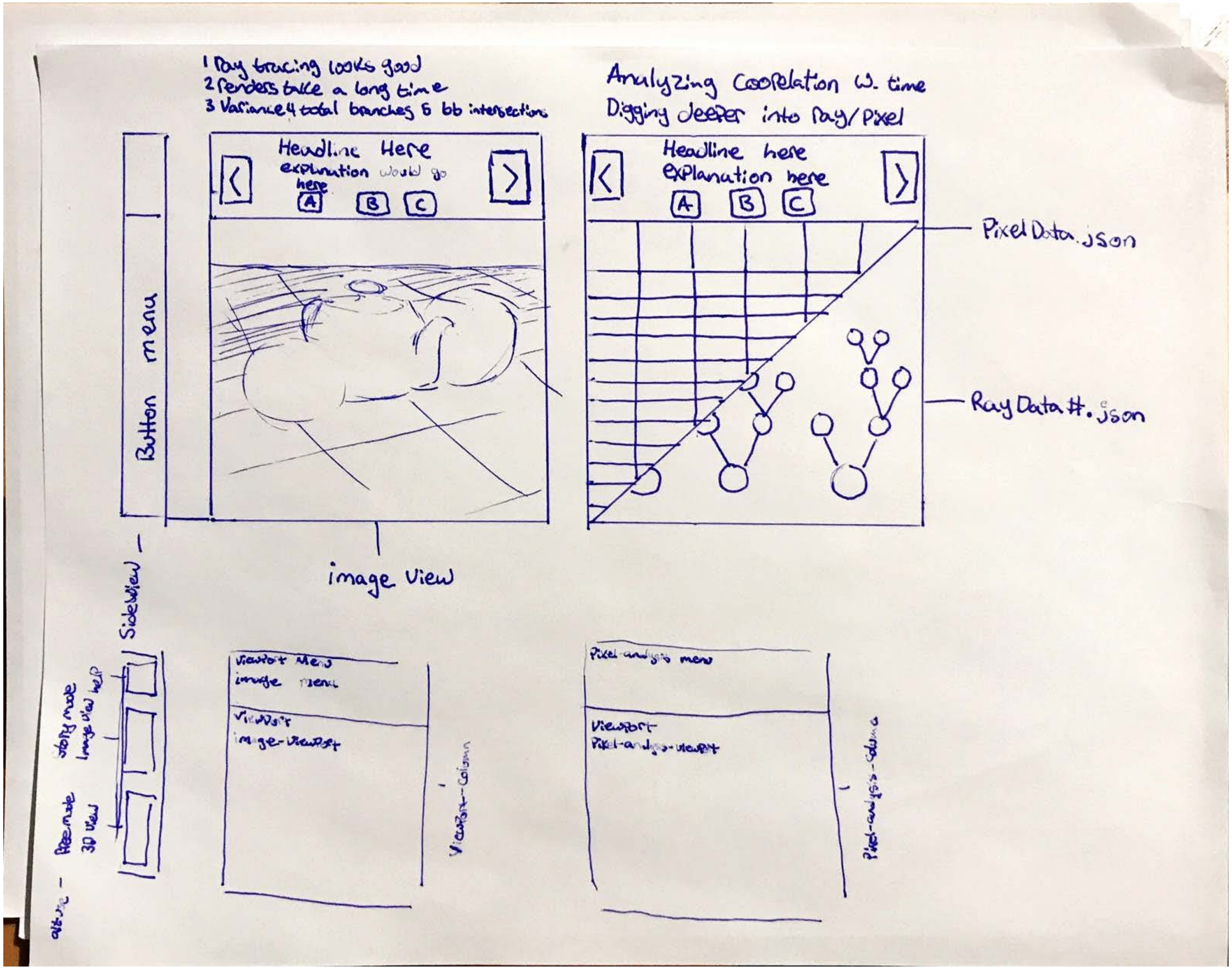
After meeting with Carolina, we decided to focus mainly on the elements of a basic ray tracer for the story. Interesting details can be teased out by sorting the parallel bars, sweeping over pixels, comparing images, and looking at the resulting ray trees.

Storyboarding Study

To incorporate more storytelling in the project, I tried storyboarding my project to clarify what I wanted to say, or what I wanted the user to find out. The top of this image is a first pass storyboard, where I was trying to determine how I'd visualize different concepts I was trying to get across. Most of these line up with the final product except for the ray tree, where I switched over to a radial tree to handle multiple trees cluttering the interface.


Ray tracing looks great, but is slow
Large color variation requires more rays
reflections & refractions require many branches
Coorelation between variance, bounces, & time
What do expensive Rays look like?
###
correlation
Ray traced & graphics look good
reflections
refractions
GI/caustics
Shadows
renders take a long time, some pixels are cheap, some are $$
time to render scene
Lets look at Color Variance
AA before vs after
Variance image
AA explanation
Lets look at total branches
why branch?
branch image
bounding box intersections
speeds up Δ inters.
Δ inters. image
inters. image
Coorelation w. render time
Dig deeper into Rays
Answer why

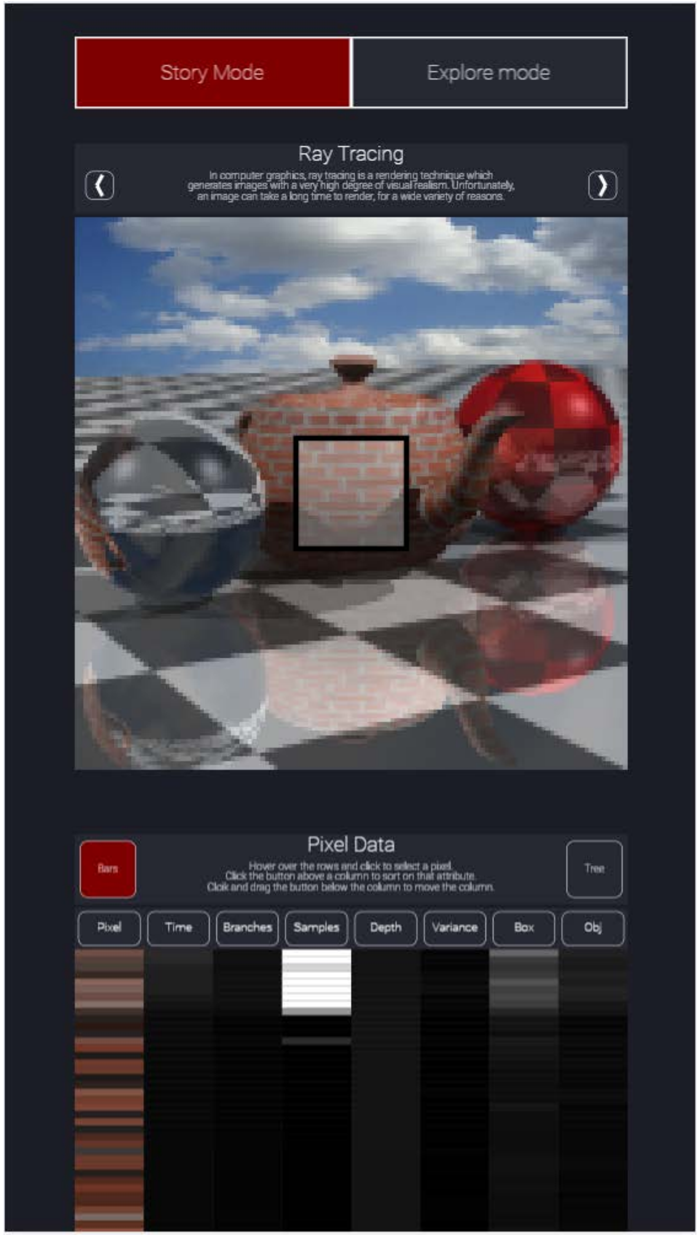
Design Evolution



I used this quite a bit to update flexbox styling for both mobile and desktop modes, and to try incorporating more of a storytelling element to my visualization.

I was also determining which visualizations would require what data.

Implementation: Portrait

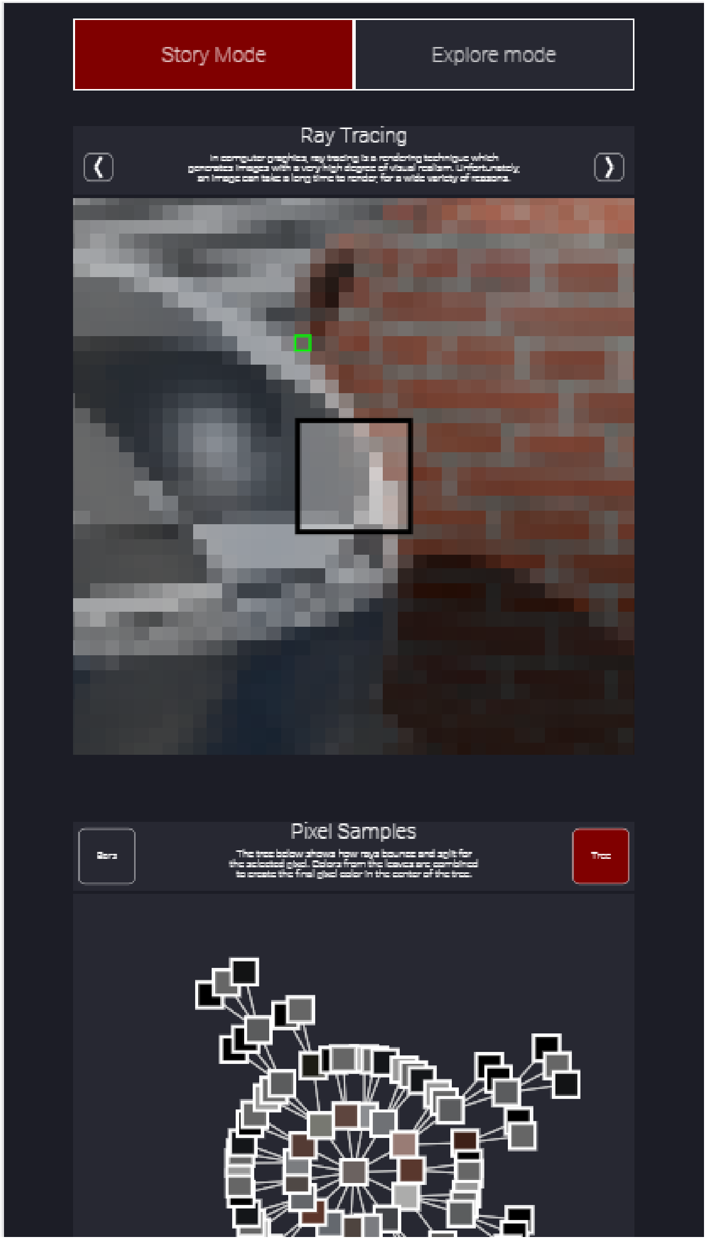


Ray Tracing

In computer graphics, ray tracing is a rendering technique which generates images with a very high degree of visual realism. Unfortunately, an image can take a long time to render, for a wide variety of reasons.

Pixel Data

Hover over the rows and click to select a pixel.
Click the button above a column to sort on that attribute.
Click and drag the button below the column to move the column.



Ray Tracing

In computer graphics, ray tracing is a rendering technique which generates images with a very high degree of visual realism. Unfortunately, an image can take a long time to render, for a wide variety of reasons.

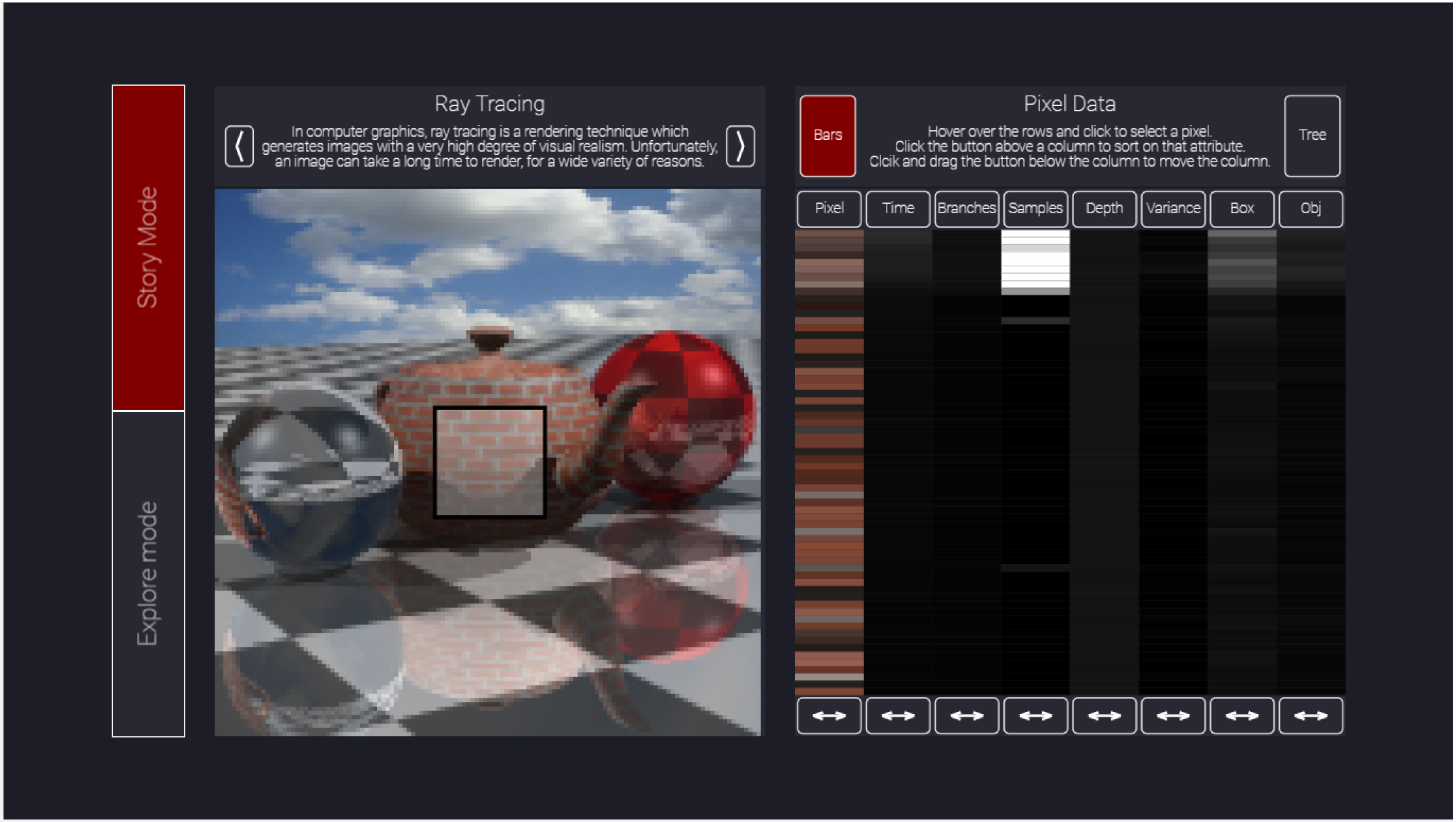
Pixel Samples

The tree below shows how rays bounce and split for the selected pixel. Colors from the leaves are combined to create the final pixel color in the center of the tree.

One of the goals was to make the project work on mobile. Originally, the navigable image spanned the entire width of the phone, but that made scrolling very difficult.

The visualizations all resize correctly when going from portrait to landscape and back.

Implementation: Landscape



Ray Tracing

In computer graphics, ray tracing is a rendering technique which generates images with a very high degree of visual realism. Unfortunately, an image can take a long time to render, for a wide variety of reasons.

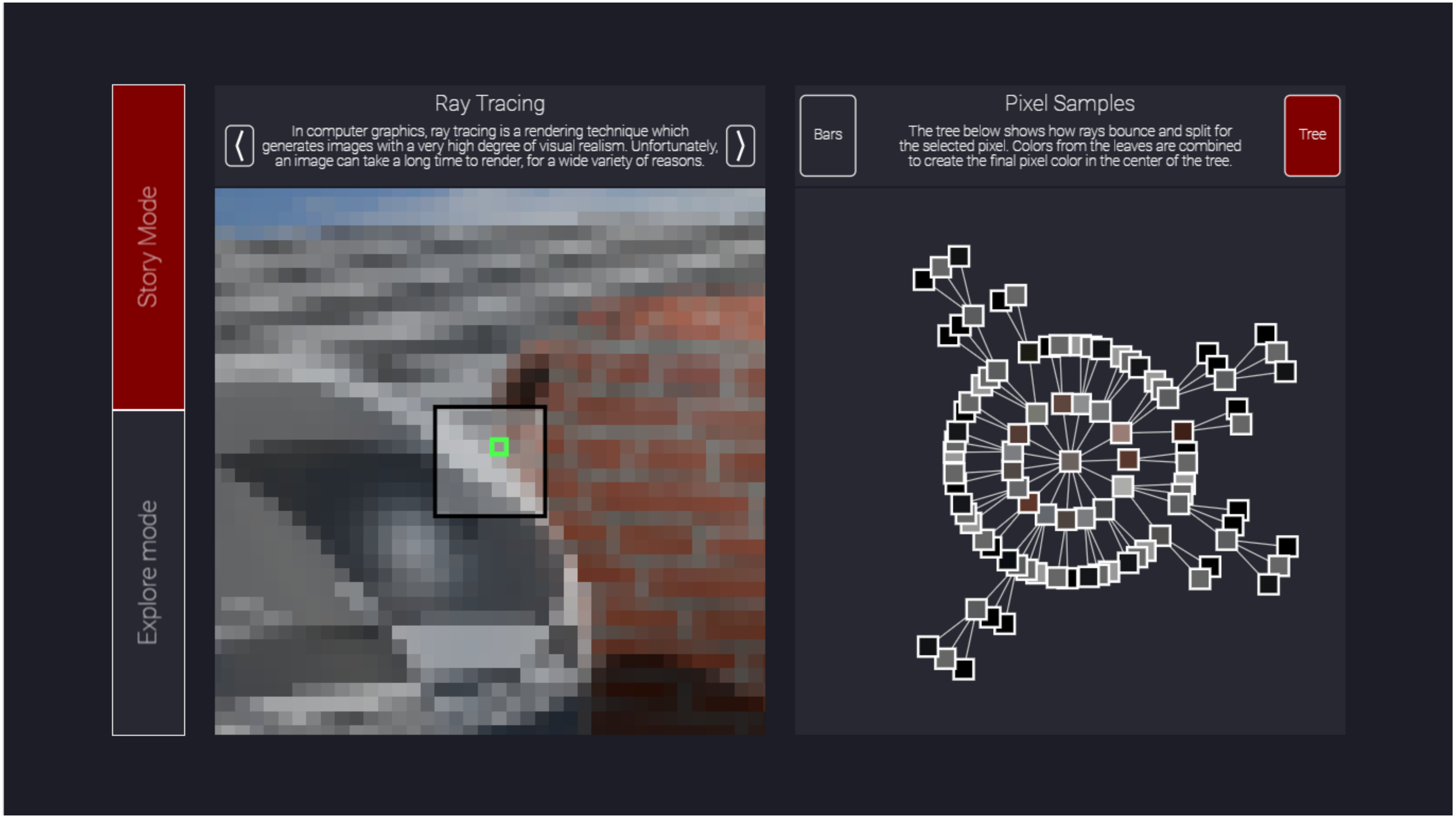
Pixel Data

Hover over the rows and click to select a pixel.
Click the button above a column to sort on that attribute.
Clcik and drag the button below the column to move the column.

In landscape, both the image view, and either the parallel bars or the ray tree can be seen.

Here, a set of pixels on the teapot surface show in the parallel bars through color, time, total branching, total samples, depth value, variance value, total box intersections, and total object intersections

Implementation: Landscape



Ray Tracing

In computer graphics, ray tracing is a rendering technique which generates images with a very high degree of visual realism. Unfortunately, an image can take a long time to render, for a wide variety of reasons.

Pixel Samples

The tree below shows how rays bounce and split for the selected pixel. Colors from the leaves are combined to create the final pixel color in the center of the tree.

Here, a pixel with high branching has been selected (shown in green). This branching is displayed using a radial tree. A radial tree was used over a standard tree to better optimize space utilization.

Evaluation

Overall I’m fairly satisfied with this project.

I feel like I learned a lot about basic ray tracing data, and how I might go about improving the performance of my personal ray tracer. I also have a deeper appreciation for global illumination, which involves many more ray trees, and can relate those concepts back to the visualizations I created in this project.

I think the visualization I created works well, although some interaction approaches can seem counter-productive at first. A 3D view is still missing, and I’m also missing a couple crucial details in terms of global illumination and photon mapping which would be needed before this tool can benefit advanced ray tracing studies.

As far as a teaching tool is concerned, I think this tool could help teachers and professors out quite a bit when explaining some intermediate ray tracing concepts, especially adaptive sampling.