Special Relativistic Ray-Tracing Techniques

Ryan ElKochta
relkocht@umd.edu

Tien Vu
vut@umd.edu

1 Introduction

Ray tracing is a technique used in rendering images of 3-dimensional scenes where light is modeled as *rays* with infinitesimal thickness. For instance, as part of this course, we have implemented a backwards path tracer. In the backwards path tracer, light rays originate at the camera and are traced into the scene. Surface bidirectional reflectance distribution functions (BRDFs) are sampled at each hitpoint to find new ray directions, and rays are continuously bounced off of objects until they hit a light source (or, until a light source is sampled using emitter sampling techniques). The color that is seen at the origin pixel on the image plane is the accumulation of light that bounces in its direction.

Notably, in this simulation, we are tracing events in the *opposite* direction that they would occur in reality. In reality, light rays originate at emitters and "bounce" off of objects until they reach the camera. However, to minimize variance, the computer simulation typically works *backwards in time,* starting at the camera instead.

For classical ray tracing, this does not affect the physical calculations much, as all objects in the scene are considered stationary, and as a result the only time-dependent calculation that must be made is when performing ray-intersections (as we always take the "first" intersection of each ray). If all objects are stationary, then the speed of light is not taken into account, and we get the same result as if light rays traveled infinitely fast.

Our final project's goal is to implement ray-tracing techniques that do not assume stationary objects in the scene and allow for a finite speed of light. We do this by replacing the Newtonian physics in classical ray tracing to take into account Einstein's theory of special relativity [6]. We compare our results to the classical ray tracer without relativistic corrections, and find that moving objects at speeds approaching light speed has a significant effect on the produced image.

Our project was quite successful. We have implemented our work on top of the Aris renderer [5]. Our work considers the effects of the following generalizations of classical ray-tracing techniques: (1) fixing the speed of light at a constant c, rather than infinite; (2) allowing objects to move in the scene at variable velocities relative to each other, potentially approaching the speed of light c; and (3) accounting for the Doppler shift on each ray bounce. We emphasize that our work does not move the entire scene at a single velocity relative to the camera; instead, *every object is allowed to have its own velocity.*

In Aris, we developed a new geometry type along with modifications to the existing `albedo` and `path` integrators to account for these effects. We also developed a handful of demonstration scenes and rendered them using the new techniques. Finally, we developed a framework to animate scenes over time, and animated our demonstration scenes to show special relativistic effects over time.

This paper is organized as follows: in section 2, we provide a brief explanation of special relativity and the Lorentz transformation. In section 3, we summarize previous work that has been done on special relativistic ray-tracing and how it relates to our work. In section 4, we explain the changes we made to the Aris renderer to incorporate relativity. In section 5, we provide graphics that demonstrate our work along with commentary. Finally, in sections 7 and 8, we conclude with a summary of our findings and details on the division of work between the two of us.

2 Background

2.1 Special Relativity

Einstein's theory of special relativity follows from two *postulates* [3][6]:

- The vacuum speed of light c is a constant in all inertial (non-accelerating) frames of reference, independent of the velocity of any involved objects.
- The laws of physics do not depend on which reference frame you are in, as long as it is non-accelerating.

These two postulates lead to some interesting implications; most notably is that space and time now become *intertwined* and dependent on the reference frame of the observer.

Say we have two reference frames S and S', with S considered stationary and S' moving at speed $\hat{V}$ relative to S. Without loss of generality, we orient our reference frames such that the origin of S and S' coincide at time $t = t' = 0$, i.e. the *measured time* in each reference frame (and note that these are different!)

Consider a coordinate in S in both space and time $[x, y, z, t]$. Such a spacetime coordinate is typically called

an *event.* We can convert from an event in S to an event $[x', y', z', t']$ in S' using the Lorentz transformation:

$$\vec{X'} = \vec{X} + \left(\frac{\gamma - 1}{||\vec{V}||^2} (\vec{X} \cdot \vec{V}) - \gamma t \right) \vec{V} \tag{2.1}$$

$$t' = \gamma (t - \frac{\vec{X} \cdot \vec{V}}{c^2}) \tag{2.2}$$

where $\vec{X} = (x, y, z)$, $\vec{X'} = (x', y', z')$, and c is the speed of light.

3 Literature Review

This project was inspired by Daniel, Dolph, and Elien's work in [2]. The authors added support for certain special relativistic effects to the POV-Ray ray tracer. They assumed that all objects in the scene are moving at the same velocity, and limited themselves to a stationary camera. Within these assumptions, they explained a method to perform ray intersections accounting for both the motion of the object and the finite speed of light. They achieved this by adding the (normalized to a ratio of c) velocity of the object to the ray and computing the intersections as normal. The hitpoints will be in the scene frame S', rather than the stationary camera frame S; they explain that they can be transformed back into the S frame to find the "correct" intersection point. We note that in special relativity, there is no "correct" reference frame, and this backwards transformation is not necessary for our technique.

The authors then give the *relativistic aberration equation* [2][3] which is used to transform ray *directions* $\vec{d}$ from the camera frame S into the scene frame S':

$$\vec{d'} = \frac{\vec{d} + \left(\frac{\gamma - 1}{||\vec{V}||^2} (\vec{d} \cdot \vec{V}) + \gamma \right) \vec{V}}{\gamma (1 + \vec{d} \cdot V)} \tag{3.1}$$

The combination of the Lorentz transformation and the above equation allowed the authors to implement a relativistic albedo integrator.

Daniel, Dolph, and Elien also explain how to compute the Doppler shift, wherein the wavelength of light is shifted depending on the relative velocity of the observer. The relativistic Doppler shift is given by [2][3]:

$$\lambda = \lambda_0 \gamma (1 + \vec{V} \cdot \vec{a}) \tag{3.2}$$

where λ is the wavelength of the shifted rays λ_0 is the wavelength initially emitted from the object, and $\vec{a}$ is the ray direction *in the observer frame.*

The problem is that most ray tracers (including Aris) do not represent ray colors as wavelengths, but instead as (R, G, B) tuples. The authors explain that the RGB colors can be converted to HSV to hue to wavelength, but do not give much detail.

Our work expands upon [2] by removing the restriction that objects in the scene move at the same speed, allowing the camera to have a velocity, and implementing the full path tracer (i.e. transforming rays and hitpoints recursively, not just once). We also give more detail on how to approximate the Doppler shift with RGB values.

Hsiung and Thibadeau [3] explain how to remove the restriction that all objects in the scene move at the same velocity using their *multiple frame intersection* technique. The idea is to split the scene into multiple reference frames S_i, each moving at some velocity $\vec{V_i}$ with respect to the stationary frame S. Each ray origin and ray direction pair is transformed using the above techniques into *each* reference frame in the scene, the ray intersection test is performed for each frame, and the intersection *times* are calculated; these times are sorted to find the earliest intersection point.

The authors did implement this technique and give results, however as far as we can tell they did not implement it recursively (i.e. implement a full path tracer). We could not find their source code. We give more detail on how to implement this in a recursive path tracer.

We could not locate any sources that explained how to perform relativistic emitter hits or emitter sampling in ray tracing; this project will explain how to do these.

The Lorentz transformation implies that an object moving at relativistic speeds will be measured as *contracted* in the direction of motion. Terrell [8] shows that this effect will *appear* to cancel out with the effects just from a finite speed of light. So we should not expect to observe Lorentz contraction.

4 Implementation

Our implementation consists of the following modifications to the Aris renderer: (1) the addition of the initial time t_0 as a scene parameter, along with an animation framework that uses this; (2) the implementation of a `RelativisticGeometry`, which wraps the `MeshGeometry` with support for objects moving at different velocities; (3) the addition of a `RelativisticAlbedoIntegrator`, a modification of the existing albedo integrator to perform intersections using the method from [3]; (4) the addition of a `RelativisticIntegrator`, a modification of the existing *path* integrator that uses the above methods; and (5) consideration of the Doppler effect in both integrators.

We started from the latest commit Aris renderer available on GitHub, and added the instructor-provided reference implementations of area emitters, sampling functions, dielectric BRDF, and the backwards path tracer.

4.1 Animations and t_0

We added a scene (global) parameter t_0 which indicates the *start time* of the rendering. Recall that since our raytracing technique traces rays backwards in time, the produced image will show what the camera sees at time t_0 in its reference frame; in other words, the rendered image depends on the rays that *reach the camera* at time t_0.

We also developed an animation script, which runs Aris multiple times to render multiple frames of a scene as time moves forward. We used this script to produce a number of special relativistic animations; we discuss these results in section 5.

4.2 Relativistic Scene Geometry

We have added a new geometry implementation to Aris: the `RelativisticGeometry`. This class stores a list of `ReferenceFrame` objects, which are defined as follows:

```
@dataclass
class ReferenceFrame:
    geometry: MeshGeometry
    velocity: Tensor
    brdf_i_offset: int
```

Each reference frame corresponds to a grouping of objects in the scene moving at the same velocity. For example, a scene with a rocket ship moving at $0.7c$ and a star moving at $0.3c$ would have two reference frames: one for the rocket ship, with a `MeshGeometry` containing all of the triangles corresponding to the rocket ship, and another for the star, containing a `MeshGeometry` containing all of the triangles for the star.

In the configuration files for the geometry, the list of paths to Blender `.obj` files has been replaced with a list of list of paths, with each sublist corresponding to a different reference frame. A new geometry option, `betas`, has been added, allowing to specify the velocity of each reference frame.

One issue that arose from storing multiple mesh geometries in the relativistic geometry is handling of BRDF indices. In Aris, BRDFs are specified as a single list specifying the BRDF for each `.obj` file in the mesh geometry. The mesh geometry `ray_intersect()` returns `geo_out.brdf_i`, which specifies the index for the object that each ray hit. The issue is that when we call the mesh `ray_intersect()`, we get `brdf_i` indices that correspond to the objects *in that frame,* not in the entire overall relativistic geometry.

Our solution to this problem was, when outputting `brdf_i` in our `ray_intersect()` implementation, to add a separate offset corresponding to which reference frame the intersection was in. When calling the sampling functions that use the primitive index, we first convert the primitive index into a reference frame number and an index within that reference frame, and wrap around that reference frame's mesh geometry.

4.3 Relativistic Geometry Ray Intersections

Much of the programming for this project was done in the `ray_intersect()` implementation for `RelativisticGeometry`. We fill in the details for, and implement, the multiple frame intersection technique from [3]. The idea is that for *every* reference frame in the scene, we transform the ray origins and directions into that frame and fire the transformed rays into the frame's mesh geometry using the standard `ray_intersect()` function.

Because special relativity intertwines space and time, transforming points and directions from one reference frame to another requires knowing not only the points and directions in the old frame but also knowing the current *time* in the old frame. To this end, we have modified the relativistic `ray_intersect()` signature as follows:

```
def ray_intersect(self,
    rays_o: Tensor,
    rays_d: Tensor,
    velocity: Tensor,
    time: Tensor)
    -> (GeometryOutput, Tensor, Tensor)
```

The additional inputs are the velocity with respect to "stationary" (i.e. the same reference frame as the camera movement) and the time in the reference frame of the ray origins. For example, for the first set of rays that are fired into the scene, we pass the camera velocity and t_0. The additional outputs are the velocity of the hitpoints (in the same frame as the inputs) and the *time* of the hit events (in the reference frame of the hitpoints). This definition ended up being a very good abstraction, as it allowed us to account for relativity in the albedo and path integrators relatively easily.

Our ray intersection works as follows: we loop over every single reference frame in the scene. The first thing we need is a way to find the *relative* velocity between the ray origin points and each reference frame. For this, we use the *relativistic velocity addition* equation [1] (note that we are normalizing v to a factor of c, so $c = 1$:

$$\vec{u} = \frac{\vec{u}' + \vec{v}}{1 + \vec{u}' \cdot \vec{v}} + \frac{\gamma}{(1+\gamma)} \frac{\vec{v} \times (\vec{v} \times \vec{u}')}{1 + \vec{v} \cdot \vec{u}'} \tag{4.1}$$

where $\vec{v}$ is the velocity of a moving frame relative to the observer, and $\vec{u}'$ is the velocity of an object within that moving frame, and γ is computed in terms of $\vec{v}$.

We apply the above equation with $\vec{v}$ as the negative of the origin points' velocity (to get the velocity of the aforementioned "stationary" frame in the origin points' frame), and $\vec{u}'$ as the velocity of the current frame we're looping over in that "stationary" frame. This gives us $\vec{u}$, the desired relative velocity between the origins and and the current frame.

Next, we use equations (2.1) and (2.2) to transform `rays_o` `rays_d`, and the ray origin event times into the current reference frame in the loop.

We ran into the following issue: equation (2.1) contains a division by the relative velocity between the objects, but it is possible that the objects' relative velocity is zero, leading to division by zero. Our simple solution to this was to have a small velocity threshold (which we set

to $0.00000001c$), below which the rays are not transformed at all.

Next, we fire the transformed rays into the current reference frame, and save the output along with the transformed rays for this reference frame. Finally, we compute the ray travel time in the hitpoints' frame of reference as `(geo_out.points - rays_o_prime).norm(dim=-1)` and use this to calculate the intersection time of each ray *as observed in the ray origins' frame.* We save these times as `sortable_times` and continue for every reference frame in the scene.

After the loop fires rays into every reference frame, we use `torch.max()` to calculate the indices of the reference frame where the latest intersection in time was seen from the origin frame. We then construct and return a `GeometryOutput`, where at each index we use the output from the ray intersection in the reference frame whose `MeshGeometry` had the most recent intersection. This follows from the work of Hsiung and Thibadeau [3]. We also correct the `brdf_i` as described earlier.

4.4 Relativistic Albedo Integrator

Our `RelativisticAlbedoIntegrator` is the same as the Aris built-in albedo integrator, except that we pass both the camera velocity and the scene's t_0 to the `RelativisticGeometry`'s `ray_intersect()` call.

4.5 Relativistic Path Tracer

Our relativistic path tracer is based on the instructor-provided path tracer. Like in the relativistic albedo integrator, we pass the camera velocity and the scene's t_0 value to the initial call to the relativistic `ray_intersect`. When `ray_intersect()` returns the velocity and time values t at each hitpoint, we simply pass these values to the next call to `ray_intersect()` in the loop.

4.5.1 Emitter Hit

Surprisingly, the emitter hit logic did not require any modifications. The `emitter_hit()` function takes the points and normals passed to it, along with the origin points `rays_o` and directions `rays_d` to calculate the amount of light added by the emitter hit. However, due to the way our relativistic ray intersect logic works, both the `rays_o` and the `rays_d` have already been transformed into the same reference frame as the hitpoints. Thus, the calculations all happen in the reference frame of the hitpoints, so the emitter hit calculation still works.

4.5.2 Emitter Sampling

Emitter sampling, however, does require some modification. In the existing implementation, the issue is that the points we sample on the emitter are in the reference frame of the emitter, whereas the hitpoints we are connecting with the emitter are in their own reference frame.

As a result, we developed the following modifications to handle relativistic emitter sampling. Denote the frame of the emitter as E and the frame of the hitpoints as H. We begin by calculating the velocity v_{eh} of the E frame with respect to the H frame using equation (4.1). The velocity of the H frame with respect to the E frame is $v_{he} = -v_{eh}$.

We sample points on the emitter in frame E as usual (using `emitter.sample()`). Next, we convert the hitpoints (in whose frame we also have time t_h) into the E frame using equations (2.1) and (2.2). We then use the distance in the E frame to calculate the ray travel time in the E frame, and from this the time we hit each emitter sampled point (again in E). We then convert the sampled points on the emitter along with their times t_e back into the H frame. Finally, we calculate the visibility directions `d_target_point` in the H frame as the direction from the hitpoints to the emitter-sampled points.

After all this, we call `emitter.le()` to calculate the L_e term. This function did require one modification: we pass it the velocity and H-frame times of the hitpoints, as this is required to perform the visibility check using the relativistic `ray_intersect()`. The distances for the final emitter sampling calculation are given in the H frame.

4.6 Doppler Effect

The Doppler effect is the change in the frequency of a wave to an observer when the source of the wave is moving. As the Doppler effect applies to waves and not just noise, the same effect on the frequency of the wavelength of light can be calculated and rendered, albeit with a few intermediate steps to convert the RGB of the pixels to HSV, convert the hues of HSV to wavelength, compute the Doppler shift accounting for the Lorentz factor γ, and finally convert back from wavelength to HSV and HSV to RGB. And of course, all of these conversions utilize PyTorch's Tensor operations such that they are performed for many pixels at once.

One point of interest is how the Doppler effect is applied to "even" colors, such as pure white with an RGB value of $(255, 255, 255)$ or a gray like $(100, 100, 100)$. Naively, one might attempt to convert these RGB values directly to HSV and then to wavelength. However, these "even" colors are not able to be presented by a single wavelength as the visible light spectrum does not include a value that directly maps to white. In reality, white and "even" colors are represented by an even amount of every other wavelength of light. Thus, a Doppler shift on these "even" colors directly from their RGB values will not induce any shift in the resulting wavelength. To account for this, every pixel's RGB values must be split into separate red, green, and blue color channels before conversion to HSV and then to wavelength. While this may not be the most physically accurate method, it accomplishes the task in a reasonable manner by breaking these "even" colors into three separate, wavelength-representable colors.

4.6.1 RGB to HSV

As per Wikipedia, conversion from RGB to HSV is a relatively simple set of steps which are as follows [4]:
Obtaining hue:

$$M = \max(R, G, B) \tag{4.2}$$

$$m = \min(R, G, B) \tag{4.3}$$

$$C = \text{range}(R, G, B) = M - m \tag{4.4}$$

$$H = 60^\circ \times \begin{cases} 0, & C = 0 \\ \frac{G-B}{C} \mod 6, & M = R \\ \frac{B-R}{C} + 2, & M = G \\ \frac{R-G}{C} + 4, & M = B \end{cases} \tag{4.5}$$

Obtaining value:

$$V = M \times 100 \tag{4.6}$$

Obtaining saturation:

$$S = \begin{cases} 0, & V = 0 \\ \frac{C}{V} \times 100, & \text{otherwise} \end{cases} \tag{4.7}$$

4.6.2 Hue to Wavelength

Conversion from the hue in HSV to wavelength can be accomplished in a variety of ways. For the sake of simplicity, a table of hue to wavelength mappings from Wikipedia's Spectral Color page [7] was loaded into Aris and then interpolated using numpy.

4.6.3 Doppler Shift

The Doppler shift itself is a simply application of formula 3.2 to modify the original wavelength depending on the velocity of the viewed object relative to the camera.

The strength of this effect is made configurable via the variable s such that:

$$\Delta\lambda = \lambda - \lambda_0 \tag{4.8}$$

$$\lambda_s = \Delta\lambda s + \lambda_0 \tag{4.9}$$

4.6.4 Wavelength to Hue

The conversion from wavelength to HSV was performed in the same manner as the inverse above, with a slight modification to account for the fact that numpy's `interpolate()` function only accepts monotonically increasing x values. Since wavelength only maps to hue, the saturation and value from the original HSV are combined with the newly Doppler-shifted hue.

4.6.5 HSV to RGB

Unfortunately, the nature of the RGB to HSV conversion does not lend itself to cleanly inverting the process as was done with wavelengths and HSV. Once again as per Wikipedia, the conversion from RGB to HSV is as follows [4]:

$$S' = \frac{S}{100} \tag{4.10}$$

$$V' = \frac{V}{100} \tag{4.11}$$

$$H' = \frac{H}{60^\circ} \tag{4.12}$$

Obtaining chroma and an intermediate value X:

$$C = V' \times S' \tag{4.13}$$

$$X = C \times (1 - |H' \mod 2 - 1|) \tag{4.14}$$

Obtaining intermediate RGB values:

$$(R_0, G_0, B_0) = \begin{cases} (C, X, 0), & 0 \le H' < 1 \\ (X, C, 0), & 1 \le H' < 2 \\ (0, C, X), & 2 \le H' < 3 \\ (0, X, C), & 3 \le H' < 4 \\ (X, 0, C), & 4 \le H' < 5 \\ (C, 0, X), & 5 \le H' < 6 \end{cases} \tag{4.15}$$

Obtaining final RGB values:

$$m = V' - C \tag{4.16}$$

$$(R, G, B) = (R_0 + m, G_0 + m, B_0 + m) \tag{4.17}$$

4.6.6 Application to Relativistic Path Tracer

The application of the Doppler effect to the relativistic path tracer was relatively simple, as it involved applying the Doppler effect wavelength modification to every bounce of the ray according to the velocity of the object.

5 Experimental Results

5.1 Albedo Integrator

The initial albedo renderings of a cube moving through the scene (see figure 1) demonstrate the visual warping of the object moving at 80% the speed of light considering only the rays from the object itself to the camera.

5.2 Path Tracing Integrator

The path tracing renderings of the cube (see figure 2 further demonstrate the affect special relativity has on both the cube itself and the surrounding scene, particularly regarding the light hitting the diffuse surface of the cube and the shadows, which seem to precede the cube.

Interestingly, in figure 2(b), the entire front face of the cube is black. This is due to the camera witnessing the unlit front face of the cube before it reached the lighting behind it.

Additionally, when the cube is moving right, a rotational effect is witnessed by the camera (see figure 3). This is easily explained by the fact that the front right edge of the cube is viewed by the camera first, while the other edges follow over time. As the speed increases, the entire right side of the cube is visible while the front is almost not. Moreover, the right side is very dark for the same reason as the unlit front face when the cube is moving away from the camera.

In our submission, you'll also find a file `cube-right-80percent-c.gif` showing the cube at 0.8c as it moves to the right over time. This animation also visualizes the way the shadows move as the cube moves very well. The shadows appear to precede the cube because, physically speaking, the light of the emitter reaches the cube far sooner than the camera. As a result, the shadows "show the future" in a sense, showing where the cube is located according to the emitter as opposed to where the cube is according to the camera. This effect should be offset by the time it takes for light to travel from the walls to the camera, which is why the effect is not extremely pronounced.

Finally, the path tracing rendering of the cbox scene (see figure 4 more clearly visualizes the way the integrator accounts for the views of objects in their local reference frame. The mirror ball in figure 4(b) reflects a view of a longer shadow and a much closer dielectric ball. Because the light reflected off of the mirror ball takes time to travel to the camera, the ball is "reflecting the future". Taking a specific example, first, the light of the dielectric ball reaches the camera first. Then the light of the mirror ball, reflecting the dielectric ball traveling towards the back of the scene, reaches the camera. Naturally, this results in a state where the camera shows effectively a later scene than that of the mirror ball. This explanation also applies to the longer shadow reflection which is the state of the shadow when the mirror ball moves towards the camera and away from the area light.

5.3 Doppler Shift

5.3.1 Albedo Integrator

The albedo renderings of a spaceship (credit and thanks to Cpt.Kirk on Sketchfab https://sketchfab.com/3d-models/republic-venator-star-destroyer-1e4aaa57565d4f098d01cf9c89e4e1e6) moving right (see figure 5 clearly demonstrates the Doppler effect on the wavelengths of light reaching the camera. Additionally, figure 6 shows how the transition of the Doppler effect becomes smaller as the camera witnesses more of the ship moving away from it.

One aspect of this Doppler effect implementation is how readily the light shifts out of the visible spectrum. This implementation simply clamps the wavelengths to the visible spectrum, hence the flat red color. The Doppler effect strength modification (see eq. 4.9) comes in handy for this particular case, tuning down the strength of the Doppler shift such that it is still visible to the camera. However, doing so is not particularly useful for an Albedo rendering as it simply spreads out the visible Doppler effect across the object.

5.3.2 Path Tracing Integrator

Figure 7 shows the Doppler effect applied to the path traced renderings of the cube from section 5.2. Note the blue shift of the cube as it approaches the camera compared to the red shift of the cube as it moves away. The aforementioned red shift is quite difficult to see in figure 7(c). However, this figure displays a more interesting effect where the light reflected onto the ceiling of the scene shows the full visible light spectrum, "pulled apart" by the Doppler effect.

6 Discussion

The objects and scenes rendered in section 5 were selected for a variety of reasons. A checkered cube was created to better visualize the warping effects of the finite speed of light, especially when the warping affects one side of the cube more than the other. Additionally, the length of the spaceship made it easier to find a position in which the Doppler effect only applied to part of the object. At 0.8c, the fact that the spaceship is entirely red shows that the ship is almost entirely moving away from the camera at any point in time visible to the current camera. In fact, objects moving a 0.8c generally had difficulty visualizing the Doppler effect, so the path tracer renderings of the Doppler effect used 0.4c instead.

7 Conclusion

In section 2 and 3, the background was laid for the Aris extension detailed in section 4. After implementation, the rendered images were listed in section 5 wherein objects warped and Doppler shifted in accordance with special relativity and the literature.

Regarding future work, we would want to see several extensions added to this Aris extension. The first would be the implementation of reference frames that are accelerating and not just moving at relativistic speeds. Additionally, as previously mentioned, the Doppler effect could be made more accurate using representations of the individual wavelengths composing the visible light in the scene as opposed to only red, green, and blue channels (although figure 7(c) demonstrates that this still produces a realistic spectrum of light when Doppler shifted). Finally, for the purposes of visualization, we would have liked to create

an alternative to t_0 where instead we specify the initial position of an object. This would prevent having to hunt for an object in a side by repeatedly creating albedo renders with varying t_0 to fit the object in the camera's view.

8 Division of Work

Ryan focused on the reading and writing the literature and background required for the implementation and worked alongside Tien to develop much of the relativistic path tracer. Tien focused on the implementation of the Doppler effect and produced the final renderings and their associated analyses.

References

[1] Shirish (SO user 160162). *What are the Lorentz Transformation equations when the relative velocity of an entity is in an arbitrary direction?* Physics Stack Exchange. URL:https://physics.stackexchange.com/q/562815 (version: 2020-06-30). eprint: https://physics.stackexchange.com/q/562815. URL: https://physics.stackexchange.com/q/562815.

[2] Jeremy Daniel, Cyrus A V Dolph, and Jean-Emile Elien. "Relativistic Ray-Tracing: The Appearance of Rapidly Moving Objects". In: (1997).

[3] Ping-Kang Hsiung and Robert Thibadeau. "Spacetime visualization of relativistic effects". In: *Proceedings of the 1990 ACM Annual Conference on Cooperation*. CSC '90. Washington, D.C., USA: Association for Computing Machinery, 1990, pp. 236–243. ISBN: 0897913485. DOI: 10.1145/100348.100384. URL: https://doi.org/10.1145/100348.100384.

[4] *HSL and HSV*. en. Page Version ID: 1258384559. Nov. 2024. URL: https://en.wikipedia.org/w/index.php?title=HSL_and_HSV&oldid=1258384559 (visited on 12/11/2024).

[5] Geng Lin. *Aris Renderer*. https://github.com/CMSC740-UMD/aris-renderer-student. 2023.

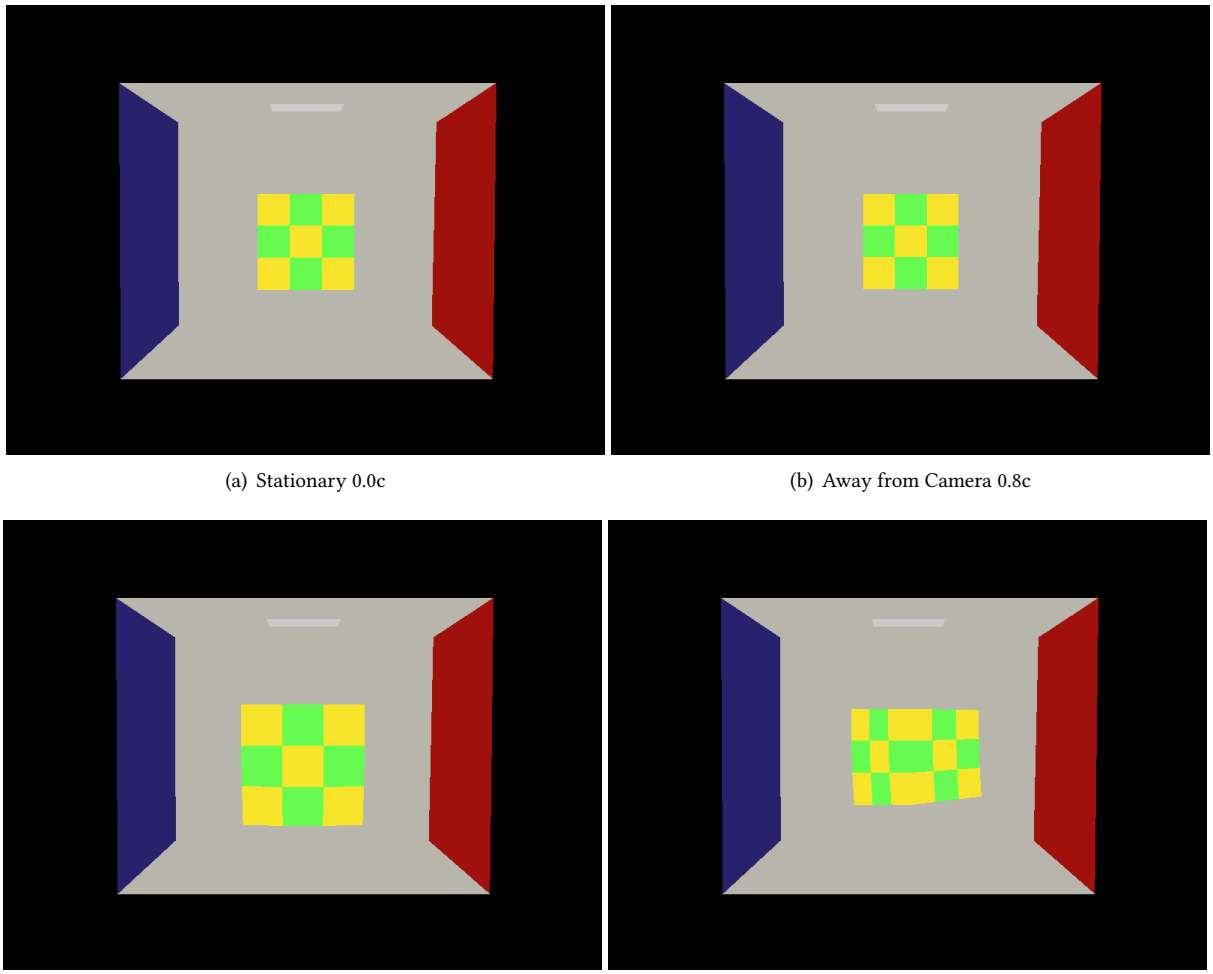
[6] C. Møller. *The Theory of Relativity*. International series of monographs on physics. Clarendon Press, 1972. ISBN: 9780198512561.

[7] *Spectral color*. en. Page Version ID: 1244084386. Sept. 2024. URL: https://en.wikipedia.org/w/index.php?title=Spectral_color&oldid=1244084386 (visited on 12/11/2024).

[8] James Terrell. "Invisibility of the Lorentz Contraction". In: *Phys. Rev.* 116 (4 Nov. 1959), pp. 1041–1045. DOI: 10.1103/PhysRev.116.1041. URL: https://link.aps.org/doi/10.1103/PhysRev.116.1041.

A Appendix

Begins on next page.



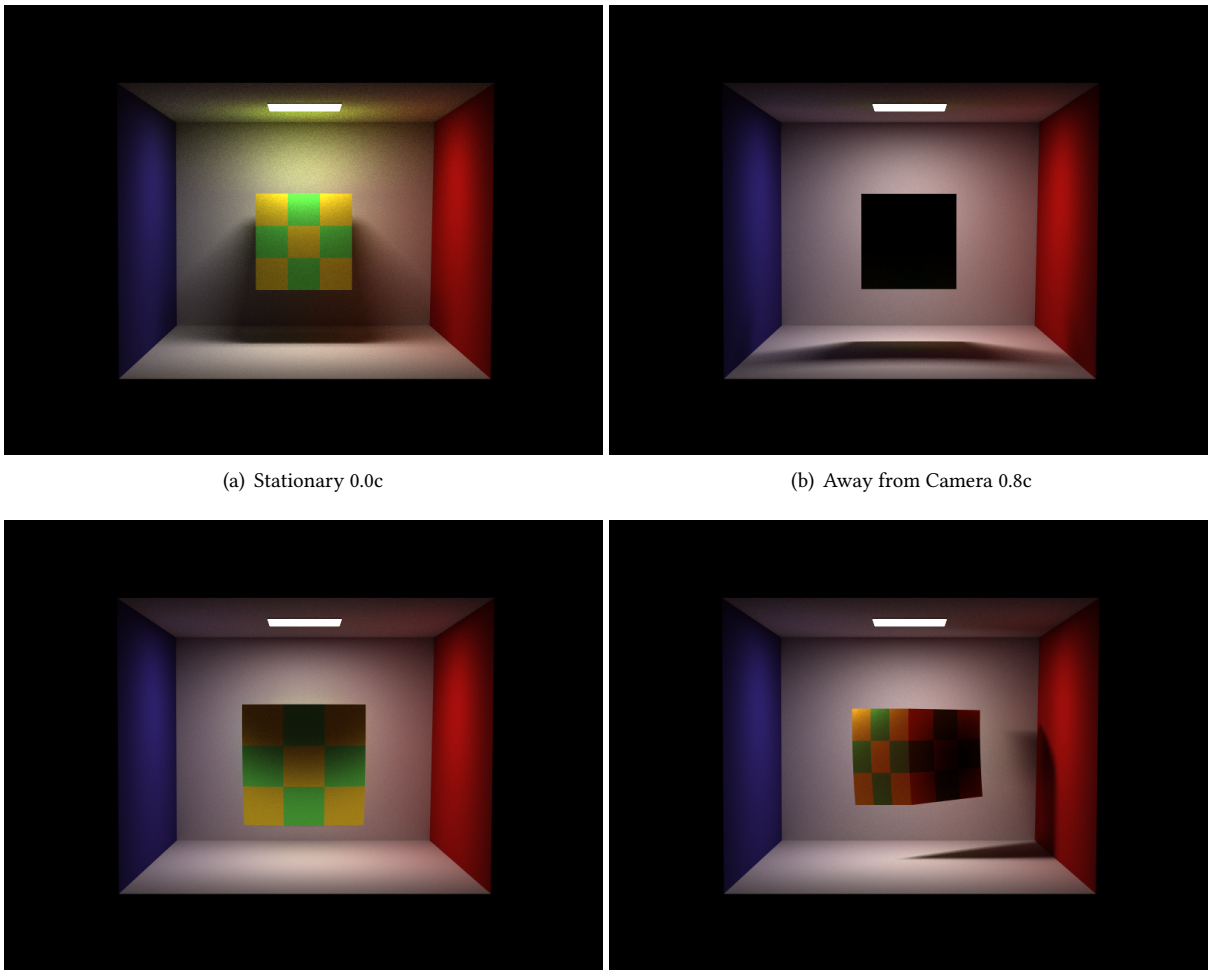
(a) Stationary 0.0c

(b) Away from Camera 0.8c

(c) Toward Camera 0.8c

(d) Right 0.8c

Figure 1: Albedo rendering of checkered cube at rest and moving at 0.8c.



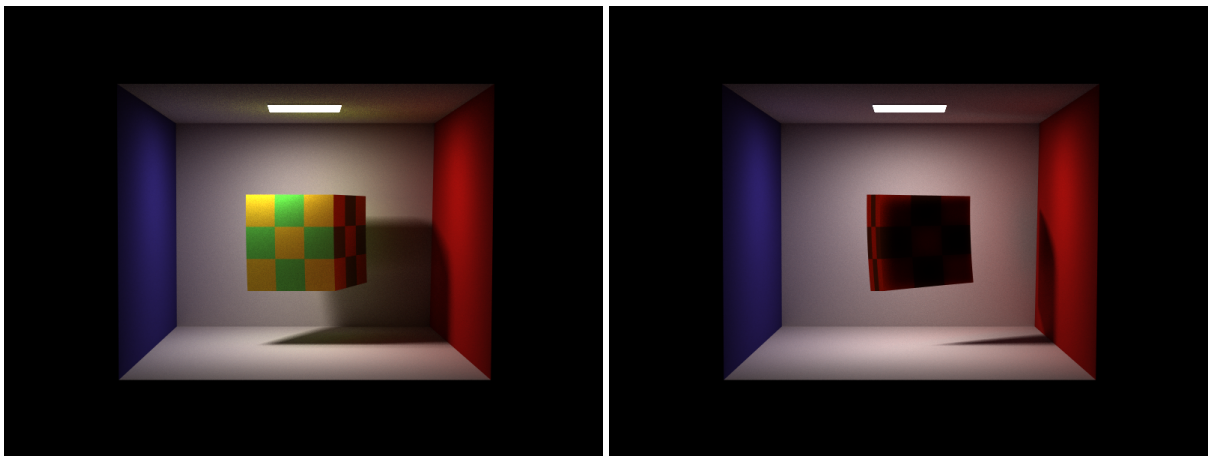
(a) Stationary 0.0c

(b) Away from Camera 0.8c

(c) Toward Camera 0.8c

(d) Right 0.8c

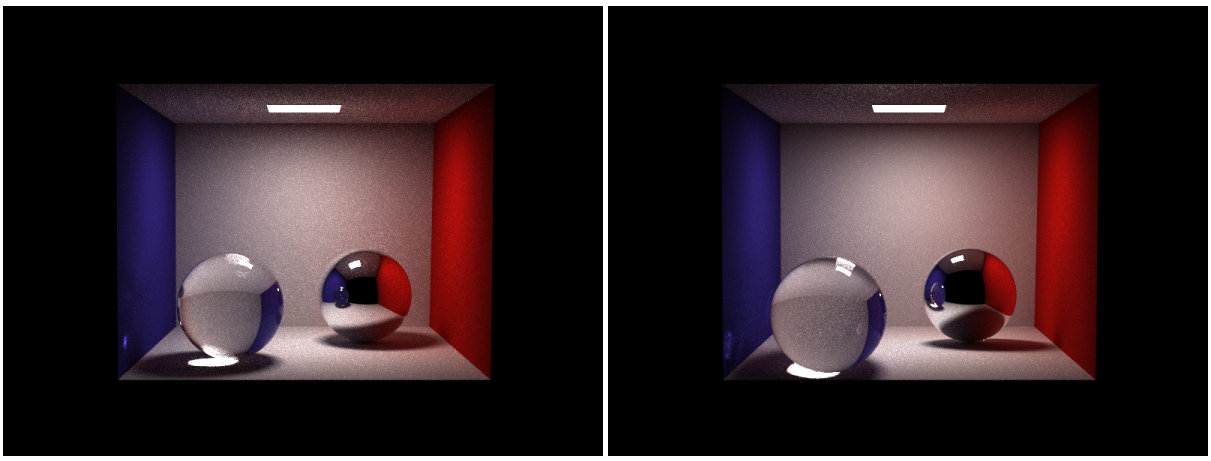
Figure 2: Path tracer rendering of checkered cube at rest and moving at 0.8c.



(a) Right 0.4c

(b) Right 0.99c

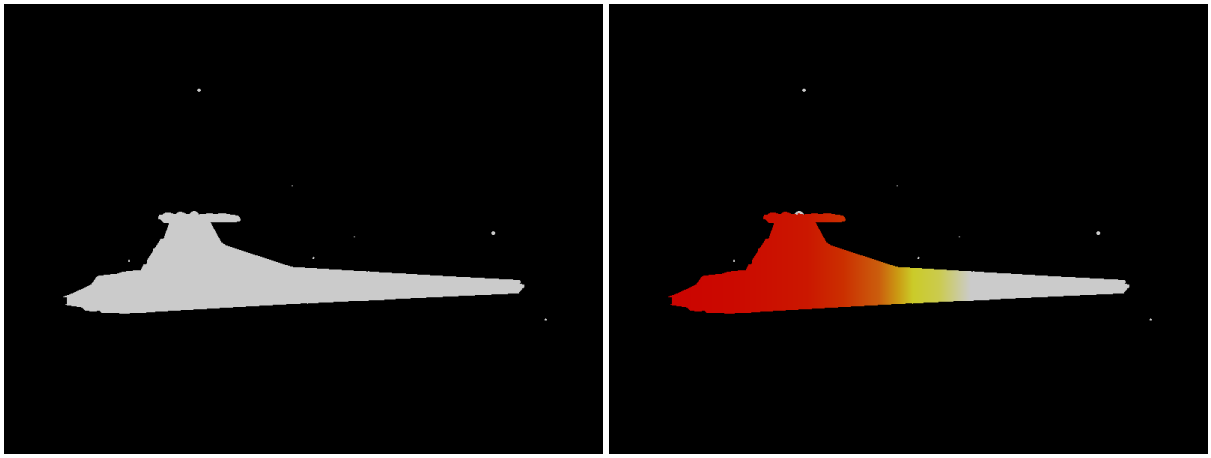
Figure 3: Path tracer rendering of checkered cube moving to the right at various speeds.



(a) Stationary

(b) Left Ball Away from Camera and Right Ball Moving Towards Camera 0.4c

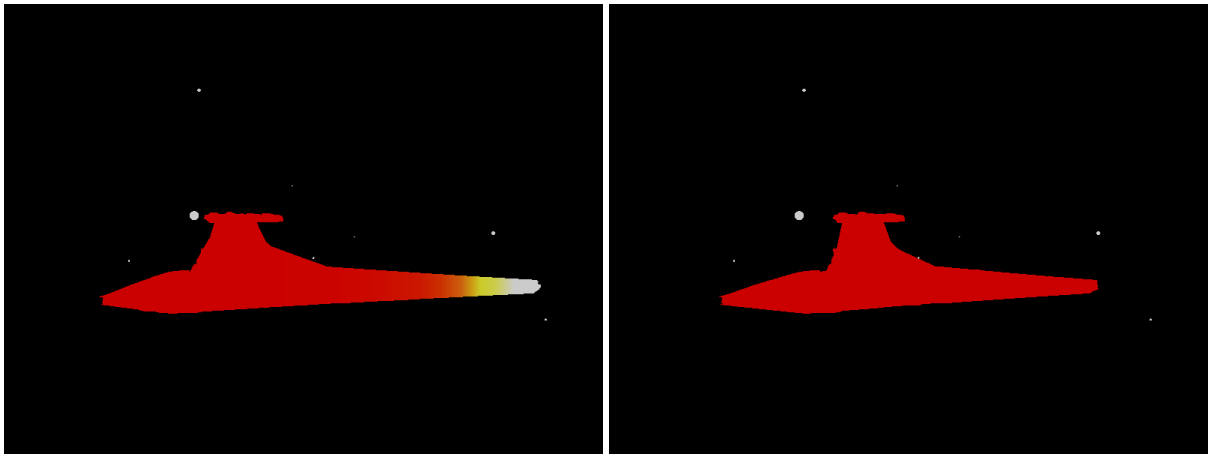
Figure 4: Path tracer rendering of mirror and dielectric balls.



(a) Before Doppler effect

(b) After Doppler effect

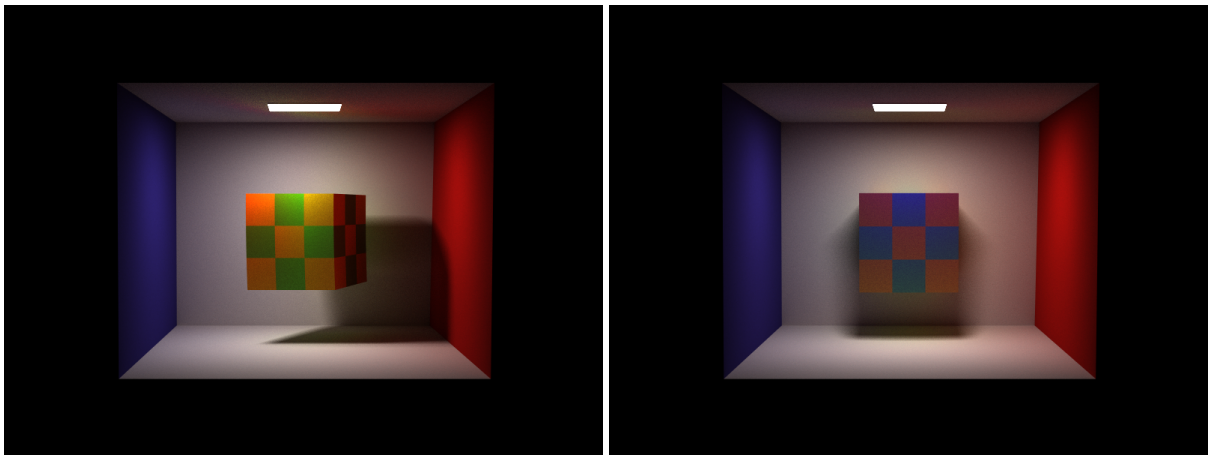
Figure 5: Albedo rendering of a space ship travelling to the right at 0.4c with and without the Doppler effect.



(a) 0.6c

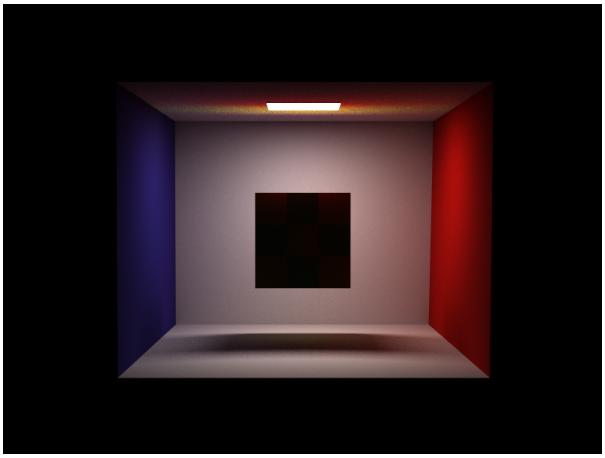
(b) 0.8c

Figure 6: Albedo rendering of a space ship travelling to the right at various speeds with the Doppler effect.



(a) Right 0.4c

(b) Toward Camera 0.4c



(c) Away from Camera 0.4c

Figure 7: Path tracer rendering of checkered cube moving at 0.4c.