Relativity Virtual Reality Project Report

King's College London

6CCP3131 Third Year Project

Leyan Ouyang

Supervisor **Prof. Eugene Lim**

Abstract

The project aimed to develop a virtual reality-based experience using Unity, Blender, and Oculus Quest. The goal was to provide an immersive and interactive experience for users to explore the view of a camera traveling close to the speed of light. Inspired by Einstein's theory of special relativity, this project explores how extreme velocities affect human perception, especially under the principles of Doppler effects, length contraction, and aberration angle. The purpose of this project was to convert abstract physics to a visually intuitive experience. In order to pursue a view of camera traveling at near-light speed, the development process involved 3D scene designation, mathematical calculations, and the construction of rendering models. Specificially, different visual effects were obtained by different implementations using different techniques: length contraction and aberration angle were achieved through vertex transformation and Doppler effect was modeled via GPU shader-based color manipulation. Users are able to navigate through various Iconic buildings and streets in London from the perspective of near-light speed camera, observing effects such as light aberration, length contraction, and color changes. Although the current version serves as a prototype, the project itself has a high future potential in the utility of game and VR development by successfully demonstrating the special relativity views. To enhance user engagement, future versions may include interactive features that allowed users to adjust heights and control the trajectory of the camera in real time.

1 Introduction

1.1 Background and Motivation

In 1905, special relativity was first proposed by Albert Einstein who published his paper titled *On the Electrodynamics of Moving Bodies* [1]. In the thesis, Einstein redefined the concepts of space and time where he also proposed the idea that motion is relative. Einstein introduced the constancy of light and relativity of motion. At that time, the theory challenged classical notion of simultaneity and established the foundation for time dilation and length contraction theory. However, visualizing the effects of special relativity was not possible in 20th century due to their abstract and intuitive nature. Nobody truly knew what the world under special relativity might look like, except from theoretical calculations. In the past, people understand the world of special relativity from papers and books, but the emergence of virtual technologies made it possible to build a world from the perspective of near-light speed observer. Special relativity can then move beyond papers and pens to eyes of people. It can not only model space, colors, and velocities, it can lead people into

a non-classical world where complex theories can be monitored and translated to sensory stimulation.

Over the past century, technology bloomed rapidly. As more and more scientists chose to dedicate themselves to the study of special relativity, many conclusions were drawn and proved by experiments and mathematical calculations. However, most of them are still non-intuitive and difficult to understand. As virtual reality took the place, it had become possible to create an immersive experience from computer coding and 3D modelling which enabled user to explore the physical phenomena of special relativity. In addition to its academic value, the project provides students with a wider field of view in learning special relativity. By transforming abstract physical theories to immersive experience, the project can be used as a teaching tool for physics teaching in high schools or universities, and can also serve as part of interactive displays in science museums or public exhibitions. Visualizing the special relativity effects can enhance educational communication, but also raise public interest in physics theories. To conclude, the project paves a way for scientific visualizations that help people to understand physics. Based on the above idea, this project attempted to construct an immersive observation perspective near the speed of light through VR technology, allowing people to see and feel the world of special relativity.

1.2 Project Overview

In order to build a virtual reality experience based on special relativity, the project was developed using Unity and Blender. The first person experience was achieved by rendering the scene on a computer and streaming it to the Oculus Quest headset. In the project, length contraction, aberration angle, and Doppler effects were simulated using Unity. Blender was used to create the 3D scenes and street views. Through referencing the iconic buildings and streets in London, establishing the trajectories of the vertices, and implementing algebraic transformations, this project recreates a visual experience from a perspective close to the speed of light.

Due to the non-classical visual effects involved in special relativity under high-speed motion, how to recreate these visual phenomena while ensuring user immersion has become a major challenge for this project. For instance, the simulations of multiple color shifts requires the performance and accuracy of shaders. Vertex transformations had to be perfectly followed and bent in a visible trajectory. The greatest challenge is the consistency with the theory. Therefore, multiple attempts and adjustments in modeling, calculations, optimization, and user experience were made during the construction.

The project offered an opportunity to develop a more structured understanding of virtual

reality design and simulation, particularly in learning how to translate the abstract physical theories to visualized experience. In addition, practical experience with Unity and Blender was gained through hands-on development. Shader implementation techniques were also explored during the process. More importantly, a deeper understanding of special relativity was gained through establishing mathematical simulations and calculations. Several challenges were encountered while implementing shaders and designing 3D environments and assets. In future iterations, the project could be extended with additional interactive features, such as a more variable camera paths and more user-controlled parameters. Further improvements should also focus on calibration of the color RGB for shaders and optimizing rendering performance for VR platforms.

2 Physical Theories and Related Work

2.1 Special Relativity: Fundamental Principles

The fundamental principles of special relativity was established by Albert Einstein a hundred year ago when he made two assumptions. Firstly, all physical laws are the same in every inertial frames. In other word, there is no reference frame that is absolutely stationary. Secondly, the speed of light in vacuum is the maximum speed in nature, it is independent of the motion state of the light source or the observer. These two assumptions had break the idea of absolute time and space in Newton's classical physics, they also redefine the definition for time and space. Within the theory, the idea of "'happening at the same time" had changed from absolutely to relatively. Whether or not the two things happen at the same time does not depend on any chronograph, it depends on the motion of the observer.

Based on the theory and assumptions provided by Albert Einstein, several non-classical physical effects have been derived. Length contraction states that when an object travels at a near-light speed relative to an observer, its length along the direction of movement will become shorter in the observer's eyes. The equation for length contraction is given in Equation 1.

$$L = L_0\sqrt{1 - \frac{v^2}{c^2}} \tag{1}$$

In the equation, L_0 represents the length of the object in its own stationary frame, which is also called proper length. L is the length of the object measured by the observer, and v represents the relative speed of the object from the observer. Thus, from the perspective of an observer traveling at near-light speed, objects in the direction of motion appear contracted.

Time dilation can be seen as the dual concept to length contraction. In time dilation

effect, when an clock is traveling at near-light speed relative to the observer, the observer would find the time shown on the clock ticks slower. The equation for time dilation is shown in Equation 2.

$$t = \frac{t_0}{\sqrt{1 - \frac{v^2}{c^2}}} \tag{2}$$

Both length contraction and time dilation are both formed from Lorentz transformation which is the relativity of time and space caused by the consistency of the speed of light.

2.2 Visual Effects of Relativistic Motion

2.2.1 Length Contraction

In order to demonstrate an immersive experience for users to see from the perspective of a near-light speed observer, it is crucial to consider about the one of the core conclusions of special relativity: Length contraction. As discussed in the section 2.1, when an observer moves in a certain direction at a speed close to speed of light relative to other objects, the length of the objects in the direction of moving will be compressed. This effect originated from Lorentz transformations and its formula is shown in Equation 1. One thing to be noted is that, this effect only reflect in the direction of motion of the observer, the size perpendicular to the direction of motion remains unchanged. In the project, length contraction is developed by analyzing the coordinates of the vertices and redefining them.

2.2.2 Relativistic Doppler Effect

The Doppler effect in the frame of special relativity not only considers changes in wavelength, but also introduces the influence of time dilation. When there is a near-light speed relative motion between the observer and the objects, the observed light frequency changes according to the frequency formula below in equation 3:

$$f = f_0 \sqrt{\frac{1 + \frac{v}{c}}{1 - \frac{v}{c}}} \tag{3}$$

In the equation, f_0 represents the frequency of light emitted from the objects, f is the frequency of light achieved by the observer, v represents the relative speed of the object from the observer. Positive v means that the object is moving closer, negative v means that the object moving away. The equation revealed the pattern of color shifts for an observer traveling at a near-light speed. While looking directly forward to the front where objects travel closer to the observer, the frequency of light traveling from the objects to the observer

become higher, so that the wavelengths tend to be shorter which is called a blue-shift. On the contrary, while looking backward where objects travel away from the observer, the frequency of light traveling from the objects to the observer become shorter, so that the wavelengths tend to be longer which is called a red-shift. The color shifts in the simulation is adjusted through post-processing shader which will be discussed in methodology.

2.2.3 Aberration of Light

Aberration of light refers to an optical phenomenon where for an observer traveling at near-light speed, the light rays from different directions undergo a deviation in their incident angles. This results in a visually focused forward view for the observer, meaning that the objects in the scene are visually compressed into a small angle cone in front. This creates an illusion of "tunneling effects" or "central focusing".

The formula for the aberration angles is provided in equation 4, and it can be derived directly from the Lorentz transformation of the wave vector between two inertial frames:

$$\cos\theta' = \frac{\cos\theta + \frac{v}{c}}{1 + \frac{v}{c}\cos\theta} \tag{4}$$

Where θ represents the incident angle of light in a stationary reference frame and θ' represents the incident angle observed from a relatively moving frame. This result shows that light rays originally coming from wide angles in the rest frame will appear more concentrated toward the direction of motion. The higher the speed, the stronger this forward compression effect becomes — leading to a significantly distorted field of view that simulates "tunnel vision".

From a rendering perspective, the aberration effect can be implemented by adjusting the coordinates of the object's vertices (physically accurate), or by using screen-space post-processing effects, where distorting shaders are applied to bend the image based on the camera's view direction.

The relativistic Doppler effect and aberration angle are fundamental consequences of Lorentz transformations, and have been thoroughly analyzed in modern physics literature [5].

2.3 Existing Relativity Visualization

In the past few decades, the investigation of the visual effects in special relativity was mainly focusing on 2D images and non-interactive animation. For instance, some physic teaching sources demonstrate the phenomena of length contraction and time dilation through 2D games and flat videos. An example of educational gamification of relativity is the 2D

game *Velocity Raptor* developed by TestTubeGames [3]. [1]. This game aimed at helping students to understand special relativity through interactions. Players can control a raptor at near-light speed and experience the effects of special relativity such as length contraction, time dilation and light aberration. The game's dynamic visuals change in real-time based on player's speed.

While *Velocity Raptor* serves as a more intuitive introduction to relativity for learners, an undergraduate project developed by computer graphics majored students at the University of Science and Technology of China (USTC) further expands the concept of visualizing special relativity by dynamic simulation and 3D physical modeling. The project was based on openGL platform[2]. It not only achieved visual distortion simulation in 3D stationary scenes but also combined Doppler effects and color shifts. The project also design different perspectives for third person and first person. Below are two video captures from the project video uploaded by USTC.

(a) Camera moving away from the scene

(b) Camera moving closer to the scene

Figure 1: Simulated relativistic views with camera moving in opposite directions.

Compared to related works, this VR project mainly focused on the first person immersive viewing experience by setting the user as a near-light speed observer through virtual simulations. Users can experience the spatial distortion and color shifts under special relativity in an intuitive way. Although the projects have different goals and focusses, they have jointly promoted the graphic visualization research of special relativity.

3 Methodology

As the project was developed collaboratively, this section details the components implemented by the author, including camera system setup, shader-based visual effects, and scene

[1] https://testtubegames.com/velocityraptor.html

[2] http://staff.ustc.edu.cn/~lgliu/Courses/ComputerGraphics_2024_spring-summer/Projects/index9.html

modeling.

3.1 Project Blueprint and Personal Objectives

This project was developed collaboratively by a group, with multiple members contributing to different components. This section focuses on the parts undertaken by the author during the development process, including early scene import, camera perspective configuration, implementation of Doppler effects, and partial modeling work. Some modules developed in the early stages were later deprecated during integration with other members' work. However, these components played a crucial role in the overall learning and hands-on process, contributing significantly to the understanding of implementation techniques and system constraints.

Initially, the overall blueprint of this project includes these three objectives. Firstly to construct a virtual scene that includes urban buildings and recognizable landmarks. The second is to set up a first-person perspective camera with physical motion laws which then became a stationary camera with objects in motion. The third is to use post=processing methods to achieve graphic rendering of relativistic visual effects, ultimately allowing users to personally experience the "visual reconstruction" of the real world in near-light speed motion through devices such as Oculus Quest.

To achieve an immersive first-person experience, Unity was adopted as the core development platform, with Blender used for modeling. Post-processing effects were implemented through shader programming to simulate relativistic phenomena such as Doppler shift and light aberration. The final project was deployed on Oculus Quest, forming a complete virtual reality interaction pipeline. These tools were chosen for their flexibility and compatibility with VR development. The combination of real-time rendering and programmable shader allowed for the creation of dynamic, perception-driven visual effects. In the following sections, each stage of the development process will be elaborated in detail.

3.2 The Earliest Prototype

After completing the preliminary project plan, the author immediately attempted to construct a simple prototype to quickly verify the feasibility of the visual effects of special relativity in Unity. As a result, a lightweight and intuitive prototype was constructed at an early stage in order to quickly validate the core visual model and the expected standard of the project. To allow for rapid validation of the core concept with minimal resources, the prototype was intentionally kept small in scale and focused on clarity and simplicity. As a

result, the prototype utilized an open map resource from online webpage[3] and was directly imported into the project scenario as the initial testing environment. A first-person camera was configured to move along a fixed street provided in the map resource while applying C# codes for sight direction control. So that the camera can look around 360 degrees while traveling. (see Appendix 6)

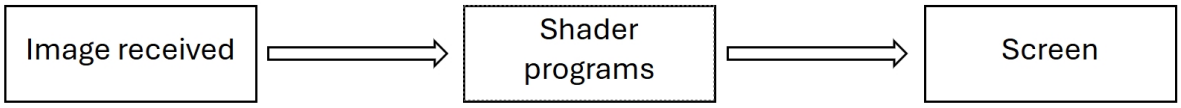
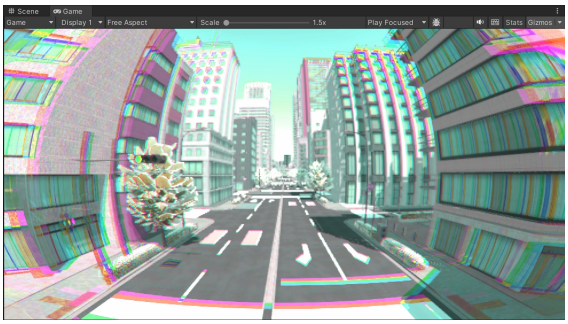


Figure 2: Process of how post-process pipeline works.

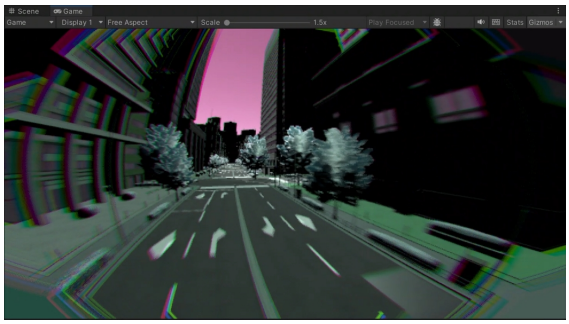
The author attempted to achieve visual relativistic effects through Unity's built-in post-processing pipeline. The post-processing pipeline refers to a series of visual effects applied to the final rendered image before it is delivered to the screen. To be intuitive, the process is shown in figure 2. These effects are typically implemented through shader programs and are often used in various first-person viewing games like car games to simulate visual distortions and enhance realism. Rather than manipulating the geometry directly, these effects were applied to the final rendered frame to achieve perceptual realism, which is also the reason why shader is a perfect choice as it can operate easily in parallel with other effects that acquire high computing power. All the color and distortion effects in this prototype was accomplished by shader programs. The use of post-processing volumes and shaders in Unity allows for flexible, real-time manipulation of visual effects in virtual environments [4].

The model of the prototype is demonstrated as figure 3 below.

[3] https://www.aigei.com/s?q=japanese+city



(a) When a camera moving at a speed along the street is looking forward.



(b) When a camera moving at a speed along the street is looking backward.

Figure 3: Prototype built at the earliest state where visual effects were accomplished by post-processing methods

In this prototype, the author assumed a moving camera that was programmed to move along a fixed street. Whereas the buildings were stationary while passing by and the color of the screen will be rendered differently according to the angle between the sight of the camera and the fixed camera moving direction. This prototype was done individually at the early exploration state of the project when every parts of the group was not collaborated. Although the technical implementation at this stage was relatively rough and some sections within this prototype was not utilized in the final version, this version played a significant role in understanding key aspects such as shader writing and camera control logic laying the foundation for the development of subsequent versions.

3.3 Implementation of Relativistic Visual Effects

The implementation of the relativistic effects and high speed motion was entirely based on shader programs. Doppler effects, aberration and motion blurring effects can be simulated by post-processing pipeline. The logic behind the simulations are elaborated below. Length Contraction was not successfully simulated by post-processing method because the individual computation for every object's length contraction cannot be done by a post processing volume that can only change the total render effect.

3.3.1 Understanding Shader in Unity

To better understand the technical approach adopted in the Doppler shift and aberration simulations, a brief explanation of how shaders function in Unity is provided. Shaders are small programs that run on the GPU and determine how objects are drawn on the screen. In Unity, shaders are responsible for controlling how light interacts with materials, how textures are applied, and how colors and visual distortions appear. Rather than directly

manipulating 3D object geometry, shaders allow for real-time, pixel-level manipulation of visual effects during rendering.

Unity supports several types of shaders through the render pipeline, including surface shaders, vertex/fragment shaders, and post-processing shaders. In this project, post-processing shaders were used to apply global color shifts to simulate relativistic Doppler effects. These were implemented using Unity's built-in Post Processing Stack (Volume System).

The Unity render pipeline works in the following stages:

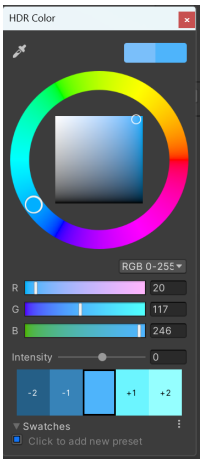
- Objects and materials are defined in the scene.
- The render pipeline collects camera, light, and material data.
- Shaders define how objects are visually rendered — including distortion, color changes, and texture mapping.
- The final frame is rendered and delivered to the screen.

Instead of modifying object meshes, the Doppler effect simulation relies on dynamically changing the color filters via post-processing shaders based on the observer's viewing direction. These effects run in parallel with other computations, making them efficient and real-time responsive.

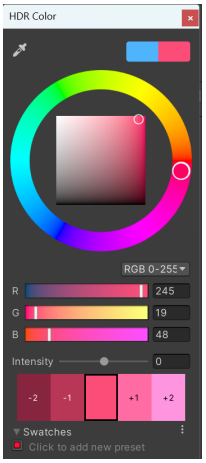
This modular shader setup allows each rendered frame to reflect relativistic visual changes accurately while maintaining smooth transitions between blue-shifted and red-shifted areas of the screen.

3.3.2 Doppler Effects in Unity

As discussed in section 2.2.2, When the observer and the observed objects are traveling relatively to each other, the frequency of light transmitting between them will change. When the object is relatively moving closer to the observer, the light received turns blue. On the contrary, if the object is relatively moving away from the observer, the light received turns red.



(a) Camera moving forward (blue shift).



(b) Camera moving backward (red shift).

Figure 4: Color filters used to simulate Doppler shift in opposite directions via Unity post-processing volumes.

To simulate the relativistic Doppler effect, the color channels of the rendered scene were dynamically adjusted through shader programs in Unity. At the very first step, two post-processing volumes were defined. One is for the front, which defines the RGB color filter for the frame received when facing straightly front while the objects traveling backward in the visual output. The other is for the back, which defines the RGB color filter for the frame received when facing straightly backward while the objects traveling away from the observer in the visual output. The parameters of the color filters are presented in figure 4.

Subsequently, the configured color filters were applied to two separate post-processing volumes. In Unity, the post-processing volumes are containers that apply visual effects based on prepared parameters for instance color grading , distortion, and visual blur to the rendered scene. The volumes can be configured as local volumes which they can only act on a part of the scene, or they can be configured as global volumes which they can act on the whole scene. In this project, the two volumes were configured as global volumes which they can affect the color of the whole scene before delivered to the screen.

To calculate the relativistic Doppler shift, a directional cosine between the observer's view and the global X-axis was computed using a dot product.

```
float cosThetaRight = Vector3.Dot(flatForward, xAxis);    //
    Facing +X
float cosThetaLeft = Vector3.Dot(flatForward, negXAxis); //
    Facing -X
```

Code 1: Angle ($\cos\theta$) between the camera's view direction and the X-axis

The Doppler factor was then calculated as follows:

```
float dopplerRight = sqrt((1 + velocity * cosThetaRight) / (1 -
    velocity * cosThetaRight));
float dopplerLeft  = sqrt((1 + velocity * cosThetaLeft) / (1 -
    velocity * cosThetaLeft));
```

Code 2: Doppler shift factor calculation

The factor determines how the RGB values should be shifted based on the viewer's direction and speed. After the factor being determined, a smoothed blend factor was used to interpolate between the two color filter volumes, ensuring visual continuity as the observer changes view directions.

```
float blendFactor = Mathf.Clamp01((cosThetaRight + 1f) / 2f); //
    // Normalize from -1 to 1 to 0 to 1
blendFactor = Mathf.Lerp(frontVolume.weight, blendFactor,
    Time.deltaTime * blendSpeed); // Smooth transition
```

Code 3: Blendfactor for smooth transition

The complete Doppler Shift script in Unity was attached in Appendix B 6.

Another thing to be noted is that due to the correlation between Doppler effect and relativistic redshift/ blueshift with observation direction, in this project, it is necessary to adjust the visual results in different directions according to the orientation of the viewing angle. To achieve this goal, the author set up two independent volumes one to simulate the blueshift caused by the forward viewing angle, the other one to simulate the redshift caused by the backward viewing angle. The orientation of the camera changes in real-time during the rendering process of each frame, which means that the program needs to dynamically calculate the Doppler factor based on the current camera orientation and the two directions, and adjust the ratio of the appearance of the two volumes accordingly. This ensures: a separately controlled color shift based on the front and back directions, smooth visual color transit and no abrupt color jumps, and an entirely covered Doppler shift effects. Later, this implementation was optimized and adapted for VR environments, as detailed in the following section.

3.3.3 Doppler Shift for VR Adaption

Unlike the earlier two-directional approach, the VR version simplified the computation into a unified cosine-based logic, enhancing both efficiency and compatibility with XR tracking. In VR environments, the observer's viewpoint is controlled by the headset's motion tracking rather than a standard game camera, so the simulation must reference the XR

camera's transform to correctly reflect the user's real-time head movement. This is why the implementation uses `xrCameraTransform` instead of relying on `Camera.main`.

To enable the Doppler effect in immersive virtual reality environments, significant adaptation work was carried out on the original shader-based simulation logic. The initial implementation was designed for a standard camera view and did not account for the motion and orientation tracking used in VR headsets.

In VR, the observer's camera is no longer fixed but is controlled by the user's head movement in real time. As a result, the direction of observation constantly changes, and the Doppler shift must be recalculated every frame accordingly. The previous static setup was replaced with a dynamic camera binding system that references the VR camera transform.

Code Highlights for XR Adaptation:

```
xrCameraTransform = Camera.main.transform;
// Automatically binds the XR camera if not already set
```

Code 4: Auto-binding XR Camera

```
Vector3 flatForward = new Vector3(...);
float cosTheta = -Vector3.Dot(flatForward, xAxis);
// Computes observer's direction relative to the world axis
```

Code 5: Directional Cosine Calculation

```
float dopplerFactor = sqrt((1 + velocity * cosTheta) / (1 - velocity *
    cosTheta));
float logDoppler = log(dopplerFactor);
// Translates Doppler shift to blending ratio using log mapping
```

Code 6: Doppler Factor and Log Mapping

```
frontVolume.weight = Lerp(...);
backVolume.weight = Lerp(...);
// Smoothly interpolates front and back volumes
```

Code 7: Dual-Volume Blending

The variable `xrCameraTransform` is named after Unity's XR framework, which supports VR, AR, and MR devices; in this project, only the VR functionality is utilized.

This VR-friendly adaptation not only allowed the simulation to function properly in virtual reality, but also significantly optimized the code. Instead of computing two independent Doppler directions, the updated program compressed the logic into a single cosine calculation and used logarithmic mapping to enhance perceptual contrast. The entire system runs

in real-time and synchronizes seamlessly with the user's head rotation in a headset, creating a fully immersive relativistic Doppler visualization. In addition to VR camera integration, this version of the simulation also introduced multiple improvements compared to the original Unity implementation. Compared with the original Unity version, the Oculus Quest implementation includes several technical upgrades:

- **Color Channel Manipulation:** Rather than relying solely on post-processing volume weights, the XR version directly modifies RGB color channels via `ColorAdjustments.colorFilter`, providing a clearer and more accurate visual representation of relativistic redshift and blueshift.
- **Unified Directional Logic:** Instead of computing two separate Doppler factors for the front and back directions, the XR version simplifies the logic into a single $\cos\theta$-based calculation, reducing code redundancy while improving runtime efficiency.
- **URP Compatibility:** The XR version uses `UnityEngine.Rendering.Universal` with URP-supported volume overrides, making it more compatible with Unity's modern rendering systems and future-proof for extended VR development.
- **Enhanced Blending Control:** The use of `Lerp()` for both volume weights and color transitions results in smoother, more perceptually natural blending across the user's field of view.

Together, these refinements improve both the physical accuracy and visual fidelity of the simulation, allowing users to experience real-time relativistic effects in a truly immersive and optimized environment. The complete Doppler Shift script in Oculus Quest was attached in Appendix C 6.

3.3.4 Aberration Angle: Shader's Method

According to the theory, aberration angle represents the changes in the angle of the incident light in the observer's field of view while the observer's moving at a velocity v. This means that in the eyes of a traveling observer, all light will be squeezed forward, the field of view becomes more focused on the central front and sparser at the edges of the field of view. In the final project simulation, the aberration angle effects was simulated by recalculating the coordinate of every vertices of each object to achieve the overall compression effect of visual angles. This approach provides a mathematically accurate simulation of view compression and offers finer control over 3D object trajectories in Unity.

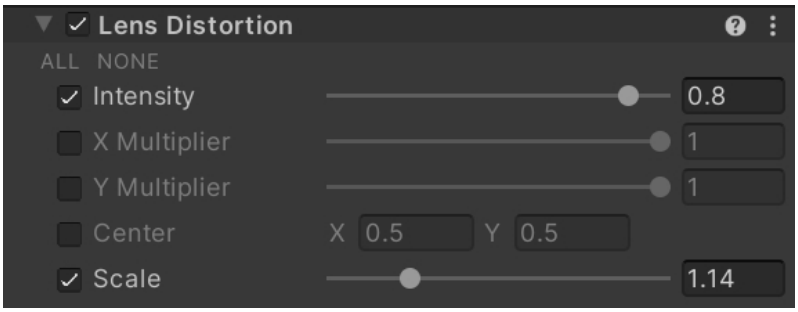
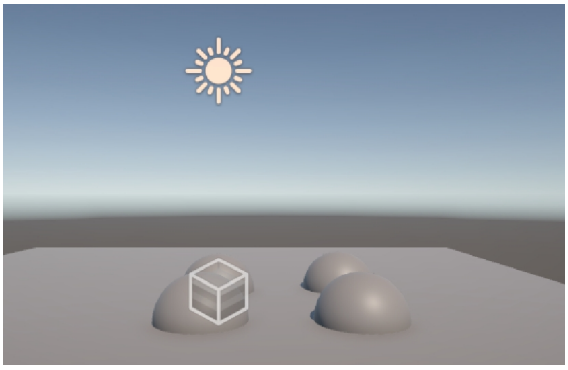
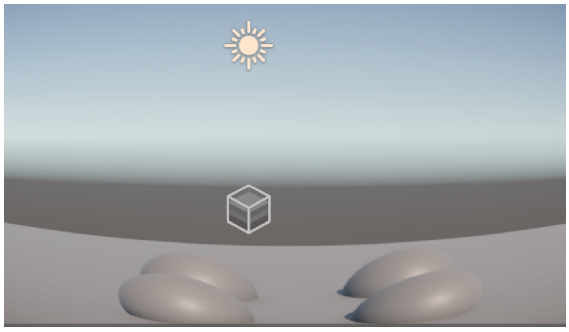


Figure 5: Lens Distortion parameter in post-processing shader module

In addition to the vertices recalculating method, post-processing methods also possess the ability to simulate aberration effects. By stretching the frame received at the image center and compressing the frame received at the image edges through Lens Distortion, a view with all the objects compressing to the frame center can be monitored. Figure 6 presents a comparative visualization: one using post-processing lens distortion to simulate aberration angle, and the other without any aberration simulation.



(a) Camera's view without aberration angles from shader.



(b) Camera's view with aberration angles from shader.

Figure 6: Comparison between having post-processing aberration angle and without post-processing aberration angle

Although applying post-processing method to simulate aberration angle is not as accurate as applying the vertices method. One can still write a shader program that can recalculate the intensity of the stretching of the image during every rendered frame in order to improve the mathematical accuracy while applying shader to simulate aberration angle. The programming process is shown below.

To calculate the relativistic aberration angle, the angle θ between the camera's view direction and the motion direction is first computed by the dot product formula:

$$\cos\theta = \vec{v}_{\text{view}} \cdot \vec{v}_{\text{motion}} \tag{5}$$

This is implemented in Unity as:

```
float cosTheta = Vector3.Dot(flatForward, movementDir);
```

Code 8: Dot product to compute $\cos\theta$

Then, using the special relativity aberration formula:

$$\cos\theta' = \frac{\cos\theta + \frac{v}{c}}{1 + \frac{v}{c}\cos\theta} \tag{6}$$

One can dynamically control the shader's lens distortion based on θ' to simulate central compression:

```
float aberrationFactor = Mathf.Clamp01(1f - cosThetaPrime); // Smaller
    theta' means more compression
lensDistortion.intensity.Override(aberrationFactor * maxDistortionValue);
```

Code 9: Mapping $\cos\theta'$ to distortion intensity

3.3.5 Aberration Angle: Vertices Method (Final method being used)

In addition to using post-processing shaders, a more mathematically accurate method was implemented to simulate the aberration angle: by directly transforming each vertex of the 3D model based on relativistic formulas. This approach ensures that the distortion of the entire geometry is physically meaningful and consistent with theoretical predictions.

Each 3D object in the Unity scene is defined by a mesh structure composed of vertices. The project script first extracted the vertex data from the object's mesh, transformed them into world space, and then converted their Cartesian coordinates into spherical coordinates to calculate their angular position relative to the direction of motion.

To apply the aberration transformation, the relativistic aberration formula was used:

$$\cos(\theta') = \frac{\cos(\theta) - \frac{v}{c}}{1 - \frac{v}{c}\cos(\theta)}$$

This equation determines the new angle θ' after applying the relativistic transformation, where θ is the original angle between the vertex and the observer's direction of motion.

Once the new angle θ' was obtained, the vertex coordinates were recalculated and transformed back into Cartesian space to update the object's geometry in Unity.

Unlike post-processing shaders, which only modify the final image by stretching or distorting pixels on screen, the vertex transformation method allows for true spatial distortion of every object. This results in more accurate simulation of directional compression and

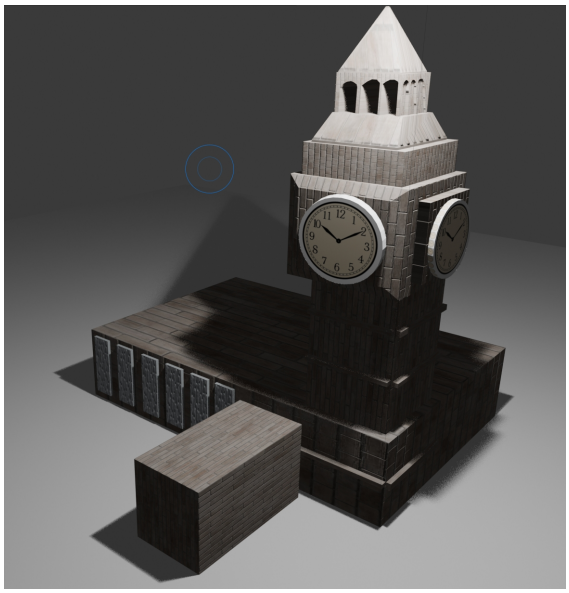
field-of-view changes under near-light-speed motion, particularly when observing the geometry of complex 3D objects from different angles. Compared to shader-based post-processing, which distorts the final image in screen space, the vertex transformation method applies the relativistic aberration directly to the 3D geometry. This ensures a physically accurate simulation that remains consistent across different viewing angles and object distances. Although shaders offer performance advantages, vertex-level manipulation provides a more precise and immersive experience in this project.

The complete script for aberration angle distortion intensity is attached in Appendix C 6.

3.4 3D Modeling

When thinking of the streets of London, one of the first landmarks that comes to mind is undoubtedly Big Ben. As a matter of fact, the author constructed a Big Ben model using Blender striving to present its most recognizable side in virtual reality. The first Big Ben model was built in Unity with only a few simple blocks. However, the author soon found out that Blender appears to be more professional at 3D building constructing. The pictures below show a comparison between the Big Ben built initially with Unity and the Big Ben built with Blender.

(a) Big Ben built with Unity.



(b) Big Ben built with Blender.

Figure 7: Comparison between the two "Big Ben"s between built by Unity and Built by Blender

The Big Ben model was built from scratch in Blender. It started with a basic low-poly shape and was improved step by step using simple modeling tools like extrusion and

subdivision. UV unwrapping was used to apply textures, and materials were adjusted to make it look more realistic under lighting in Unity. The clock shown on the Big Ben made with Blender is implemented by drawing image textures. Once completed, the model was exported and integrated into the Unity scene as an element of the London environment.

4 Results

The final simulation presents a dynamic first-person experience that captures relativistic visual phenomena as the user moves through the virtual London environment. Figure 8 gives the rendered scene under near-light-speed conditions, where relativistic Doppler shift and aberration are visually noticeable.

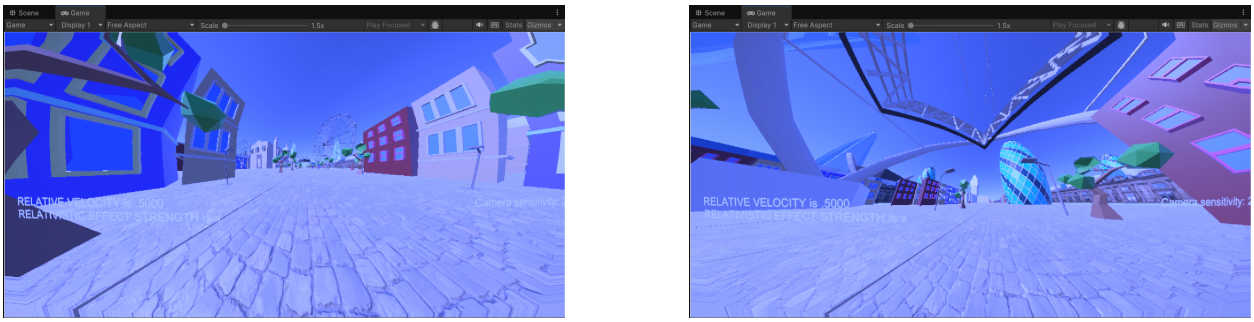


Figure 8: Final result

Upon launching the application and wearing the Oculus Quest, users are immediately placed on a virtual London street. As they move forward at near-light speed, buildings appear compressed, and the colors of the surroundings dynamically shift based on their relative direction of motion. Looking forward, the streets stretch with a blue tint, while glancing backward reveals redshifted scenery. This seamless transition allows users to perceive the effects of relativity not as abstract equations, but as real, intuitive experiences.

5 Discussion

This project successfully implemented a virtual reality experience based on Unity and Blender, combined with the Oculus Quest headset, aimed at visualizing key visual effects under special relativity, including length contraction, aberration angle and Doppler effect. Users can intuitively observe the change of shapes and colors of the surrounding objects under a near-light speed through a first-person view.

A major highlight of this project is the transformation of complex physics theories into dynamic visual experience. Length contraction was achieved by recalculating vertices, Doppler effect was implemented by programming shader, and aberration angle was not only attempted to realize through post-processing shader, but also more accurately simulated by modifying the coordinates of the 3D model. The project fully utilized the rendering capabilities of Unity and the scalability of Blender modeling, establishing a bridge between technical and physical understanding.

Despite the project successfully simulating the effects of special relativity, there were various technical challenges during the process of development. For instance, the accuracy of post-processing simulation of aberration angle is low and it is hard to achieve a consistent distortion as the direction of the camera changes with mouse if applying dynamic lens distortion control at the initial stage while constructing the prototype.In addition, when combining the Doppler effect with length contraction and aberration angle simulations, an abnormal color shift appeared due to version incompatibility and remnants from the prototype's map model. Also, Oculus Quest holds a certain limitation for the complexity of shader in Unity , which slightly affects the smoothness of the simulation. Finally, due to a vertices budget, some of the designs were not fully retained in the final product.

This project covers a complete virtual reality development process from model construction, perspective control, shader application and vertices recalculation, enabling participants to possess a systematic practice and understanding of related technologies. During the practice of converting physical formulas into visual parameters, a deeper understanding of spacial compression and color shift involved in special relativity was gained, as well as a better technical mastery of the C# programming language and graphics rendering systems.

If the project continues to be developed in the future, several aspects can be further optimized. Firstly, more interactive features could be introduced, such as user-controlled speed parameters and additional camera trajectory options. The project undoubtedly has the potential to achieve its full value when integrated with game design principles and educational application development. By incorporating interactive elements, adjustable parameters, and narrative-driven exploration mechanisms, the simulation can gradually evolve from a technical prototype into an engaging educational tool or even an entertaining game. These extensions would not only enrich the user experience but also greatly enhance the accessibility and appeal of complex scientific concepts, especially for younger audiences or non-specialist learners. Furthermore, visualizing relativistic effects in a VR environment holds significant potential for education. It can assist students in intuitively understanding abstract concepts that are otherwise difficult to grasp through equations and textbooks.

Additional improvements may include further optimization of shader performance to

deliver a smoother and more visually compelling experience. New visual effects, such as gravitational lensing or relativistic time dilation clocks, could be implemented in future versions by extending the existing framework. A multi-camera system could be implemented to provide users with a broader perspective, allowing observation of relativistic phenomena from multiple reference frames. Code structure refactoring and GPU-based optimization could further enhance the performance, especially when deploying on standalone VR devices with limited computing power. Finally, environmental fidelity can be enhanced by improving the textures of 3D assets, including street scenes and celestial backgrounds. AI-based tools may also be introduced to automatically generate dynamic urban settings, famous landmarks from other regions, or even immersive environments inspired by movies and literature. This design approach, which is centered around the concept of immersive interaction, is also highly compatible with modern fields such as game design, virtual exhibitions, and interactive museum experiences. In conclusion, the project functions not only as a scientific stimulation, but also as potential platform that bridges technology, education and the game industry.

6 Conclusions

This project utilized virtual reality to overcome the challenge of making special relativity more comprehensible in traditional education, allowing users to actually see physical phenomena under high-speed motion. This is not only a technical practice, but also an exploration of methods to disseminate knowledge. During the developing process, theoretical formulas were no longer just theoretical derivations on paper, but are transformed into interactive and immersive visual experience, giving knowledge a tangible medium through which knowledge can be intuitively perceived. The immersive quality was particularly enhanced through the first-person camera system, which responds dynamically to user perspective and motion, thereby reinforcing the authenticity of relativistic distortions such as visual contraction and red-blue shifts.

More importantly, the project validates the bridging role of virtual reality technology and science education. As recent research suggests, virtual reality technologies are becoming increasingly important in enhancing learning experiences by providing immersive and interactive simulations [2]. The project demonstrates a new possibility that complex physical concepts do not have to exist solely in professional contexts, but can be intuitively understood and actively explored widely by people. Such a medium is especially valuable for younger audiences or learners without a scientific background, as it encourages curiosity and deeper engagement with physics concepts that are typically considered abstract or inaccessible.

In a broader perspective, this approach also reflects the emerging trend of interdisci-

plinary education, where technical tools such as game engines, 3D modeling, and virtual interfaces are increasingly applied to the communication of scientific, historical, and even artistic knowledge. The immersive aspect of VR transforms learning from passive reception to active discovery. Users are not just seeing the effects of relativity—they are feeling them, navigating through them, and constructing meaning in a personal way.

In the future, this project still has a broad space for further development, including enhancing interactivity, expanding scene expressiveness, and further integration with teaching content. As a growing project, it carries not only the world of relativity, but also the direction of future scientific communication.

References

[1] Einstein, A. (1905). *Zur Elektrodynamik bewegter Körper*. Annalen der Physik, 322(10), 891–921.

[2] Freina, L., & Ott, M. (2015). A literature review on immersive virtual reality in education: state of the art and perspectives. In *The International Scientific Conference eLearning and Software for Education* (pp. 133–141). Nicolae Balcescu Land Forces Academy.

[3] TestTubeGames. (2012). *Velocity Raptor – A Game about Special Relativity*. Retrieved from https://testtubegames.com/velocityraptor.html

[4] Unity Technologies. (2023). *Unity Manual: Post-Processing*. Retrieved from https://docs.unity3d.com/Manual/PostProcessingOverview.html

[5] Ryder, L. (1996). *Introduction to General Relativity*. Cambridge University Press.

Appendix A: Camera Movement Script

```
using UnityEngine;

public class CameraControl : MonoBehaviour
{
    public float mouseSensitivity = 100f;       // Mouse sensitivity
    public float movementSpeed = 200f;          // Camera movement speed
    public bool lockToWorldZ = true;            // Lock movement direction
        to global Z-axis
    private float xRotation = 0f;               // Vertical rotation angle
    private float yRotation = 0f;               // Horizontal rotation
        angle

    void Start()
    {
        Cursor.lockState = CursorLockMode.Locked;     // Lock the cursor
            to the center of the screen
        yRotation = transform.eulerAngles.y;          // Initialize
            horizontal angle based on current orientation
    }

    void Update()
    {
        // Mouse control for looking around
        float mouseX = Input.GetAxis("Mouse␣X") * mouseSensitivity *
            Time.deltaTime;
        float mouseY = Input.GetAxis("Mouse␣Y") * mouseSensitivity *
            Time.deltaTime;

        // Vertical rotation
        xRotation -= mouseY;
        xRotation = Mathf.Clamp(xRotation, -90f, 90f);

        // Horizontal rotation
        yRotation += mouseX;

        // Apply combined rotation to the camera
        transform.localRotation = Quaternion.Euler(xRotation, yRotation,
            0f);

        // Press E to turn around (180 degrees)
        if (Input.GetKeyDown(KeyCode.E))
```

```
        {
            yRotation += 180f;
        }

        // Forward movement logic
        if (lockToWorldZ)
        {
            // Move in a fixed global direction (aligned to Z-axis)
            Vector3 worldForward = Quaternion.Euler(0, -90f, 0) *
                Vector3.forward;
            transform.position += worldForward * movementSpeed *
                Time.deltaTime;
        }
        else
        {
            // Move along the camera's forward direction
            transform.position += transform.forward * movementSpeed *
                Time.deltaTime;
        }
    }
}
```

Code 10: Unity C# script for camera movement

Appendix B: Doppler Shift Script in Unity

```
using UnityEngine;
using UnityEngine.Rendering;
using UnityEngine.Rendering.Universal;

public class DopplerEffectPostProcessing : MonoBehaviour
{
    public Volume frontVolume; // Volume for blue shift (-X direction)
    public Volume backVolume;  // Volume for red shift (+X direction)
    public float velocity = 0.8f; // Speed as a fraction of light speed
        (v/c)
    public float blendSpeed = 5f; // Controls how smoothly the transition
        happens

    private ColorAdjustments frontColorAdjustments;
    private ColorAdjustments backColorAdjustments;
    private Transform cameraTransform;

    void Start()
    {
        cameraTransform = Camera.main.transform; // Get the main camera
            transform

        // Get Color Adjustments from Volume
        if (frontVolume.profile.TryGet(out frontColorAdjustments) &&
            backVolume.profile.TryGet(out backColorAdjustments))
        {
            Debug.Log("Color␣Adjustments␣found␣in␣both␣Volumes");
        }
        else
        {
            Debug.LogError("Missing␣Color␣Adjustments␣in␣Volume␣
                Profiles!");
        }
    }

    void Update()
    {
        if (frontColorAdjustments == null || backColorAdjustments ==
            null) return;

        // Get the camera's forward direction (only X and Z components)
```

```
Vector3 flatForward = new Vector3(cameraTransform.forward.x, 0,
    cameraTransform.forward.z).normalized;

// Reference X-axis directions
Vector3 xAxis = Vector3.right;  // +X direction (Red shift)
Vector3 negXAxis = Vector3.left; // -X direction (Blue shift)

// Compute the angle (cos$\theta$) between the camera's view
    direction and the X-axis
float cosThetaRight = Vector3.Dot(flatForward, xAxis);   //
    Facing +X
float cosThetaLeft = Vector3.Dot(flatForward, negXAxis); //
    Facing -X

// Compute Doppler shift factors
float dopplerRight = Mathf.Sqrt((1 + velocity * cosThetaRight) /
    (1 - velocity * cosThetaRight));
float dopplerLeft = Mathf.Sqrt((1 + velocity * cosThetaLeft) / (1
    - velocity * cosThetaLeft));

// Smooth Doppler blending factor based on cosTheta
float blendFactor = Mathf.Clamp01((cosThetaRight + 1f) / 2f); //
    // Normalize from -1 to 1 to 0 to 1
blendFactor = Mathf.Lerp(frontVolume.weight, blendFactor,
    Time.deltaTime * blendSpeed); // Smooth transition

// Apply volume weight gradually
frontVolume.weight = 1f - blendFactor; // More blue when facing -X
backVolume.weight = blendFactor; // More red when facing +X

// Smooth color transitions
Color targetFrontColor = new Color(1 / dopplerLeft, 1,
    dopplerLeft);
Color targetBackColor = new Color(1 / dopplerRight, 1,
    dopplerRight);

frontColorAdjustments.colorFilter.value =
    Color.Lerp(frontColorAdjustments.colorFilter.value,
    targetFrontColor, Time.deltaTime * blendSpeed);
backColorAdjustments.colorFilter.value =
    Color.Lerp(backColorAdjustments.colorFilter.value,
    targetBackColor, Time.deltaTime * blendSpeed);
```

```
        // Debugging output
        Debug.Log($"Blend Factor: {blendFactor}, Front Color (-X Blue
            Shift): {frontColorAdjustments.colorFilter.value}, Back Color
            (+X Red Shift): {backColorAdjustments.colorFilter.value}");
    }
}
```

Code 11: Unity C# script for Doppler Shift

Appendix C: Doppler Shift Script in Oculus Quest

```
using UnityEngine;
using UnityEngine.Rendering;

public class DopplerEffectPostProcessing : MonoBehaviour
{
    public Volume frontVolume; // Blue-shift volume (-X direction)
    public Volume backVolume;  // Red-shift volume (+X direction)
    public float velocity = 0.8f;     // Relative velocity ratio v/c,
       must be less than 1
    public float blendSpeed = 5f;     // Speed of blending transition
       between volumes
    public Transform xrCameraTransform;

    void Start()
    {
        // Automatically bind to Camera.main if no XR camera is set
        if (xrCameraTransform == null && Camera.main != null)
        {
            xrCameraTransform = Camera.main.transform;
            Debug.Log("XR␣camera␣automatically␣bound␣to␣Camera.main");
        }
    }

    void Update()
    {
        if (xrCameraTransform == null) return;

        Vector3 flatForward = new Vector3(xrCameraTransform.forward.x, 0,
           xrCameraTransform.forward.z).normalized;
        Vector3 xAxis = Vector3.right;
        float cosTheta = Mathf.Clamp(-Vector3.Dot(flatForward, xAxis),
           -0.999f, 0.999f); // Prevent edge case errors

        // Doppler factor (based on real physical formula)
        float dopplerFactor = Mathf.Sqrt((1 + velocity * cosTheta) / (1 -
           velocity * cosTheta));

        // Map dopplerFactor to blue/red weights:
        // Facing -X (cos$\theta$ = -1): dopplerFactor > 1 means
           blue-shift
        // Facing +X (cos$\theta$ = +1): dopplerFactor < 1 means red-shift
```

```
        // Use logarithmic scaling to enhance visual contrast
        float logDoppler = Mathf.Log(dopplerFactor); // log > 0:
            blue-shift, log < 0: red-shift
        float blend = Mathf.Clamp01((logDoppler + 1f) / 2f); // Normalize
            log scale to 0~1 range

        // Reverse the logic so that higher blue weight means stronger
            blue-shift effect
        float blueWeight = blend;
        float redWeight = 1f - blend;

        frontVolume.weight = Mathf.Lerp(frontVolume.weight, blueWeight,
            Time.deltaTime * blendSpeed);
        backVolume.weight = Mathf.Lerp(backVolume.weight, redWeight,
            Time.deltaTime * blendSpeed);

    }
}
```

Code 12: Unity C# script for Doppler Shift

Appendix D: Aberration Angle Script

```
public VolumeProfile profile; // volume profile containing LensDistortion
private LensDistortion lensDistortion;

void Start() {
    // Get lens distortion
    profile.TryGet(out lensDistortion);
}

void Update()
{
    // Get the forward direction of the camera (ignoring Y-axis)
    Vector3 flatForward = new Vector3(cameraTransform.forward.x, 0,
        cameraTransform.forward.z).normalized;
    Vector3 movementDir = Vector3.forward; // Assume motion is along
        global Z-axis

    // Calculate $\theta$ (angle between camera view and motion direction)
    float cosTheta = Vector3.Dot(flatForward, movementDir);

    // Calculate relativistically transformed $\theta$' using aberration
        formula
    float beta = velocity; // velocity as a fraction of the speed of
        light (v/c)
    float cosThetaPrime = (cosTheta + beta) / (1 + beta * cosTheta);

    // Use $\theta$' to control lens distortion (the smaller $\theta$',
        the more compressed the view)
    float aberrationFactor = Mathf.Clamp01(1f - cosThetaPrime); // Map to
        range [0, 1]
    lensDistortion.intensity.Override(aberrationFactor *
        maxDistortionValue); // Apply scaled distortion intensity
}
```

Code 13: Unity C# script for Aberration Angle