A relativistic simulator in python

Maria-Alexandra Boboaca[1,*] and *Madalina* Boca[1,**]

[1]Department of Physics, University of Bucharest, Atomistilor 405,Magurele, 077125, Ifov, Romania

Abstract. We present a 3D relativistic simulator developed using the Ursina Engine in Python, inspired by sandbox-style games. The simulator incorporates core principles of special relativity—such as time dilation, length contraction, and the relativistic Doppler effect. It is designed as a first person perspective game, where both the player and the surrounding objects can move, which allows phenomena as the Terrell rotation to be illustrated in an intuitive manner. The simulator features multiple scenarios, each of them illustrating a different example. The main elements of theory on which the simulator is based are presented in the documentation that accompanies each scenario. Our simulator can serve as an educational tool for university or high school students. The software is modular and designed for extensibility, allowing future integration of other effects. Another key aspect is the simplicity of the implementation, which facilitates extension and modification. The rich documentation that accompanies the code allows for easy understanding of both the program and the physics behind it. This also creates the possibility of using it as a template for other projects and allows users to experiment with the underlying physics by directly modifying the code. This would contribute to cementing the students' understanding of the special theory of relativity.

1 Introduction

In teaching theoretical physics, graphical simulators offer a decisive advantage as complementary tools to traditional exposition methods. In the field of theoretical physics, experiments are impractical or prohibitively expensive to stage in the classroom, and the resulting phenomena—especially in relativity and quantum mechanics—are often counterintuitive. Interactive visualizations let students manipulate parameters, observe immediate consequences, and link equations to observable behavior, thereby building intuition that is hard to achieve with formulas alone. Simulators are reproducible, low-cost, and safe, making them an effective complement to traditional lectures and problem sets while bridging the gap between mathematical derivations and physical insight.

Another advantage of a simulator is that, if it is written in a high-level programming language, it can be easily modified by the user to adjust the physics or add new features or scenarios. Experimenting with the equations and directly observing how parameter changes affect the appearance of objects is a powerful way to practice the theoretical concepts, in parallel with the traditional problem solving methods.

*e-mail: maria-alexandra.boboaca@s.unibuc.ro
**e-mail: madalina.boca@unibuc.ro

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

We present in this paper a 3D relativistic simulator in Python, written in the Ursina Engine, a Python game engine that allows for easy creation of 3D games [1]. The simulator incorporates core principles of special relativity—such as length contraction, the Doppler effect, and the Terrell rotation. The code is deliberately kept simple, making it easy to understand and modify. Likewise, the simulated systems are chosen to be sufficiently simple that the exact theory can be implemented without approximations.

Although less complex and feature-rich than other simulators already published [2, 3], our simulator is able to illustrate the main concepts of the special relativity theory, and it can be used as an educational tool for high school and university students. The program was designed in such a way that the computational resources required are minimal, so it can be run on a computer with a modest configuration; the details of the implementation are presented in the appendix A.2.

The structure of our paper is the following: In the next section we review the main elements of the special relativity theory, and in the following section we present each scenario of our simulator. In the appendix we present the details of the implementation of the simulator.

The code is available at https://github.com/madalina-boca-at-unibuc-ro/srt.

2 Theory

Here we review the main elements of the special relativity theory [4], illustrated in the simulator scenarios. The metric used throughout the paper is

$$g \equiv (+, -, -, -), \tag{1}$$

the position four-vector is $x^\mu = (ct, x, y, z)$ and the Lorentz relativistic factor is $\gamma = \frac{1}{\sqrt{1-\beta^2}}$. The Lorentz matrix of the boost transformation from the laboratory frame K to a frame K' moving with velocity $\mathbf{v} = c\boldsymbol{\beta}$ is given by:

$$x'^\mu = \Lambda^\mu{}_\nu x^\nu, \qquad \Lambda^\mu{}_\nu = \begin{bmatrix} \gamma & -\gamma\beta_x & -\gamma\beta_y & -\gamma\beta_z \\ -\gamma\beta_x & 1 + \frac{\gamma^2}{1+\gamma}\beta_x^2 & \frac{\gamma^2}{1+\gamma}\beta_x\beta_y & \frac{\gamma^2}{1+\gamma}\beta_x\beta_z \\ -\gamma\beta_y & \frac{\gamma^2}{1+\gamma}\beta_x\beta_y & 1 + \frac{\gamma^2}{1+\gamma}\beta_y^2 & \frac{\gamma^2}{1+\gamma}\beta_y\beta_z \\ -\gamma\beta_z & \frac{\gamma^2}{1+\gamma}\beta_x\beta_z & \frac{\gamma^2}{1+\gamma}\beta_y\beta_z & 1 + \frac{\gamma^2}{1+\gamma}\beta_z^2 \end{bmatrix} \tag{2}$$

with

$$x_\mu = g_{\mu\nu} x^\nu = (ct, -x, -y, -z). \tag{3}$$

The length contraction formula is given by:

$$l = \frac{l_0}{\gamma}, \tag{4}$$

and tells us that the length of an object moving with respect to an observer is contracted by a factor γ with respect to the length of the object at rest.

Another element of the special relativity theory which appears in the simulator scenarios is the velocity addition formula: If an entity e_A is moving with the velocity $\boldsymbol{\beta}_A$ with respect to the laboratory frame K, and an observer O is moving with the velocity $\boldsymbol{\beta}_O$ with respect to e_A, then the velocity $\boldsymbol{\beta}_{AO}$ of the entity with respect to the observer is given by

$$\boldsymbol{\beta}_{AO} = \frac{1}{1 - \boldsymbol{\beta}_A \cdot \boldsymbol{\beta}_O} \left[\boldsymbol{\beta}_A - \boldsymbol{\beta}_O + \frac{1}{c^2} \frac{\gamma_O}{1 + \gamma_O} \boldsymbol{\beta}_O \times (\boldsymbol{\beta}_O \times \boldsymbol{\beta}_A) \right]. \tag{5}$$

The Doppler effect is the change of the frequency perceived by an observer if the source of the signal is moving with the velocity $\boldsymbol{\beta}$ with respect to the observer. The formula for the Doppler effect is given by:

$$\lambda = \lambda_0 \gamma (1 + \boldsymbol{\beta} \cos\theta), \tag{6}$$

where λ_0 is the wavelength of the signal in the source's frame, λ is the wavelength perceived by the observer, and θ is the angle between the direction of the velocity and the source position vector relative to the observer.

Finally, the last topic covered in this section is the Terrell rotation. The topic is related to the distinction between the standard description of the transformation of the time and space under a boost transformation, and the actual change of physical appearance of an object as seen by an observer (a real person) with respect to which the object is moving.

The length contraction, discussed before, defines the length of the object as the difference between two marks of a ruler placed along the object, in contact to the two ends of the object, assuming that the marks are read at the same moment with respect to the observer's frame. Now, this is in fact not the same thing as what an observer would actually see (or record with a camera pointing at the object) when the object is moving. The difference is due to the fact that the light emitted by the two ends of the object travels at the finite speed c, and therefore they cannot be seen at the same moment. A sketch of the situation is shown in Figure 4: in panel (a) we present on observer O watching a cube of side l, as it moves along Ox with velocity v. Assume the person takes a photo at the exact moment when the corner A is "aligned" with the camera. If the speed of light is very large (infinite), then the photons which left the corners D and A at the same moment reach the camera simultaneously, and the photo looks like in the figure. In panel (b) we present the actual situation. Since the speed of light is finite, the photons which reach the camera at the same time were emitted by the corners D and A at different moments in time. The result is that the observer sees the cube as being "rotated". In this very simple geometry, the distance $|DD'|$ can be calculated by using the relations

$$\left|DD'\right| = v\,(t - t')\,, \qquad \frac{|DD'|^2 + (l + |OA|)^2}{t'} = \frac{|OA|}{t} \tag{7}$$

from which we get a quadratic equation in t' and then, using its solution, we can compute the distance $|DD'|$.

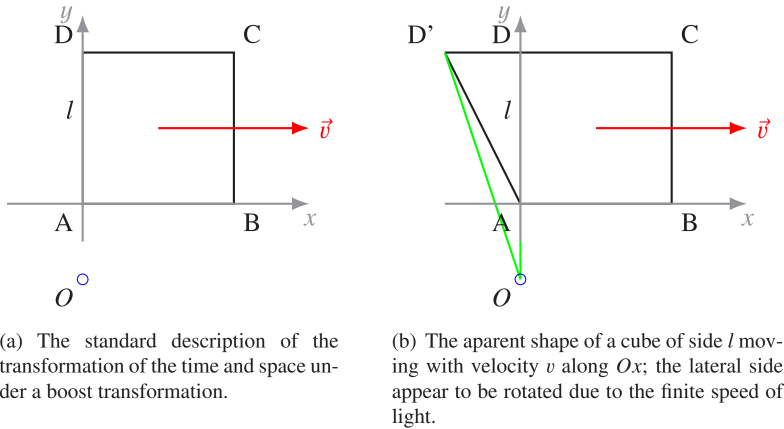
The Terrell rotation was mentioned for the first time by Lampa in 1924, and later independently by Terrell [5] and Penrose [6] in 1959. The experimental verification of the Terrell rotation was achieved by Hornof et al. [7] in 2025.

We also mention here that strictly speaking, "rotation" is not the most accurate term to describe the phenomenon, but it is the one that is most commonly used in the literature. In fact, surfaces appear distorted. In some simple cases the distortion can be expressed analytically. Let us consider, for example a plane surface, orthogonal on the Oz axis of the observer's frame, and moving with velocity v along the Oz axis. The coordinate of a point P on the surface is then given by the equation

$$\mathbf{r}_P(t) = (x, y, vt) \tag{8}$$

Assuming that the observer is in the origin of the reference frame, it will see at the time t the light which was emitted by the point P at the retarded time

$$t_r = t - \frac{|\mathbf{r}_P(t_r)|}{c} \tag{9}$$



(a) The standard description of the transformation of the time and space under a boost transformation.

(b) The aparent shape of a cube of side l moving with velocity v along Ox; the lateral side appear to be rotated due to the finite speed of light.

Figure 1. The schematic of the Terrell rotation.

solving the above equation for t_r we get the coordinates of the apparent position of the point P at the time t:

$$x_{\text{apparent}} = x_P(t_r) = x, \qquad y_{\text{apparent}} = y_P(t_r) = y,$$
$$z_{\text{apparent}} = z_P(t_r) = vt_r = \gamma^2 \left[(\beta ct) - \beta \sqrt{(\beta ct)^2 + y^2 + z^2} \right]. \tag{10}$$

The above equations are implemented, in a slightly modified version, in two scenarios based on the Terrell rotation phenomenon: `Terrell rotation (cube)` and `Terrell rotation (surface)`. The first one considers a cube, whose edges are rigid, and which "rotates", and the second one simulates a surface made of a mesh of points, and which "distorts".

In the following section we present in detail the scenarios implemented in the simulator.

3 The simulator scenarios

The simulator presented here has been developed to be used as an educational tool for high school and university students. It is designed to be used in a first person perspective, and the scenarios are designed to be interactive, allowing the user to experiment with the physics and to understand the concepts of special relativity. Each scenario has a `running_state` attribute that can be set to `True` or `False`, and it is used to control the running of the scenario; the physics takes place only when the `running_state` is `True`, and the user can toggle it by pressing the `Space` key. In all scenarios the default value of the `running_state` is `False`, i.e. the physics is stopped when the scenario is loaded.

In the following we present each scenario in detail.

3.1 "A simple moving sphere"

This scenario allows the users to see the length contraction of a sphere moving with respect to an observer at rest. The axes of the observer's frame are shown, and also the path of the sphere in the observer's frame is shown to guide the eye. The velocity and the direction of motion of the sphere can be set in the `Settings` menu. The observer can move only when

the physics is stopped, the change of position serving only to change the viewpoint of the observer in order to help the visualisation of the sphere for any direction of its velocity.

In order to implement the length contraction, we used the predefined `scale(contract,1,1)` which has the advantage of being robust and simple, but applies the contraction along one of the directions x, y, or z (in the example here, it contracts the sphere by the factor `contract` along the x-axis).

To apply a contraction along the arbitrary direction of motion, we introduce a new entity (`carrier`) whose local x-axis is aligned with the direction of motion of the sphere. The orientation of the carrier along the velocity $\mathbf{v}$ is done by using the `look_at` method of the Ursina engine. The sphere is parented to this carrier and uniformly scaled by $1/\gamma$ in the carrier's local coordinates. At each fixed time step the carrier is advanced by $v\Delta t$, where v is the velocity of the sphere and Δt is the time step. This guarantees that contraction is always applied along the velocity direction independently of the object's orientation.

To improve their understanding of the contraction, users are encouraged to play with the settings and observe the contraction in different directions, but also to edit the code to introduce features from other scenarios, such as the Doppler effect. They also can replace the sphere by another entity, such as a cube or a cylinder and experiment with the contraction procedure. A screen capture of the scenario is shown in Figure 2, for a sphere moving with the velocity $0.5c$ along the diagonal of the xOy plane.

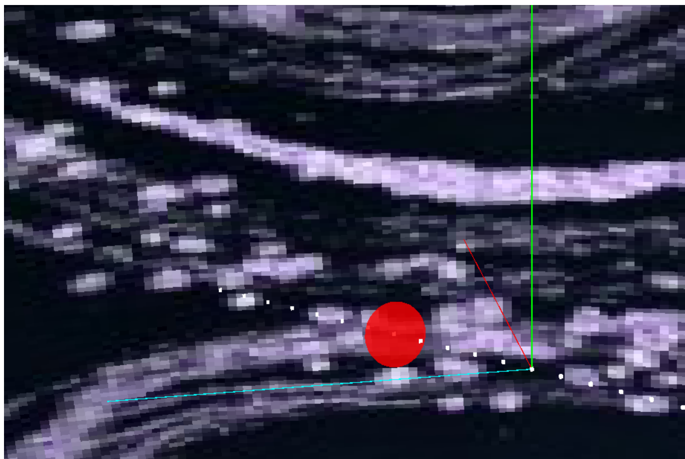


Figure 2. The length contraction of a sphere moving with the velocity $0.5c$ along the diagonal of the xOy plane with respect to an observer at rest.

3.2 "The Doppler effect"

This scenario illustrates color shifts due to the relativistic Doppler effect for a set of point like particles distributed randomly and moving with arbitrary velocity; also, when the scenario is loaded each particle gets a random "at rest" wavelength. Two point-clouds are rendered simultaneously: (i) the laboratory view (semi-transparent spheres) and (ii) the observer's view (opaque spheres), so users can compare "what the lab sees" with "what the moving observer sees". The observer `moving_state` is always `true` and the key `Space` toggles the physics (particles advance only when "running"), the key `TAB` toggles mouse lock, the keys `WASD + Shift` translate the observer, and the key `R` resets the camera. When paused, the

observer can be repositioned with a non-physical high speed ($5c$) purely as a navigation aid; when running, the observer uses the physical speed `v_0` from the settings.

In the simulation each particle from the two clouds is rendered with the color corresponding to the Doppler-shifted wavelength, as calculated with respect to the laboratory frame and, respectively, observer's frame. For the moving observer the Doppler shift is calculated using the relative velocity of each particle with respect to the observer, implemented in the `compute_relative_velocity` function. For rendering, the colors are mapped to an RGB code, as described in the appendix A. The number of particles included in each cloud can be set in the `Settings` menu, which makes the simulation more or less demanding on the computer.

A screen capture of the scenario is shown in Figure 3, for a set of 20 particles moving with the velocity $0.1c$, the observer's velocity is also set to $0.1c$; in the figure one can see the two clouds of particles, partially overlapping, but rendered with different colors. The students are

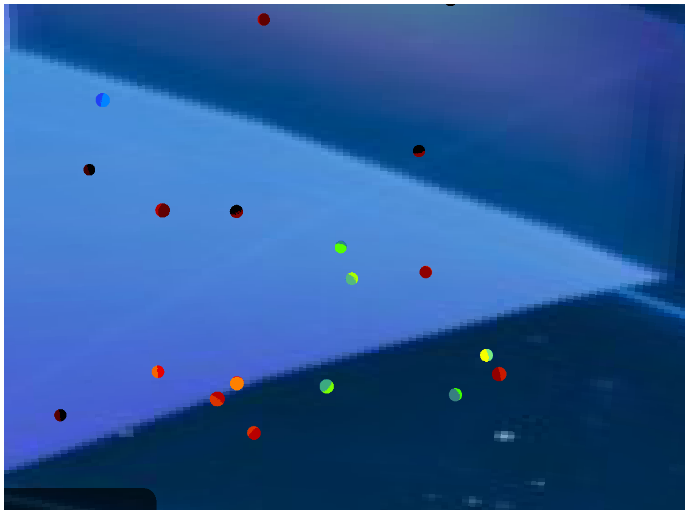


Figure 3. The Doppler effect for a set of 20 particles moving with the velocity $0.1c$ in arbitrary direction; the particles in the laboratory view are semi-transparent, and the ones in the observer's view are opaque. They appear partially overlapping, but rendered with different colors.

encouraged to play with the settings and to observe the Doppler effect for different velocities and number of particles. They can also edit the scenario, incorporating new features, such as the stellar aberration, or retarded time-geometry.

3.3 "The Terrell rotation"

The code has two scenarios based on the Terrell rotation phenomenon: `Terrell rotation (cube)` and `Terrell rotation (surface)`. The first one considers a cube, whose edges are rigid, and which "rotates", and the second one simulates a surface made of a mesh of points, and which "distorts". In both scenarios the observer is assumed at rest, and its `moving_state` is always `True` only when the physics is stopped, for repositioning purposes. The repositioning actually helps the users to see that the apparent distortion depends not only on the velocity of the surface, but also on the observer's position. As in the other scenarios, the key `Space` toggles the physics (the object advances only when "running"), the key

`TAB` toggles mouse lock, the keys `WASD + Shift` translate the observer (at the repositioning speed of $5c$), and the key `R` resets the camera.

In both scenarios it is visible the real surface, and the apparent distorted surface; in the case of the cube, the transformation (10) is implemented for its 8 corners, which are connected by rigid edges. In this case the sides of the apparent cube appear as "rotated". The second version of the scenario is implemented for a surface made of a mesh of points, and which "distorts". In this case the surface appears as a distorted plane.

Figure 4. The Terrell rotation for a surface: in white the real flat surface, moving with the velocity $0.5c$ along the x-axis, in red the apparent distorted surface.

In the figure 4 we show a screen capture of the scenario `Terrell rotation (surface)`, for a surface moving with the velocity $0.5c$ along the x-axis. The real surface is white, and the apparent distorted surface is red. When the scenario is loaded the physics is paused, and the observer can be repositioned. The surface starts moving when the physics is toggled with `Space`. The user can see that the red surface appears to start at a later time than the white surface, and that the points closer to the observer appear to be start earlier than the points farther from the observer. This leads to the distortion of the surface, which is visible in the figure.

The students are encouraged to include into the code new features, such as the Doppler effect, which would mean that different points in the mesh have different colors.

4 Conclusion

We have presented a 3D relativistic simulator developed in Python using the Ursina Engine. The simulator illustrates key phenomena of special relativity—such as length contraction, the Doppler effect, and the Terrell-Penrose rotation—through interactive scenarios.

The primary aim of the project is educational. For high school students, the simulator provides intuitive visualizations of relativistic effects that are otherwise difficult to grasp. For university students, it offers a platform to explore the underlying theory more deeply, with the possibility of extending the code to include additional effects.

The implementation is intentionally simple, requiring only basic knowledge of Python and modest computational resources. The modular design facilitates extensibility, enabling advanced users to modify the existing scenarios or create new ones. In this way, the simulator not only serves as a teaching aid but also as a framework for experimentation with the foundations of special relativity.

We hope that this tool will contribute to strengthening students' understanding of relativistic concepts, bridging the gap between abstract mathematical formalism and observable physical consequences.

A Details of the implementation

A.1 The conversion of the wavelength to the RGB code

In the illustration of the relativistic Doppler effect we need to convert the wavelength of the light emitted by each particle in the RGB code, which will give the color of the particle, as seen by the observer. Although in python there are packages available, as for example `Colour` [8], we decided to implement our own version of the conversion, maintain full control and allow customization. The algorithm is based on the method described in [9]: for a given wavelength λ we calculate the RGB code using the following formula:

$$(R,G,B) = \begin{cases} (0,0,0), & \lambda < 380 \text{ or } \lambda > 780, \\ \left(0.6 - 0.41\frac{410-\lambda}{30},\ 0,\ 0.39 + 0.6\frac{410-\lambda}{30}\right), & 380 \le \lambda \le 410, \\ \left(0.19 - 0.19\frac{440-\lambda}{30},\ 0,\ 1\right), & 410 \le \lambda \le 440, \\ \left(0,\ 1 - \frac{490-\lambda}{50},\ 1\right), & 440 \le \lambda \le 490, \\ \left(0,\ 1,\ \frac{510-\lambda}{20}\right), & 490 \le \lambda \le 510, \\ \left(1 - \frac{580-\lambda}{70},\ 1,\ 0\right), & 510 \le \lambda \le 580, \\ \left(1,\ \frac{640-\lambda}{60},\ 0\right), & 580 \le \lambda \le 640, \\ (1,0,0), & 640 \le \lambda \le 700, \\ \left(0.35 + 0.65\frac{780-\lambda}{80},\ 0,\ 0\right), & 700 \le \lambda \le 780. \end{cases} \tag{11}$$

The function `wavelength_to_rgb` is implemented in the file `lambda2rgb.py`, along with several variants on the above algorithm. Users can choose the variant, or can implement their own by editing the file.

The `Wavelength to RGB` scenario demonstrates the implementation of the algorithm in a 3D representation of the RGB color space. We represent the visible spectrum by placing 1000 particles, each corresponding to a wavelength between 380 nm and 780 nm, along a continuous curve in the 3D RGB color space. The coordinates of each particle are given by the red, green, and blue components associated with its wavelength. By changing the implementation of the `wavelength_to_rgb` function, the user can change the colors, and the shape of the curve.

A.2 The implementation of fixed-time step physics updates

In order to simulate the physics in each scenario, we need to update the physics (positions/ velocities/ colors) of each entity in the scene. The main disadvantage here is that in Ursina internal rendering framerate is not fixed, it can fluctuate depending on the (GPU/engine load, OS scheduling, etc.). Even if it is possible to implement physics updates at a variable framerate, recalculating the physics for each frame could lead to significant performance issues and it might be unnecessary, if the Ursina internal framerate gets too large. Therefore, we

Figure 5. The implementation of the `wavelength_to_rgb` function in the `Wavelength to RGB` scenario. $N = 1000$ particles, each corresponding to a wavelength between 380 nm and 780 nm, are placed at coordinates given by their red, green, and blue components.

decided to implement a fixed-time step physics update. The global variable `hz` is fixed, and the physics is updated in equal time quanta $\Delta t = 1/\mathtt{hz}$, regardless of the engine's instantaneous FPS. To implement this fixed-timestep logic, we use an accumulator that gathers the real elapsed time since the last physics update, by summing Ursina's `time.dt`. When enough time has accumulated to cover one fixed step, we perform one physics update and subtract Δt from the accumulator. The global variable `hz` can be set in the `Settings` menu; the default value is 20 Hz. This implementation allows *at most* one physics step per rendered frame, which reduces the work per frame when the render FPS if high and prevents the code stalling on slow machines. Each scenario holds a `FixedStep` instance, and the entities' `update(self)` method checks if the physics update should be done or not. This ensures that, even if the update(self) method of a entity is called automatically by Ursina, the physics calculations are only performed when the fixed-time step logic allows it. The UI and camera are, on the other hand, allowed to update every frame, which preserves the smooth feel of the simulation.

A.3 The implementation of the observer

The Observer class represents the viewpoint of the user in the simulator. It is implemented as a derived class of Ursina's Entity and acts as the reference frame that carries the camera. At initialization the camera is parented to the Observer, and the motion of the observer is inherited by the camera. The observer has `moving_state` attribute that toggles between `True` and `False`, and it is used to control the motion of the observer. If the `moving_state` is `True`, the observer moves with the camera, and if it is `False`, the observer is stationary. The motion is controlled by the `W`, `A`, `S`, `D` keys in the horizontal plane, and the `Left-Shift`, `Right-Shift` keys in the vertical plane. The speed of the observer can be set in the `Settings` menu, to a physical value less than the speed of light; the default value is 0.1 c. The program

also allows the repositioning of the observer, that can take place only when the scenario's `running_state` is False. For repositioning the observer motion is controlled by the same keys `W`, `A`, `S`, `D`, `Left-Shift`, `Right-Shift` but at the larger speed $5c$. In each scenario, pressing the TAB key switches between two modes: Camera-control mode in which the mouse is locked and hidden, and relative mouse motion rotates the observer's camera (free look), and UI-interaction mode in which the mouse is visible and free, allowing the user to interact with menus or buttons.

References

[1] Ursina Game Engine. https://www.ursinaengine.org, https://github.com/pokepetter/ursina

[2] Sherin, Z.W., Cheu, R., Tan, P., Kortemeyer, G.: Visualizing relativity: The OpenRelativity project. American Journal of Physics **84**(5), 369–374 (2016) https://doi.org/10.1119/1.4938057

[3] Nakayama, D., Oda, K.-y.: Relativity for games. Progress of Theoretical and Experimental Physics **2017**(11) (2017) https://doi.org/10.1093/ptep/ptx127

[4] Jackson, J.D.: Classical Electrodynamics, 3. ed., [nachdr.] edn. Wiley, Hoboken, NY (2009)

[5] Terrell, J.: Invisibility of the Lorentz Contraction. Physical Review **116**(4), 1041–1045 (1959) https://doi.org/10.1103/PhysRev.116.1041

[6] Penrose, R.: The apparent shape of a relativistically moving sphere. Mathematical Proceedings of the Cambridge Philosophical Society **55**(1), 137–139 (1959) https://doi.org/10.1017/S0305004100033776

[7] Hornof, D., Helm, V., Dios Rodriguez, E., Juffmann, T., Haslinger, P., Schattschneider, P.: A snapshot of relativistic motion: visualizing the terrell-penrose effect. Communications Physics **8**(1), 161 (2025) https://doi.org/10.1038/s42005-025-02003-6

[8] Colour Science. https://www.colour-science.org/, https://github.com/vaab/colour

[9] Mihai, D., Străjescu, E.: From wavelength to RGB filter. U.P.B. Sci. Bull. Series D **69**(2), 69 (2007)