Politecnico di Torino

POLITECNICO DI TORINO

College of Computer Engineering, Cinema and Mechatronics

Master's Degree in Computer Engineering

Master's Degree Thesis

AETHER

Advanced Engine for Three-dimensional High-resolution Emission Rendering

Supervisors
Prof. Bartolomeo Montrucchio
Dr. Antonio Costantino Marceddu

Candidate
Dennis Gobbi

December 2025

Abstract

This thesis presents the development of a modeling software called AETHER, designed to visualize the dynamics of the coma of a comet using the Godot Game Engine. When comets, mainly composed of ice and dust, are close enough to the Sun, they sublimate, creating two distinct phenomena: the coma (a cloud of gas and dust surrounding the nucleus) and the tail. Both of these phenomena are affected by solar gravity, radiation pressure, and solar wind.

The modeling tool aims to reproduce a numerical model of the phenomena occurring in the inner coma over a given period, compared to telescopic images, as well as their temporal evolution, with particular attention to the visual accuracy. The Godot Engine, an open source 2D and 3D game engine, has been chosen for the development because, compared to other alternatives, it is open source, lighter, and easier to use.

Key aspects of the implementation include the definition of the geometrical conditions of the apparition, the physical parameters of a comet's nucleus (orientation of the spin axis, rotation period) along with its emission regions, from which jets of dust are emitted, and the modeling of the emitted dust particles based on their physical parameters (size and density) and the forces to which they are subjected (emission velocity, solar gravity, and radiation pressure).

Several tests have been conducted by comparing the results of AETHER with telescopic images of a comet. The results show that the Godot Engine can be used to build a robust and physically accurate tool, without sacrificing performance and realism.

Being open source, under license **GPL-3.0**, this project aims to contribute to the growing field of scientific modeling.

Future works include expansion of the mathematical/physical model to also take into account precession of the spin axis and the dynamic motion of the comet along the orbit.

Contents

List of Figures

Acronyms

Symbols

2D Two-Dimensional 12–14, 16, 19
3D Three-Dimensional 11–16, 19, 21, 23, 25, 28, 34, 35, 51

A

AETHER Advanced Engine for Three-dimensional High-resolution Emission Rendering 3, 12, 21–23, 32, 37, 39, 40, 47–49
API Application Program Interface 32, 33
AU Astronomical Unit 1, 43, 44, 47–50

C

CCD Charge-Coupled Device 43
CPU Central Processing Unit 20, 39, 51
CSV Comma Separated Values 43

D

DEC Declination 45, 46

F

FOV Field of View 43

G

GLSL OpenGL Shading Language 20
GPGPU General Purpose Graphical Processing Unit 12
GPU Graphical Processing Unit 12, 18, 20, 39, 51
GUI Graphical User Interface 42

H

HDR High Dynamic Range 18
HTTP Hypertext Transfer Protocol 32

I

IDE Integrated Development Environment 19

J

JPEG Joint Photographic Experts Group 22
JPL Jet Propulsion Laboratory 32, 33, 42, 51
JSON JavaScript Object Notation 33

N

NASA National Aeronautics and Space Administration 32, 42

O

OpenGL Open Graphics Library 18

P

PA Position Angle 4–6, 44
PNG Portable Network Graphics 22

R

RA Right Ascension 45, 46
RAM Random Access Memory 20

S

STO Sun-Target-Observer 4, 6, 33, 44

U

UI User Interface 15, 17, 19, 23, 34, 36–38, 52
URL Uniform Resource Locator 32

V

VCS Version Control System 20

Chapter 1

Introduction

1.1 What is a Comet

Comets are small celestial bodies composed primarily of ice, dust, and rocky material. When a comet approaches the Sun, it begins to warm and release gases in a process known as outgassing. This activity results in the formation of a large, gravitationally unbound coma, a diffuse spherical envelope of gas and dust surrounding the nucleus of the comet, and, often, one or more tails composed of dust and ionized gas. The coma can reach sizes up to 15 times the diameter of Earth, while the tail may extend even beyond one Astronomical Unit (AU) (1 AU $\approx$ 149.597.870.700 meters).

Comets tend to have highly elliptical orbits and have a wide range of orbital periods, ranging from several years to potentially millions of years. Short-period comets originate in the Kuiper belt (or its associated scattered disk), which lies beyond Neptune's orbit, while those with a more extended period often originate in the Oort cloud, a spherical cloud of icy bodies extending from outside the Kuiper Belt to halfway to the nearest star (α-Centauri). Interstellar comets also originate outside our Solar System and pass through it on hyperbolic trajectories.

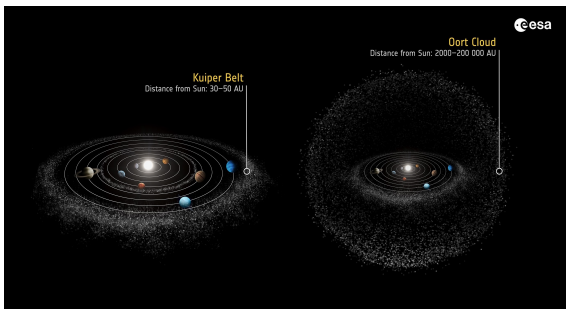


Figure 1.1: Position of the Kuiper Belt and the Oort cloud with regard to the Solar System. Image Credit: `https://www.esa.int/ESA_Multimedia/Images/2014/12/Kuiper_Belt_and_Oort_Cloud_in_context`

1.2 Description of the Modeling Approach

The modeling process is essential as it allows one to estimate the properties of the nucleus of a comet and the dust emissions from the analysis of the coma morphology. However, the modeling does not perform a direct derivation of the parameters describing the nucleus, the active regions, and the dust from the images. On the contrary, the different variables on which the model is based, which are not precisely identifiable a priori, are estimated by means of a series of trial-and-error simulations of the dust jets, after an initial input of plausible hypotheses obtained on the basis of the knowledge of the processes involved, together with a good understanding of ground-based observations and the processing techniques applied to the images.

Although in modeling there is always a certain degree of uncertainty, in this approach the estimates are not arbitrary, but based on a direct comparison between the model and the observations:

1. The entered parameters are always within a plausibility range and are subject to the modeling to invariable physical laws (e.g., the geometric conditions of observation, solar gravity, and radiation pressure).

2. Furthermore, one major strength of the model lies in the fact that these variables are closely interrelated, and a small estimation error of one or more of them has profound implications on the resulting model, as it would determine outputs that are inconsistent with what is shown by the ground-based images.

3. Lastly, the models are usually not drawn based on a single image, but derived from multiple sessions, and the results of the modeling are compared with images taken on several different dates, to verify that the parameters entered are accurate and consistent with the changes in the comet apparition occurring over time.

Some intrinsic limits are recognized on the modeling application of the inner coma structures and their extrapolation to the morphology characterization of the comet. For example, the model is not designed to describe the tail since it considers only the discrete dust emissions, which are only partially responsible for the formation of the coma (and of the tail) compared to an isotropic release of dust from the nuclear surface illuminated by the Sun. This methodology allows for the detailed analysis of the collimated structures in the inner coma; however, it does not allow for establishing their contribution to the total dust emissions.

Furthermore, the model assumes a spherical nucleus, whereas a complex three-dimensional topography of the nucleus may often itself be responsible for the development of peculiar coma morphologies, depending on the locations of the active sources.

Finally, it is recognized that some approaches, mainly based on a statistical probability, may provide multiple hypothetical solutions that do not necessarily find confirmation in a direct comparison with the observations.

1.3 Defining the Problem

The study of comets is a key area in astronomy and astrophysics, as these celestial bodies let us understand more of our Solar System. In particular, simulating the dynamics of dust and ice particles emitted by a comet over a certain period is crucial for understanding the physical and chemical processes regarding the composition of the nucleus of the comet and the interaction between the latter and the Sun.

Since these simulations rely heavily on precise physical models, the underlying software must be mathematically accurate and reliable. To our knowledge, there are only a few computer programs aimed at modeling cometary coma to estimate the main parameters of the nucleus [1]–[3].

However, these programs are generally proprietary and not available to the public. In addition, they often have some limitations, e.g., they were written in old programming languages and/or the numerical models are constrained to a two-dimensional space, meaning the dust particle trajectories lack depth and spatial realism.

These limitations highlighted the need for a new, modern software solution built with up-to-date technologies and a more intuitive, user-friendly interface. The goal was to create a tool capable of accurately simulating a fully three-dimensional environment, while remaining efficient enough to run on standard hardware without excessive computational resources.

In collaboration with a group of amateur astronomers working jointly with professional astronomers of the Osservatorio Astrofisico di Asiago, this thesis work presents the design and development of a new simulation tool, called **Advanced Engine for Three-dimensional High-resolution Emission Rendering (AETHER)**, intended to provide a more accurate, stable, and flexible environment for modeling cometary particle dynamics.

The following is a brief description of the thesis content:

- Chapter 2 illustrates the basic concepts needed to understand the following chapters.
- Chapter 3 offers a brief overview of the Godot Game Engine, which was used to develop the software.
- Chapter 4 reports the architecture and manual of the software, both from the user and developer perspectives.
- Chapter 5 contains experiments conducted on the software.
- Chapter 6 draws the conclusion and some hints about the future work.

Chapter 2

Basic Concepts

In this chapter, we will introduce the fundamental concepts in physics and astrophysics necessary to understand the following chapters.

2.1 Notions/Terms

2.1.1 Equatorial, Orbital, and Ecliptic Planes

In astronomy, defining the reference planes used to describe positions, orientations, and motions of celestial bodies is essential. Understanding the relationship between these planes is crucial for interpreting angular parameters such as the Position Angle (PA) or Sun-Target-Observer (STO).

- **Equatorial Plane:** is the plane perpendicular to the rotation axis of a celestial body. For Earth, it is aligned with its equator and is mainly used to describe the position of astronomical objects in the celestial coordinate systems (Right Ascension and Declination)
- **Orbital Plane:** is the plane in which a celestial object orbits another body. It is defined in relation to a reference plane (e.g., the Ecliptic plane) by two parameters: inclination and longitude of the ascending node.
- **Ecliptic Plane:** is the Earth's orbital plane around the Sun.

As shown in Figure 2.1, the intersection between the equatorial and ecliptic planes defines the two equinoxes and the two solstices.

2.1.2 Spin Axis and Comet Orientation

Each celestial body in the universe, including planets, stars, comets, and even whole galaxies, rotates (or spins, in scientific jargon) around an imaginary line called the axis. The orientation of this axis plays a pivotal role in a wide range of physical phenomena, such as seasonal cycles on planets, the behavior of magnetic fields, or, in our case, the emission of material from a rotating body.

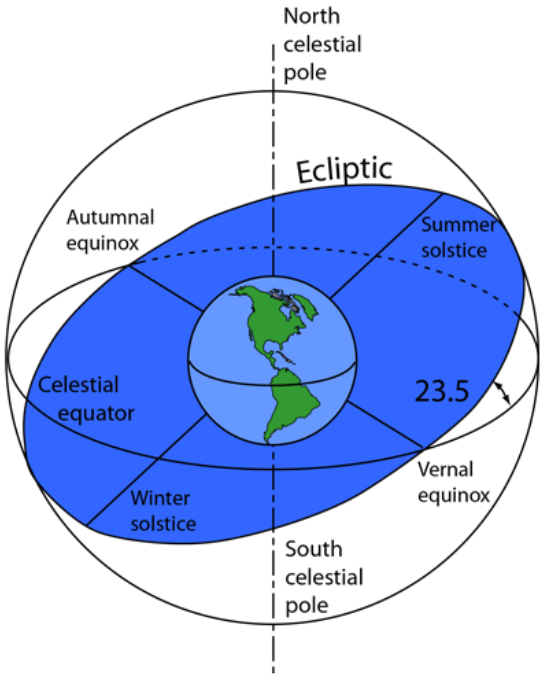


Figure 2.1: Equatorial (celestial equator) and Ecliptic planes (in blue).
Image Credit: `http://hyperphysics.phy-astr.gsu.edu/`

In the case of comets, the spin axis heavily influences the pattern and direction of dust and gas emissions. As the comet rotates, different surface regions are exposed to solar radiation at different time intervals, which in turn affasects the sublimation of ices and the formation of jets. Due to this reason, accurately modeling the orientation of the spin axis (and thus the comet) is crucial for correctly simulating trajectories of particles emitted from a comet surface.

The orientation of the spin axis of a comet in space is conventionally defined by the equatorial coordinates α and δ to which the positive (north) pole is directed. However, in our reference frame, it is intended to reproduce the nucleus and coma as seen by an Earth-based Observer (or a telescope image). The spin axis orientation is defined by two angles [4]:

- **Inclination**: Ranges from $-90°$ to $+90°$ and defines the angle between the spin axis and the sky plane as seen by the Observer. It indicates the tilt of the spin axis with respect to the Observer's line of sight.

- **Position Angle (PA)**: Ranges from $0°$ to $360°$ and defines the orientation of the projection of the spin axis onto the sky plane. It is measured counter-clockwise from the celestial north toward the east.

These two angles can be computed from the equatorial coordinates of the pole by appropriate formulas.

2.1.3 Sun Position

In addition to the comet spin axis orientation, the position of the Sun with respect to the comet also plays a pivotal role in simulating and affecting the emission of the dust particles from the comet surface and their trajectories in space. This is because jets are emitted only when emission regions are not isolated, which is determined by the solar position and orientation described by two key parameters:

- **Sun-Target-Observer (STO) Angle:** Ranges from 0° to 180° and it is the phase angle between the Sun, the comet (target), and the Observer (the Earth in our case), where 0° corresponds to a position of the Sun behind the Observer, 90° to a position lying on the sky plane and 180° to a position on the opposite side of the Observer, illuminating the hidden hemisphere of the comet nucleus.
- **PA:** Ranges from 0° to 360°, and just like the comet spin axis, PA, the Sun PA defines the direction of the Sun projected onto the sky plane as seen from Earth. It is measured counter-clockwise from the celestial north toward the east.

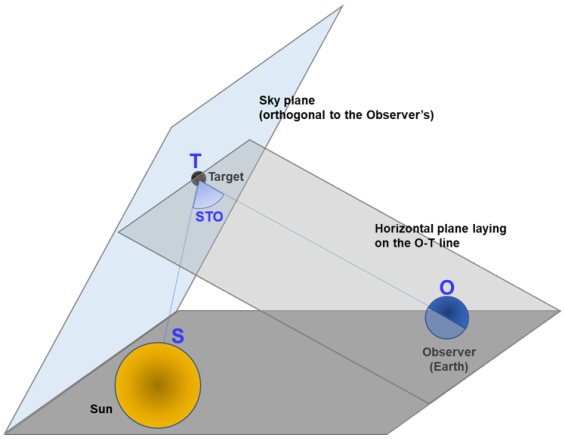


Figure 2.2: STO angle between the Sun, a generic target, and the Earth (Observer).

2.2 Orbit and Kepler Laws

The principles of orbital mechanics govern the motion of celestial bodies, including comets, around the Sun. These were first formulated empirically by Johannes Kepler in the early 17th century and later given a theoretical foundation through Newton's laws of motion and gravitation. Kepler's laws remain central to the understanding and modeling of orbital dynamics.

2.2.1 Kepler's Laws of Planetary Motion

Kepler's three laws describe the trajectories and velocity variations of orbiting bodies:

- **First Law:** The orbit of a planet is an ellipse with the Sun at one of the two foci, with the perihelion being the point where the planet is closest to the Sun, while the aphelion is the point where the planet is furthest.
- **Second Law:** A line joining the orbiting body and the Sun sweeps out equal areas during equal time intervals. This makes the orbiting body move faster when it is closer to the Sun.

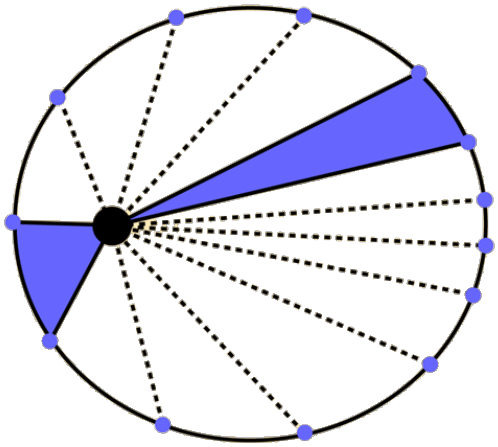


Figure 2.3: The colored areas are swept in the same time and are of the same area, illustrating how the body moves faster while closer to the Sun.
Image Credit: `http://hyperphysics.phy-astr.gsu.edu/`

- **Third Law:** The square of the orbital period T is proportional to the cube of the semi-major axis a:

$$T^2 \propto a^3$$

This law links the orbital duration with the size of the orbit, making it useful to compare orbits of different bodies.

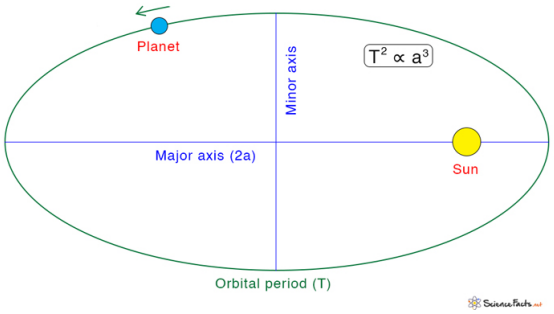


Figure 2.4: The orbital period T is proportional to half of the major axis (a).
Image Credit: `https://www.sciencefacts.net/`

2.2.2 Orbital Elements

For a complete description of an orbit, several parameters are required beyond those provided by Kepler's laws (note: each angle described below is measured

counter-clockwise):

- **Semi-Major Axis (a):** Defines the size of the orbit as half the longest diameter of the ellipse.
- **Eccentricity (e):** Describes the shape of the orbit. $e = 0$ corresponds to a circular orbit, while values approaching 1 indicate increasingly elongated ellipses.
- **Inclination (i):** The angle between the orbital plane and a chosen reference plane, typically the ecliptic.
- **Longitude of the Ascending Node (Ω):** The angle from a fixed reference direction (e.g., the vernal equinox) to the point where the orbit passes upward through the reference plane (ascending node).
- **Argument of Periapsis (ω):** The angle from the ascending node to the perihelion, measured in the orbital plane.
- **True Anomaly (ν):** The angle between the direction of perihelion and the current position of the body, as seen from the focus of the ellipse (i.e., the Sun).

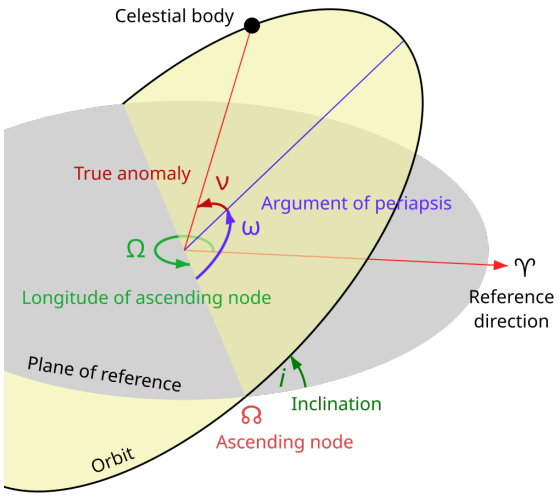


Figure 2.5: Schematic representation of the orbital elements of an elliptical orbit.
Image Credit: `https://en.wikipedia.org/wiki/Orbital_elements`

2.3 Dust Particle Position

For a correct simulation of cometary activity, it is essential to model the particle acceleration precisely. As illustrated by Jean-Baptiste Vincent in [2], the particle position at a given time t can be defined as follows:

$$x(t) = x_0 + v_{0,x}t - \frac{1}{2}a_0t^2 \tag{2.1}$$

$$y(t) = y_0 + v_{0,y}t \tag{2.2}$$

$$z(t) = z_0 + v_{0,z}t \tag{2.3}$$

Here, x_0, y_0, and z_0 represent the particle starting position (i.e., the origin) along the X, Y, and Z axis; $v_{0,x}$, $v_{0,y}$, and $v_{0,z}$ represent instead the starting *velocity* along the X, Y, and Z axis; finally, a represents the particle acceleration.

We can see that the orbital position of a particle is accelerated by the Sun only along the X axis, which is aligned with the Sun-comet direction. In contrast, on the Y and Z axes, the position is linear with regard to the velocity and time.

2.3.1 Dust Particle Acceleration

The dust particles are accelerated by gas drag from sublimating ice on the nucleus surface or from discrete active sources up to a height of approximately 10–20 times the nuclear radius, when drag vanishes. The particles are then exposed to the solar gravity and radiation pressure forces F_{grav} and F_{rad}. The two forces act in the same direction, but in the opposite sense, so that the net force is a central force that varies as $1/r^2$, implying that the trajectory of the dust particles will be a Keplerian orbit around the Sun with an effective gravitation reduced by the factor $(1 - \beta)$:

$$F_{\text{total}} = F_{\text{grav}} (1 - \beta) \tag{2.4}$$

where the β parameter is the ratio between the radiation force and the gravitational force:

$$\beta = \frac{F_{\text{rad}}}{F_{\text{grav}}} \tag{2.5}$$

The dynamics of the dust particles emitted from the comet surface are thus influenced not only by their initial emission velocity, but are mainly determined by the β coefficient, which in turn is strongly related to the dust properties, i.e., size and density.

In our modeling, we reproduce the motion of the particles in a reference frame associated with the comet nucleus. The nucleus orbits the Sun with an acceleration due to the Sun's gravitational force. The same acceleration is imparted to the dust particles separated from the nucleus. Since our models are drawn in a reference frame that moves along with the comet nucleus, it is not necessary to consider the gravitational acceleration given to the particles by the Sun when computing their motion relative to the comet nucleus. Hence, once the initial velocity value is defined, we only compute the acceleration given to the particles by the radiation pressure, which acts as a force in the Sun-comet direction.

Let $L_\odot$ be the luminosity of the Sun and r the distance between the Sun and the comet. The radiation flux created by the Sun at a distance r will be:

$$\varepsilon = \frac{L_\odot}{4\pi r^2} \tag{2.6}$$

We assume the dust particles are little spheres with a radius $R = d/2$ (where d is the particle diameter in m). The amount of radiation energy falling on a dust particle per unit of time will be:

$$\frac{\Delta\varepsilon}{\Delta t} = \varepsilon \cdot \pi \left(\frac{d^2}{4}\right) = \frac{L_\odot R^2}{4r^2} \tag{2.7}$$

The momentum given to a dust particle per unit of time due to this energy will be:

$$\frac{\Delta p}{\Delta t} = (1+\alpha)\left(\frac{1}{c}\right) \cdot \frac{\Delta\varepsilon}{\Delta t} = \frac{(1+\alpha)L_\odot R^2}{4\pi r^2 c} \tag{2.8}$$

where c is the speed of light (m/s) and α is the albedo (reflection coefficient, dimensionless) of the particle. According to the law of conservation of momentum, the change per unit of time is equal to the force acting on the particle:

$$F = \frac{\Delta p}{\Delta t} = ma \tag{2.9}$$

where m is the mass of the dust particle and a is the acceleration given to it. If the density of the cometary dust is ρ, then the mass of a particle is:

$$m = \frac{4}{3}\pi R^3 \rho \tag{2.10}$$

and the resulting acceleration is:

$$a = \frac{1}{m} \cdot \frac{\Delta p}{\Delta t} = \frac{3}{16} \cdot \frac{(1+\alpha)L_\odot}{\pi R \rho c r^2} \tag{2.11}$$

2.3.2 Equatorial and Orbital System

As the equations above are initially formulated in the orbital frame, they are not directly applicable in the equatorial coordinate system, which is more convenient for computation and input data handling.

For this reason, all particle positions and velocities were initially computed in the equatorial frame. The resulting vectors were then transformed into the orbital frame using a coordinate transformation using the orbital basis matrix. This approach ensures consistency with the physical orientation of the comet orbit.

2.4 Representing Orientation: Euler Angles and Quaternions

To precisely describe the orientation of a celestial body like a comet, we must define its rotation in a Three-Dimensional (3D) space. While several mathematical tools exist for this, Euler angles and quaternions are the most common, each with distinct advantages and disadvantages.

2.4.1 Euler Angles

Euler angles represent a 3D orientation as a sequence of three elemental rotations around the axes of a coordinate system. A common convention is yaw, pitch, and roll, corresponding to rotations around the Z, Y, and X axes.

The primary characteristic of Euler angles is that the final orientation critically depends on the sequence in which the rotations are applied. A rotation of α degrees around the X-axis followed by β degrees around the Y-axis results in a different final orientation than performing the Y-axis rotation before the X-axis one. This sequence-dependence is a fundamental aspect of 3D rotations, which are not commutative.

While intuitive, Euler angles suffer from a critical flaw known as **gimbal lock**. This occurs when a rotation aligns two of the three axes, causing the system to lose one degree of rotational freedom. This can lead to unpredictable behavior and numerical instability in simulations.

2.4.2 Quaternions

Quaternions provide a more robust and computationally stable method for representing 3D rotations, effectively avoiding the issues inherent in Euler angles. They are a mathematical extension of complex numbers, composed of four components: one scalar (real) part and three vector (imaginary) parts. A quaternion q is expressed as:

$$q = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k} \tag{2.12}$$

where w, x, y, and z are real numbers.

We use unit quaternions to represent orientations, where the magnitude is one. The key intuition is that a quaternion encodes a rotation not as a sequence of turns, but as a single rotation around a specific axis in space. The vector part (x, y, z) defines this axis of rotation, while the scalar part (w) is related to the magnitude of the rotation angle around that axis.

The advantages of using quaternions are significant:

- **Avoidance of Gimbal Lock:** Because a quaternion represents an orientation as a single axis-angle rotation rather than sequential rotations, it is immune to gimbal lock.

- **Computational Efficiency:** Composing rotations (i.e., applying one after another) is computationally faster and more numerically stable with quaternions than with matrix multiplication derived from Euler angles.
- **Smooth Interpolation:** Quaternions excel at interpolating between two orientations, a process known as "slerp" (spherical linear interpolation). This produces smooth and natural-looking rotations, which are crucial for animations and continuous simulations.

Due to their reliability and efficiency, quaternions are the preferred method for handling 3D rotations in computationally intensive fields like computer graphics, robotics, and astrophysics. In **AETHER**, both Euler angles and quaternions are used to handle rotations.

2.4.3 Fundamental 3D Graphics Concepts

To understand how a simulation tool like **AETHER** visualizes celestial bodies and their dynamics, it is essential to grasp the basic components of 3D graphics. These concepts form the building blocks for creating and rendering any three-dimensional scene.

- **Mesh, Vertices, Edges, and Faces:** At its core, a 3D model is a *polygonal mesh*, which is a collection of points in 3D space called **vertices**. When two vertices are connected, they form an **edge**. Three or more edges that form a closed loop create a **face** or **polygon**. The arrangement and density of these vertices, edges, and faces define the shape and surface of a 3D object.
- **Materials and Textures:** A mesh on its own is just a colorless wireframe. To give it a realistic appearance, we apply **materials** and **textures**. A **material** defines the physical properties of a surface, such as its color, shininess, and transparency. A **texture**, on the other hand, is a Two-Dimensional (2D) image that is wrapped around the 3D model to add surface detail, such as patterns, bumps, or intricate designs. While a texture is an image, a material determines how that texture interacts with light.
- **Shaders:** Small programs that run on the Graphical Processing Unit (GPU) and determine the final appearance of a pixel. They take the geometry, materials, textures, and lighting information from the mesh as input to calculate the color, brightness, and other visual properties of each pixel on the surface of an object. There are different types of **shaders**, such as **vertex shaders** that manipulate the position of vertices, **fragment (or pixel) shaders** that compute the color of individual pixels, and **compute shaders** to perform General Purpose Graphical Processing Unit (GPGPU) programming.
- **Lighting:** A crucial element that brings a 3D scene to life. It involves placing virtual light sources in the scene that illuminate the objects. The way light interacts with the materials of the objects, creating highlights and shadows, adds depth, realism, and mood to the final image.

- **Camera:** In a 3D environment, the **camera** is the virtual viewpoint from which the scene is rendered. It determines the perspective, field of view, and what part of the 3D world is visible to the Observer. The final 2D image is a snapshot of the 3D scene taken from the position and orientation of the **camera**.

Chapter 3

Godot

This chapter outlines the main features and structure of the Godot Engine, the platform on which the software is built. It also explains its relevance to the development of the project.

3.1 Overview

Godot is an open-source game engine developed for 2D and 3D interactive applications. Distributed under the MIT License, it has gained traction due to its lightweight framework, ease of use, and growing developer community. Initially developed by Juan Linietsky and Ariel Manzur, Godot supports deployment across various platforms, including Windows, Linux, macOS, Android, iOS, and HTML5.

While its primary purpose is game development, Godot is increasingly used in domains such as education, engineering, and scientific visualization, owing to its scripting capabilities, real-time rendering, and cross-platform functionality.

3.2 Architecture and Node System

Central to Godot's architecture is its hierarchical **scene-tree system**, where all content is organized as a tree of *nodes*. Every component, be it visual, physical, auditory, or script-based, is a node, enabling modular construction and composition.

A **scene** in Godot is a structured collection of nodes. Scenes can be embedded as nodes within other scenes, promoting reuse and encapsulation. For example, a "Spaceship" scene (as illustrated in Figure 3.1) containing mesh, collision, and animation nodes can be instanced multiple times with ease within a larger "Battlefield" scene.

3.2.1 Node Categories

All nodes derive from the base class `Node` and fall into three main categories:

Figure 3.1: An example of a scene (the battlefield) containing multiple instances of the same Battleship scene. The red circle represents the Node3D node, which is usually used as a root node for a 3D scene. Adding another battleship is as easy as dragging the scene into the battlefield one.

- **3D Nodes** (`Node3D`): For 3D applications. These nodes operate in 3D space and include types such as `Sprite2D`, `AnimatedSprite2D`, `Area2D`, `CollisionShape2D`, and `Camera2D`.
- **3D Nodes** (`Node3D`): Used for 3D content. These nodes support 3D spatial transformations and include `MeshInstance3D`, `Light3D`, `Camera3D`, `Area3D`, and `RigidBody3D`.
- **User Interface (UI) Nodes** (`Control`): Designed for user interfaces. These nodes render in screen space and include `Button`, `Label`, `Panel`, `TextEdit`, and `Container`. UInodes support layout management and anchoring.

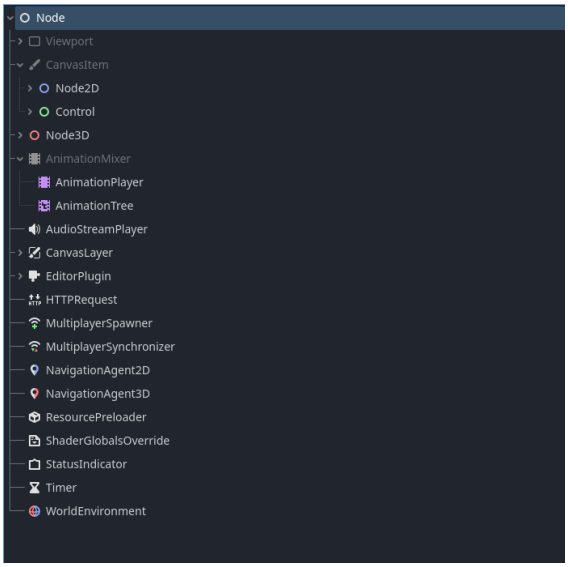


Figure 3.2: All of the top-level nodes available in Godot. Each of these (base) nodes has many children available with additional features. We can also see that each node derives/extends the basic node Node.

3.2.2 Transform2D and Transform3D

Godot provides dedicated classes for handling spatial transformations:

- **Transform2D**: Used by `Node2D` objects, it represents position, rotation, and scale in 2D space. It consists of a 2x2 `Basis` matrix encoding rotation and scale, and a `Vector2` origin for translation. Typical operations include translating, rotating, scaling, and combining transforms.
- **Transform3D**: Used by `Node3D` objects, it represents 3D transformations. As mentioned earlier, it includes a 3x3 `Basis` matrix for rotation and scale, and a `Vector3` origin for translation. It supports chaining, inversion, and conversion to other representations like quaternions or Euler angles.

3.2.3 Rotations in 3D Space

Godot provides a comprehensive system for handling rotations in 3D space, which is crucial for simulations and visualizations. The `Node3D` class, the base for all 3D objects, inherently supports multiple representations for orientation. Rotations are managed through the `transform` property, which is a `Transform3D` object containing a `Basis` and an `origin`. The `Basis` is a 3x3 matrix that represents rotation, scale, and shear, while the `origin` is a `Vector3` for translation.

Godot offers direct support for:

- **Euler Angles**: These are exposed in the inspector of the editor as the "rotation" property for user-friendliness, allowing for intuitive adjustments along the X, Y, and Z axes. However, they are generally not recommended for representing complex rotations due to issues like gimbal lock.
- **Quaternions**: For more robust and stable rotations, Godot provides the `Quaternion` data type. Quaternions are well-suited for interpolating between orientations and avoiding the pitfalls of Euler angles. They can be created from and converted to other rotation representations.
- **Basis Vectors**: The `Basis` itself, composed of three orthogonal vectors, defines the local coordinate system of the node. Manipulating the basis directly allows for precise control over the orientation of the object.

3.2.4 Scene Composition and Instancing

Scenes can be saved independently and instanced within other scenes. This system supports:

- Encapsulation of logic and data.
- Reuse across multiple contexts.
- Runtime instancing and manipulation.

Godot provides two inheritance models:

- **Class inheritance**: Script or node inherits from another class.
- **Scene inheritance**: A new scene extends an existing one.

3.2.5 Signals and Communication

Inter-node communication is handled via **signals** (Figure 3.3), an event system that enables decoupled interaction. A node emits a signal that other nodes can connect to and handle independently.

For instance, a `Button` can emit a `pressed()` signal that triggers a function in another node, such as opening a menu.

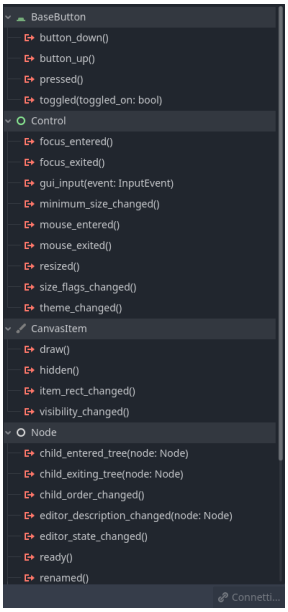


Figure 3.3: Some of the signals available for the Button node, one of the most common UI/Control nodes. We can see that some of the signals are inherited from its parent nodes (*Control* > *CanvasItem* > *Node)*.

3.2.6 Execution Lifecycle

Godot's execution model is structured around callback functions:

- `_ready()`: Triggered when the node enters the scene tree.
- `_process(delta)`: Called every frame (non-physics).
- `_physics_process(delta)`: Called at a fixed timestep, usually at a rate of 60 frames per second. Useful to handle physics behavior.
- `_input(event)`: Handles input events.

These functions are defined in scripts to control runtime behavior.

3.2.7 Advantages of the Node System

The node architecture offers:

- **Clarity**: Clear hierarchy and single-responsibility nodes.
- **Reusability**: Scenes and nodes can be reused across projects.
- **Modularity**: Complex systems built from simpler components.
- **Flexibility**: Runtime creation and modification of nodes.

3.3 Scripting with GDScript

Godot's primary scripting language is **GDScript**, a high-level, dynamically typed language with Python-like syntax. It is optimized for engine integration and rapid development.

Additional language support includes:

- **C#** (via Mono).
- **C++** (via GDNative or custom modules).
- **Python**, **Rust**, and others (via community bindings).

3.4 Rendering and Performance

Godot 4 provides three rendering backends aimed at different hardware and performance requirements. The renderer can be chosen when creating a project or changed later in the project settings.

- **Forward+** — The default for desktop platforms. This most advanced renderer uses Vulkan, Direct3D 12, or Metal through the `RenderingDevice` backend. It uses clustered forward rendering to manage dynamic lights efficiently and supports features like real-time global illumination, screen-space reflections, volumetric fog, subsurface scattering, and High Dynamic Range (HDR). It is designed for modern GPUs.
- **Mobile** — Used by default on mobile devices but also works on desktop. Like Forward+, it uses Vulkan, Direct3D 12, or Metal via the `RenderingDevice` backend, but has fewer features. It is faster for simple scenes and better suited to lower-power hardware.
- **Compatibility** (Open Graphics Library (OpenGL) Compatibility) — The fallback for low-end desktop, mobile, and web platforms. This renderer uses OpenGL and has the fewest features. It is the default on web builds and works well on older or restricted systems where maximum portability is more important than visual quality.

3.5 Physics Engine

Godot integrates both 2D and 3D physics engines, with support for:

- Rigid and kinematic bodies.
- Collision detection (shapes and areas).
- Raycasting and overlap checks.

Deterministic stepping is available for simulations requiring reproducibility.

3.6 Editor and Tooling

The Godot editor is a cross-platform, self-contained development environment implemented using the engine. It provides an integrated interface for scene construction, scripting, animation, debugging, and project deployment/export.

3.6.1 Scene Editing

The editor includes dedicated views for 2D, 3D, and UI composition. Nodes can be created, arranged, and grouped hierarchically via a visual interface. Transformations such as translation, rotation, and scaling can also be manipulated directly in the scene viewport. Each workspace includes tools appropriate to its dimensional context, with overlays for collision shapes, anchors, and other runtime markers/-gizmos.

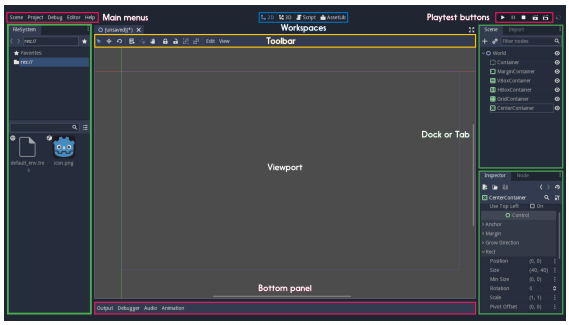


Figure 3.4: Overview of the editor. In green, we can see various decks/tabs that can be used to access and manipulate different parts of the editor.
Image Credit: `https://docs.godotengine.org/`

3.6.2 Scripting Environment

Godot provides an integrated code editor with support for GDScript. Features include syntax highlighting, autocompletion, inline error feedback, and access to in-engine documentation. Navigation tools such as symbol lookup and go-to-definition are also supported. Besides the integrated code editor, plugins are also available for many Integrated Development Environment (IDE)s and text editors, such as JetBrains IDEs and Visual Studio Code.

3.6.3 Debugging and Profiling

The engine includes runtime debugging tools that are accessible during execution. Functionality includes:

- Breakpoint-based debugging with call stack and variable inspection.
- Live scene tree inspection and property editing.
- Object memory usage and reference tracking.
- Frame profiler for Central Processing Unit (CPU) and GPU performance analysis.

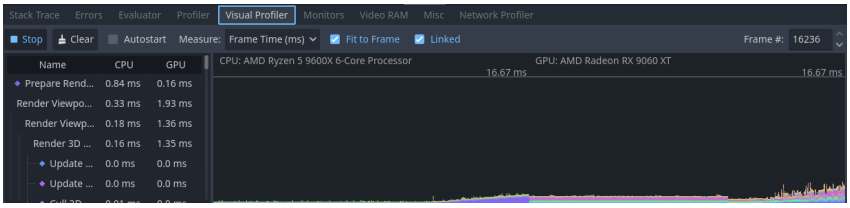


Figure 3.5: Example of the built-in visual profiler in Godot. Other profiling tabs can be used, such as "Profiler" or "Video Random Access Memory (RAM)", to assess the performance of the application better.

3.6.4 Shader

A visual shader editor is included to create vertex and fragment shaders via a node-based interface. It is intended for users without shader programming experience, though raw shader code can be written when needed. The shader language is based on OpenGL Shading Language (GLSL), with some engine-specific extensions.

3.6.5 Project Configuration and Exporting

The editor provides interfaces for setting up input maps, rendering parameters, language localization, and platform-specific behavior. Using configurable templates, the applications can be exported to desktop, mobile, and web platforms.

3.6.6 Version Control Compatibility

Godot projects are composed of plain-text files, including scenes (`.tscn`), scripts (`.gd`, `.cs`), and project metadata. This format facilitates use with version control systems such as Git. The editor does not include built-in Version Control System (VCS) tools, though third-party plugins exist for basic Git integration, such as built-in Visual Studio Code's.

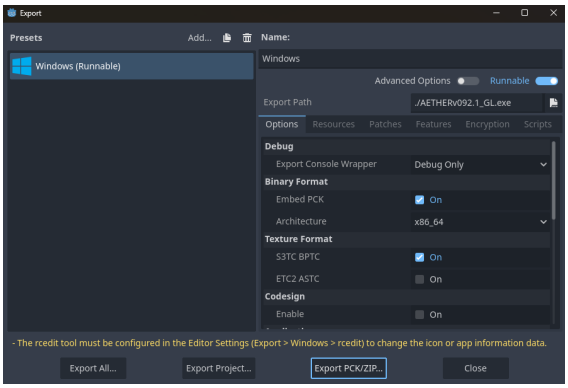


Figure 3.6: Export tab inside the Godot Editor. Can be accessed through Project>Export in the Editor. New export templates can be easily added with the Add button.

3.6.7 Live Editing and Hot Reloading

During execution, the editor supports live property editing and scene updates. Scripts and resources can be modified and reloaded at runtime, reducing iteration times during testing and development.

Godot's tooling emphasizes accessibility, self-containment, and rapid iteration. While its main focus is game development, the flexibility of the editor supports a wider use in simulation, education, and custom tool development.

3.7 Limitations

Notable constraints include:

- Lower performance than Unity or Unreal Engine for high-end 3D workloads.
- Limited ecosystem for non-game plugins and scientific tooling.

3.8 Conclusion

Godot provides a versatile platform for building interactive applications, including tools for scientific visualization. Its scripting flexibility, real-time rendering, and modular architecture make it a strong candidate for developing educational and research-focused software. In this project, Godot forms the core of the **AETHER** modeling environment, enabling real-time simulation of cometary dust emissions with improved flexibility and fidelity over prior approaches.

Chapter 4

AETHER

This chapter presents the design and implementation of **AETHER (Advanced Engine for Three-dimensional High-resolution Emission Rendering)**.

The following sections describe the overall architecture of the application, the main software components, and the user interface. Particular attention is given to the rationale behind architectural choices, algorithm implementations, and the optimizations implemented to ensure acceptable performance during execution.

Alternative solutions and design approaches are discussed and compared with the final implementation for core components and algorithms. Benchmarks and technical evaluations are included to illustrate the efficiency of the adopted methods.

4.1 Architecture

4.1.1 Code Structure

As shown in Figure 4.1, the project follows a clear folder hierarchy to separate assets, gameplay scenes, scripts, and shaders. This organization supports maintainability, reusability, and ease of navigation during development:

- **asset**: Stores texture files and graphical elements used for the user interface in both Portable Network Graphics (PNG) and Joint Photographic Experts Group (JPEG) format.
- **scenes**: Includes all Godot scene files (`.tscn`) that define levels, user interface layouts, and reusable scene components.
- **scripts**: Stores the GDScript (`.gd`) and other supported language scripts that define the behavior and logic of the application.
- **shaders**: Contains custom shader programs for the comet.

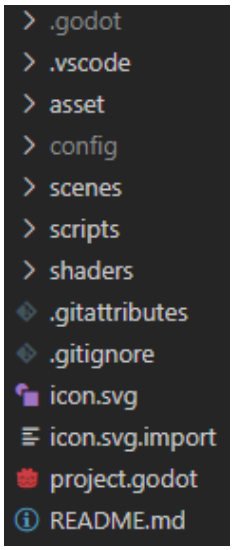


Figure 4.1: Folder hierarchy of the project within the Godot editor. Additional autogenerated folders and files are also present.

4.1.2 User Interactions

The user interface predominantly comprises buttons and sliders, the main tools for performing actions and inputting parameter values. Specifically:

- **Buttons:** Provide direct control over core functions such as starting, pausing, and stopping simulations; creating and managing configurations; opening file dialogs for saving and loading configurations; saving simulation output; and other key actions.
- **Sliders:** Sliders allow users to input numerical values or adjust simulation parameters dynamically. They enable fine-grained control over different parameters.
- **File Picker:** Integrated, platform-agnostic, file picker dialogs facilitate saving, loading input files, and saving output data, supporting smooth workflow management.
- **Mouse Controls:** In addition to the UI elements, mouse input is used for 3D navigation. Users can rotate, pan, and zoom the simulation view, allowing detailed exploration of the emission regions.

4.2 Developer Manual

4.2.1 Inter-Module Communication

The architecture of **AETHER** promotes modularity and maintainability by ensuring that different components remain as loosely coupled as possible. This design principle allows individual modules to be developed, tested, and modified independently without causing ripple effects throughout the codebase.

Signals

To achieve this, communication between modules primarily relies on using *signals*, a messaging system provided by the Godot Engine. Signals enable one module to emit events that other modules can listen for and respond to, without requiring direct references or tight dependencies between them.

Godot's signals are an implementation of the well-known **Observer Pattern** (Figure 4.2), where objects (observers) subscribe to and react upon events emitted by other objects (subjects), promoting a clean separation of concerns and loose coupling.

Godot provides many **built-in signals** for its nodes, such as `button_pressed` for Button nodes, allowing interaction with standard engine components. Moreover, signals can be connected programmatically or through the Godot editor's interface, where developers can visually link signals from one node to methods in another, thereby simplifying event-driven programming without the need for manual connection code.

For example, a module can define and emit a signal like this:

```
# simulation.gd
signal simulation_started

func start_simulation():
    emit_signal("simulation_started")
```

Another module can connect to this signal and react accordingly:

```
# ui.gd
# ready is called when a Node is added to the
   Scene Tree
func _ready():
    var sim = get_node("/root/Simulation")
    sim.connect("simulation_started", self,
       "_on_simulation_started")

# this method is called when the
   simulation_started signal is emitted
func _on_simulation_started():
    print("Simulation␣has␣started!")
```

Groups

In addition to signals, the application utilizes Godot's `SceneTree.call_group` method to broadcast method calls to all nodes belonging to a specific group.

Groups in Godot are named collections of nodes that allow the engine to treat multiple nodes as a single entity for messaging purposes. Nodes can be dynamically added to or removed from groups, enabling flexible communication without maintaining explicit references to each node.

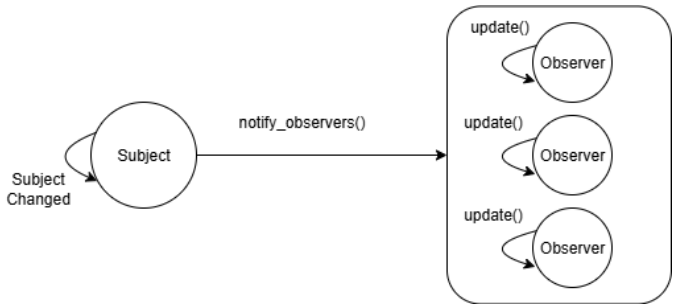


Figure 4.2: Diagram illustrating the Observer Pattern: the subject (signal emitter) notifies multiple observers (signal listeners) about events, enabling decoupled communication.

For example, to use groups:

```
# comet.gd
# Adding node to group
func create_emitter():
    # creating an emitter
    emitter.add_to_group("emitter")

func update_emitter(params):
    get_tree().call_group("emitter","update_emitter",params)
```

Each node in the `emitter` group should implement the `update_emitter` method:

```
# emitter.gd
func update_emitter(params):
    # Update the emitter based on params
```

Conclusion

This combination of signal-based and group-based communication results in:

- **Decoupled modules:** Modules do not need to know internal details about each other, only the signals they emit or listen to, or the groups to which nodes belong.
- **Improved scalability:** New features or components can be integrated by connecting to existing signals, creating new ones, or adding nodes to appropriate groups, minimizing changes to existing code.
- **Enhanced testability:** Modules can be tested in isolation by simulating signals or group calls without requiring the entire system to be active.

4.2.2 Comet module

One of the most important modules, the `comet_mesh.gd` script defines the `Comet` class, which extends `MeshInstance3D`, a node used to display 3D geometry, and acts as the core controller for simulating the geometry, orientation, and rotation of the comet, and particle emission.

_ready()

At initialization, the `_ready()` function creates axis indicators aligned to the mesh of the comet, sets the initial rotation, and updates other components:

```
func _ready() -> void:
    var _x_axis := axis_scene.instantiate() as AxisArrow
    add_child(_x_axis)
    _x_axis.set_axis_type(AxisArrow.AXIS_TYPE.X)
    _x_axis.set_height(mesh.height)
    ...
    starting_rotation = rotation
    update_comet_orientation()
```

_process()

The simulation advances in the `_process()` loop when the animation state is active, calling the `tick()` function to spawn particles and rotate the comet:

```
func _process(_delta: float) -> void:
    match animation_state:
        ANIMATION_STATE.STARTED, ANIMATION_STATE.RESUMED:
            for _i in speed_sim:
                tick(step_counter)
                n_steps -= 1
                step_counter += 1
            total_sim_time += _delta
```

Simulation

The `tick()` method is responsible for iterating emitters and applying rotation:

```
func tick(n_iteration: int) -> void:
    for emitter: Emitter in get_tree().get_nodes_in_group("emitter"):
        emitter.tick_optimized(n_iteration)
    animation_slider.tick()
    rotate_object_local(Vector3.UP, deg_to_rad(angle_per_step))
```

For faster computations, `instant_simulation()` spawns all particles at once:

```
func instant_simulation() -> void:
    simulation_setup()
    for emitter: Emitter in get_tree().get_nodes_in_group("emitter"):
        emitter.instant_simulation(n_steps, angle_per_step)
    animation_slider.instant_simulation()
```

Before running a simulation, `simulation_setup()` aligns the comet to the Sun, normalizes the transform, stores orbital transformation data, and calculates the total number of steps:

```
func simulation_setup() -> void:
    Util.equatorial_rotation = quaternion
    look_at(Util.sun_direction_vector, Vector3.UP)
    transform.basis = transform.basis.orthonormalized()
    n_steps = int(num_rotation * frequency * 60 / jet_rate)
    angle_per_step = 1.0 / (frequency * 60.0 / jet_rate) * 360.0
```

The animation control functions `animation_started()`, `animation_paused()`, and `animation_stopped()` manage playback, pausing, and cleanup.

Comet Position

The spatial orientation of the comet is handled by `point_y_axis_toward()`, which aligns the local Y-axis of the comet toward a given target vector:

```
func point_y_axis_toward(target_position: Vector3) -> void:
    var direction := (target_position - global_transform.origin)
    var target_quat := Quaternion(Vector3.UP, direction.normalized())
    quaternion = target_quat
```

Finally, `update_comet_orientation()` computes the correct orientation from astronomical parameters (direction, inclination) and applies it by orienting the Y-axis of the comet toward the direction vector:

```
func update_comet_orientation() -> void:
    var azimuth_rad := deg_to_rad(Util.comet_direction)
    var inclination_rad := deg_to_rad(-Util.comet_inclination - 90)
    var x := sin(inclination_rad) * sin(azimuth_rad)
    var y := cos(inclination_rad)
    var z := sin(inclination_rad) * cos(azimuth_rad)
    var direction := Vector3(x, y, z).normalized()
    point_y_axis_toward(global_transform.origin + direction)
```

4.2.3 Emitter module

Another key module is the Emitter, which manages the behavior of a single jet of particles originating from the surface of the comet. It controls the properties of the jets, such as emission speed and diffusion, calculates particle trajectories under solar radiation pressure, and handles the efficient rendering of thousands of particles using a `MultiMeshInstance3D`.

_ready()

Upon entering the scene tree, the `_ready()` function initializes the core components of the emitter. It sets up the material properties for the particles and prepares the `MultiMeshInstance3D` for efficient rendering via the `init_multimesh()` method. It converts the user-defined latitude and longitude into a 3D direction vector, computing the initial emission normal of the jets. It also computes the particle acceleration based on physical properties as described in the Equation 2.11.

```
func _ready() -> void:
    ...
    var lat_rad := deg_to_rad(latitude)
    var lon_rad := deg_to_rad(longitude)

    initial_norm = Vector3(
        cos(lat_rad) * cos(lon_rad) * 5,
        sin(lat_rad) * 5,
        cos(lat_rad) * sin(lon_rad) * 5
    ).normalized()
    norm = initial_norm

    init_multimesh(mm_emitter)
    add_child(mm_emitter)
    ...
    update_acceleration()
```

_process()

In the `_process()` loop, it simply performs a check each frame to check if the emitter is lit so that it can change its color accordingly.

```
func _process(_delta: float) -> void:
    is_lit = is_lit_math()
    var material = particle_mesh.get_surface_override_material(0)
    if is_lit:
        material.albedo_color = color
    else:
        material.albedo_color = color.darkened(0.5)
```

Illumination and Activity

Particle emission is conditional on whether the jet is illuminated by the Sun. Based on the underlying premise that the comet nucleus is a perfect sphere, the problem of determining illumination is greatly simplified, as there is no complex self-shadowing from surface topography to consider. Therefore, the `is_lit_math()` function can perform this check efficiently by calculating the dot product between the Sun's

direction vector and the current global normal of the jets. A positive result indicates that the angle between these two vectors is less than 90 degrees, meaning the emitter is on the sun-facing hemisphere and can actively release particles.

```
func is_lit_math() -> bool:
    var comet_basis: Basis = get_parent().global_transform.basis
    var global_space: Vector3 = (comet_basis * norm).normalized()
    global_space = global_space.rotated(
            Vector3.LEFT,
            deg_to_rad(Util.sun_direction + 90))

    var result: float = (Util.sun_direction_vector).dot(global_space)
    result = (-Util.sun_direction_vector).dot(norm)
    is_lit = result > 0
    return is_lit
```

Simulation Modes

The emitter supports two primary simulation modes: a step-by-step, real-time mode and a pre-calculated, instantaneous mode.

The real-time simulation is driven by the `tick_optimized()` function, which is called for each step of the simulation. It first updates the positions of all existing particles by calling `_accelerate_particle()` and then spawns a new particle with `_spawn_particle()` if the emitter is currently illuminated.

```
func tick_optimized(_n_iteration: int) -> void:
    # moving each particle
    for i in range(0,
        mm_emitter.multimesh.visible_instance_count, Util.n_points + 1):

        _accelerate_particle(i)
        _generate_diffusion_particles(i)

    if is_lit_math():
        var last_id := mm_emitter.multimesh.visible_instance_count + 1
        if last_id < mm_emitter.multimesh.instance_count:
            mm_emitter.multimesh.visible_instance_count =
                last_id + Util.n_points

        _spawn_particle(last_id)
    update_norm()
```

For rapid visualization of the entire coma, the `instant_simulation()` function computes the final positions of all particles emitted over a given number of steps. It iterates through each potential step, checks for illumination, and calculates the final transform for each spawned particle. The resulting data is aggregated into a buffer

and sent to the `MultiMesh` simultaneously, providing a significant performance boost.

```
func instant_simulation(_n_steps: int, _angle_per_step: float) -> void:
    ...
    var particle_transforms: Array[Transform3D] = []
    var mm_buffer: PackedFloat32Array = PackedFloat32Array()

    for i in range(_n_steps):
        ...
        if not is_lit_math2(i, _angle_per_step, _normal):
            continue

        var ith_transform := _accelerate_particle2(_n_steps - i, _normal)
        particle_transforms.append(ith_transform)

        if diffusion <= 0:
            _append_data_to_mm_buffer(mm_buffer, ith_transform, color)
    ...
    mm_emitter.multimesh.set_buffer(mm_buffer)
```

Particle Physics and Trajectory

The core of the particle trajectory simulation resides in the acceleration calculation and its application over time. The `update_acceleration()` function computes the acceleration imparted by solar radiation pressure using a standard physics formula that incorporates the Sun's luminosity, the distance of the comet from the Sun, and properties of the particle itself (albedo, diameter, and density).

```
func update_acceleration() -> void:
    # Ls / 4PI * c *(AU*sun_comet_distance)^2
    var eps: float = Util.SUN_LUMINOSITY /
      ((4 * PI) *
           Util.LIGHT_SPEED *
           pow(Util.AU * Util.sun_comet_distance, 2))

    var P: float = eps * (1 + Util.albedo)
    # P * 3 / (4 * d/2 * p)
    var _a: float = P * 3.0 /
            (4.0 *
                ((Util.particle_diameter / 1000.0) / 2.0) *
                (Util.particle_density * 1000.0))
    ...
    self.a = _a
```

This acceleration is then used in the `_accelerate_particle()` function, which calculates a displacement of the particle at each time step. The displacement is

determined by the initial velocity of the particle and the effect of solar pressure over its time alive, correctly oriented within a basis aligned to the Sun's position.

The calculation is performed within a specific coordinate system or `Basis` aligned with the Sun to simplify the physics. On this basis, the Sun's radiation pressure acts consistently along a single axis (the negative X-axis). The function first calculates the displacement due to the initial velocity of the particle (`V * t`). It then calculates the displacement from solar pressure using the standard kinematic equation $1/2 \cdot a \cdot t^2$, applying it along the predefined sun-facing axis. This local displacement vector is then transformed back into the global coordinate space and added to the initial spawn position of the particle to determine its final transform for the current frame.

```
func _accelerate_particle(i: int) -> void:
    # --- 1. Get Particle-Specific Data ---
    var _normal_dir_as_color := mm_emitter.multimesh.
            get_instance_custom_data(i) as Color
    var _normal_dir := Vector3(
            _normal_dir_as_color.r,
            _normal_dir_as_color.g,
            _normal_dir_as_color.b)
    var new_basis := Util.orbital_basis
    var time_passed: float = time_alive[i] * Util.jet_rate * 60.0
    time_alive[i] += 1
    # --- 2. Calculate initial vel and accel in global space
    var global_initial_velocity: Vector3 = _normal_dir * speed
    var sun_accel_magnitude: float = 0.5 * a * (time_passed ** 2)
    # --- 3. Change of Basis ---
    var local_velocity := global_initial_velocity * new_basis
    # --- 4. Calculate offset in the new space  ---
    var local_offset := Vector3(local_velocity * time_passed)
    local_offset.x -= sun_accel_magnitude
    # --- 5. Convert local offset to global
    var global_offset := local_offset * new_basis.transposed()
    # --- 6. Calculate Final Global Position ---
    var final_global_pos := initial_poss[i] + global_offset
    var scaled_final_pos := final_global_pos / (Util.scale)
    # update total space travelled by the particle
    total_space[i] += (scaled_final_pos - global_pos[i]).length()
    global_pos[i] = scaled_final_pos
    # --- 7. Create the Final Transform and Set it ---
    var final_global_transf := Transform3D(new_basis, scaled_final_pos)
    mm_emitter.multimesh.set_instance_transform(i, final_global_transf)
```

To add visual realism, a cloud of secondary particles is generated by the method `_generate_diffusion_particles()` around the primary path of the particle. This function calculates a "point cloud radius" based on the total distance the primary particle has traveled and the user-defined diffusion factor. It then spawns several

secondary particles at random positions within this radius, creating the characteristic fanned-out appearance of a cometary jet.

```
func _generate_diffusion_particles(i: int) -> void:
    if Util.n_points <= 0:
        return # no diffusion particles to generate
    var center_particle := mm_emitter.multimesh.get_instance_transform(i)
    var pc_radius := total_space[i] * (diffusion / 100) * randf()

    for j in range(1, Util.n_points + 1):
        var new_pos := Util.generate_gaussian_vector(0, 1, pc_radius)
        mm_emitter.multimesh.set_instance_transform(i + j,
                Transform3D(Basis(),center_particle.origin + new_pos))
        ...
```

4.2.4 JPL Horizons Import Module

To facilitate the setup of realistic simulation scenarios, **AETHER** integrates a module to directly query and import ephemeris data from the National Aeronautics and Space Administration (NASA) Jet Propulsion Laboratory (JPL) Horizons system. The JPL Horizons online service provides precise ephemerides for solar system objects.

API Interaction and Data Retrieval

The import process is managed by the `jpl_tab.gd` module, which provides a dedicated user interface for querying the Horizons Application Program Interface (API). The user can specify the target celestial body by name (e.g., "C/2013 R1"), a date range for the ephemeris data, and the desired time step between data points.

When a search is initiated, the module constructs and sends an Hypertext Transfer Protocol (HTTP) request to the JPL Horizons Web API. The request Uniform Resource Locator (URL) is built with several key-value parameters that define the query, including the object identifier (`COMMAND`), the start and stop times, the step size, and the specific physical quantities to be returned (e.g., astrometric position, distances, angular separations).

```
  func _on_search_btn_pressed()
      var params := {
          "format": "json",
          "COMMAND": "'%s'" % query,
          "OBJ_DATA": "NO",
          "MAKE_EPHEM": "YES",
          "EPHEM_TYPE": "OBSERVER",
          "CENTER": "'500@399'", # Geocentric
          "STEP_SIZE": "'%sh'" % int(step_size),
          "ANG_FORMAT": "DEG",
          "QUANTITIES": "'%s'" % quantities
```

```
    }
    # [...]
    var error := http_request.request(url)
```

Data Parsing and Processing

The JPL Horizons API returns a JavaScript Object Notation (JSON) object containing the requested ephemeris data within a single, multi-line string. This raw text block requires parsing to extract the meaningful values. The `parse_ephemeris()` function is responsible for this task.

First, the function isolates the main data table by locating the start (`$$SOE`) and end (`$$EOE`) markers within the text block. It then uses a regular expression to find and extract key orbital elements that appear before the main table, specifically the longitude of the ascending node (`OM`), the argument of perihelion (`W`), and the inclination (`IN`).

A second, more complex regular expression is then applied to iterate through each line of the ephemeris table. This pattern is designed to capture the distinct columns of data for each time step, such as the date, time, right ascension, declination, STO angle, and various distances.

```
func parse_ephemeris():
    var data_start_marker := "$$SOE"
    var data_end_marker := "$$EOE"
    # [...]
    # Isolate the ephemeris body
    var eph_body := data.substr(...)

    # Regex to find orbital elements like OM, W, IN
    var om_result := om_w_in_regex.search(data)
    # [...]

    # Regex to parse each line of the ephemeris table
    for line in eph_lines:
        var result := jpl_regex.search(line)
        if result == null:
            continue
        var entry := {
            "date": result.get_string(1),
            "time": result.get_string(2),
            "right_ascension": result.get_string(3),
            "declination": result.get_string(4),
            "sun_pa": result.get_string(5),
            # ... and so on for all other columns
        }
        # ...
```

Display and Integration

Once parsed, the extracted data is organized into a structured format. The orbital elements (OM, W, IN) are displayed in their respective fields in the UI. The time-series ephemeris data is then used to populate a table.

The `populate_container()` function dynamically creates UI elements (`Label` and `HBoxContainer` nodes) for each row of data and adds them to a scrollable list. The user can then reference this imported data to set the parameters in the "Comet" and "Simulation" tabs.

4.2.5 Camera Module

The camera module provides two distinct camera modes for navigating and inspecting the 3D simulation environment: a target-focused `RotatingCamera` and a `FreeRoamCamera`. A `CameraManager` handles the seamless transition between them.

RotatingCamera

The `RotatingCamera` is designed to orbit a central target, which in this case is the comet nucleus. This camera is ideal for observing the overall structure of the coma and the behavior of particle jets from a consistent focal point. It supports two projection modes: perspective and orthographic, allowing the user to switch between a realistic 3D view and a projection that preserves parallel lines, which can be useful for analytical purposes.

Controls

- **Rotation:** Holding the right mouse button and dragging the mouse rotates the view of the camera around the target. The sensitivity of this rotation can be adjusted.
- **Zoom:** The mouse wheel is used to zoom in and out, changing the distance of the camera from the target. The zoom level is constrained within a minimum and maximum range.
- **Reset View:** Pressing the "R" key resets the camera to its initial distance and orientation.
- **Toggle Projection:** The "P" key switches between perspective and orthographic projection modes.

Implementation

The position of the camera is calculated based on the yaw and pitch angles, which are updated in response to mouse input. The final position is determined by placing the camera at a specified distance from the target along its viewing axis.

The `look_at()` function ensures the camera is always pointing towards the target. When switching to orthographic mode, the `size` property of the camera is adjusted to match the visible area of the perspective view.

```
# rotating_camera.gd

# Update the camera's position and orientation based on the
    current state
func _update_camera_transform() -> void:
    if _target_node:
        _target_position = _target_node.position

    # Calculate camera position based on yaw, pitch, and distance
    var new_transform := Transform3D.IDENTITY
    new_transform = new_transform.rotated(Vector3.UP, _yaw) #
        Apply yaw
    new_transform = new_transform.rotated(new_transform.basis.x,
        _pitch) # Apply pitch

    # Position the camera 'distance' units away
    position = _target_position + new_transform.basis.z * distance

    # Make the camera look at the target
    look_at(_target_position, Vector3.UP)
```

FreeRoamCamera

The `FreeRoamCamera` offers a first-person-style navigation, allowing the user to move freely throughout the 3D space without being tethered to a specific target. This mode is beneficial for close-up inspection of particle trajectories and for exploring the simulation from any desired angle of view.

Controls

- **Movement:** Standard WASD keys are used for forward, left, backward, and right movement. "E" and "Q" control movement up and down, respectively.
- **Rotation:** Holding the right mouse button captures the cursor and allows the user to look around by moving the mouse.
- **Speed Control:** The mouse wheel adjusts the maximum movement speed. Holding "Shift" temporarily increases speed, while "Alt" decreases it.
- **Reset View:** Pressing the "R" key returns the camera to its default starting position and rotation.

Implementation

The physics-based movement of the camera uses an acceleration and deceleration model to provide a smoother feeling of motion. A velocity vector is updated based on user input, which is then applied to the position of the camera in the `_process` loop. Mouse input is used to adjust the rotation of the camera on its local axes, with pitch being clamped to prevent the camera from flipping upside down.

```
# free_roam_camera.gd

# Updates camera movement based on keyboard input
func _update_movement(delta: float) -> void:
    # Computes desired direction from key states
    _direction = Vector3(
        (_d as float) - (_a as float),
        (_e as float) - (_q as float),
        (_s as float) - (_w as float)
    )

    # Computes the change in velocity
    var offset := _direction.normalized() * _acceleration *
       _vel_multiplier * delta \
       + _velocity.normalized() * _deceleration *
          _vel_multiplier * delta

    # ... (speed modifiers and velocity clamping) ...

    translate(_velocity * delta * speed_multi)
```

CameraManager

The `CameraManager` is a simple but crucial component that manages which camera is currently active. It references both the `RotatingCamera` and `FreeRoamCamera` instances.

Implementation

A single function, `change_camera()`, handles the logic for switching between the two camera modes. When called, it checks which camera is currently active, turns it off by setting its `enabled` property to false, and activates the other camera by setting its `enabled` property to true. It also updates the `current` property of each camera to inform the engine which view of the camera should be rendered. This ensures that only one camera is processing input and rendering to the viewport at any time. Additionally, it updates a UI label to inform the user of the current camera mode.

```
# camera_manager.gd
```

```
# Switch between a rotating camera and a free roam camera.
func change_camera() -> void:
    if _fr_camera.current:
        # sets the current camera to the rotating camera
        _rot_camera.current = true
        _fr_camera.current = false
        # disable input for the free roam camera and enable the
           rotating camera
        _rot_camera.enabled = true
        _fr_camera.enabled = false
        # ... (update UI label)
    else:
        # sets the current camera to the free roam camera
        _fr_camera.current = true
        _rot_camera.current = false
        # disable input for the rotating camera and enable the
           free roam camera
        _fr_camera.enabled = true
        _rot_camera.enabled = false
        Util.current_camera_label.text = "Free␣Roam␣Camera"
```

4.2.6 Save-Load System

To facilitate the reuse of complex simulation setups, **AETHER** includes a system for saving and loading all relevant parameters to a configuration file. The architecture is designed to be both centralized and modular, ensuring that individual components manage their own data while a single entity handles the core file operations.

Architecture

The system is built around a singleton component named `SaveManager`, globally accessible throughout the application. The primary role of the manager is to hold an instance of Godot's `ConfigFile` object, which is an in-memory representation of the configuration file. The `SaveManager` exposes simple methods to read this object from or write it to the disk.

Save Process

When the user initiates a save operation, a signal is emitted to all relevant UI modules, instructing them to persist their current state. Each module implements a `save_data()` function, which is responsible for gathering the values from its respective input fields (e.g., text boxes, sliders) and writing them to the `ConfigFile` instance within the `SaveManager`. Data is organized into sections (e.g., "[comet]", "[simulation]") for clarity and ease of understanding.

```
# Example: Saving data from a UI panel
func save_data() -> void:
    SaveManager.config.set_value("comet", "radius",
       float($Control/EditRadius.text))
    SaveManager.config.set_value("particle", "albedo",
       float($Control/EditAlbedo.text))
    # ... and so on for all other parameters
```

Once all modules have written their data, the `SaveManager` serializes the entire `ConfigFile` object into a human-readable, INI-style format and saves it to the specified file path.

Load Process

The loading process is the reverse. First, the user selects a configuration file, which the `SaveManager` loads and parses into its internal `ConfigFile` object. Subsequently, a signal is broadcast to the UI modules, triggering their `load_data()` functions. Each module is then responsible for querying the `SaveManager` for the values it needs and using them to populate its own UI elements.

```
# Example: Loading data into a UI panel
func load_data() -> void:
    var radius = SaveManager.config.get_value("comet", "radius",
       0)
    $Control/EditRadius.set_value(float(radius))

    var albedo = SaveManager.config.get_value("particle",
       "albedo", 0)
    $Control/EditAlbedo.set_value(float(albedo))
```

This separation of concerns ensures that the SaveManager remains independent of the details of individual UI components. Its role is limited to brokering configuration data, while each module maintains full responsibility for managing its own state.

4.2.7 Optimizations

The following section presents the performance optimizations introduced in the system.

MultiMesh Batching

Rendering Approach Using individual nodes for each particle results in excessive draw calls and significant per-node overhead. The `MultiMeshInstance3D` class addresses this by allowing a single node to represent many instances, all rendered in a single draw call. Each instance can store its own transform, color, and custom per-instance data, enabling independent positioning, orientation, and visual properties for each particle without the overhead of separate nodes.

In **AETHER**, each emitter is associated with its own `MultiMeshInstance3D`, which manages all particles emitted from that source. This approach ensures that even emitters producing tens of thousands of particles can be rendered efficiently while maintaining per-particle control.

Additionally, the particle representation was changed from a full spherical mesh to a simpler point mesh, drastically reducing geometry complexity and improving rendering performance while remaining visually sufficient for dust and coma particles.

Implementation

The `MultiMesh` is allocated once with a fixed maximum instance count based on the number of particles needed. Active particles increase the `visible_instance_count`, avoiding dynamic memory allocation during the simulation. Each emission normal of the particle and other parameters are encoded in the `custom_data` of the instance, while color variations are passed via the instance color.

Particle updates are performed differently depending on the simulation mode. In the instantaneous mode, all particle transforms are precomputed and stored in a contiguous `PackedFloat32Array`, which is then uploaded to the GPU in a single call. In real-time mode, only the transforms of newly spawned or moved particles are updated each frame.

The corresponding code for updating particle transforms is as follows:

```
# Real-time simulation (_accelerate_particle in emitter.gd)
# Update the transform of the i-th active particle
mm_emitter.multimesh.set_instance_transform(i,
    instance_local_transform)

# Instantaneous simulation (instant_simulation in emitter.gd)
# Loop through all steps to compute particle transforms
_append_data_to_mm_buffer(mm_buffer, ith_transform, color)
# After all transforms are computed, upload the entire buffer at
    once
mm_emitter.multimesh.set_buffer(mm_buffer)
```

Editor-based and Parameter Tweaks

Rationale Some expensive features do not visually affect the dust-like particles and thus can be turned off to reduce work for both GPU and CPU. By tweaking certain parameters through both the editor and the code, the execution time of the simulation, both instant and not, significantly improved. The following tweaks were performed:

Tweaks

- **Shadows and GI.** Disable shadow casting and receiving for particle materials; render them unshaded.
- **Static typing in GDScript.** By default, GDScript is dynamically typed, meaning a variable can change its type at runtime. However, Godot also supports static typing, which enforces type consistency, reduces the likelihood of runtime errors, and allows the engine to apply additional optimizations during execution.
- **Material Mode.** Use a single material for all the particles.

Emitter Illumination

In `AETHER`, the illumination of an emitter is determined to decide whether it should spawn particles. Two approaches are implemented: a raycasting-based method (`is_lit`) and an analytic dot-product method (`is_lit_math`).

The raycasting method checks for occlusions between the emitter and the light source, handling arbitrary geometries but at a higher computational cost:

```
# Raycasting approach
var space_state := get_world_3d().direct_space_state
var query :=
   PhysicsRayQueryParameters3D.create(light_source.global_position,
   global_position)
query.collide_with_areas = true
query.exclude = [$Particle/ParticleArea.get_rid()]
var result := space_state.intersect_ray(query)
is_lit = result.is_empty()
```

The dot-product method assumes a spherical emitter surface and computes illumination using the normal of the emitter and the direction of the Sun. It is ≈ 1.5x faster ($\sim 20\mu s$ vs $\sim 30\mu s$) and more accurate for spherical geometries, but it does not work if the comet surface is not spherical:

```
# Dot-product approach
func is_lit_math() -> bool:
    var comet_basis: Basis = get_parent().global_transform.basis
    var global_space_normal: Vector3 = (comet_basis *
       norm).normalized()
    global_space_normal =
       global_space_normal.rotated(Vector3.LEFT,
       deg_to_rad(Util.sun_direction + 90))
    var result: float = (-Util.sun_direction_vector).dot(norm)
    return result > 0
```

In practice, `is_lit_math` is used for efficiency on the rough spherical surface of the comet, allowing the simulation to determine which emitters are illuminated quickly. This trade-off prioritizes performance and accuracy for spherical objects, while sacrificing generality for complex non-spherical geometries.

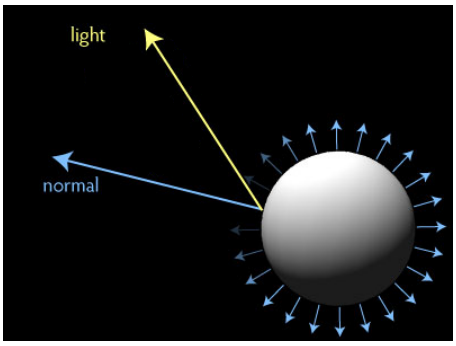


Figure 4.3: Illustration of how the dot product determines if a point on a sphere is illuminated. The surface normal at the point (represented by the blue vector) is compared with the light direction (represented by the yellow vector). The cosine of the angle between them, as indicated by their dot product, represents the intensity of illumination: positive values correspond to lit regions, while zero or negative values correspond to shadowed areas.

4.3 User Manual

4.3.1 Installation

To install the application, download the latest `.exe` build from the GitHub repository of the project.[1] For legacy or older systems, it is strongly suggested to use the legacy build (`GL` version) available in the releases section.

Compiling from Scratch

To build the application yourself, it is **recommended** to compile **Godot Engine 4.4** from source with the double-precision flag enabled. Alternatively, you may use the stable 4.4 release build, which might cause some accuracy issues.

1. Compile the Godot Engine 4.4 and export templates from source with the double-precision flag enabled, or download the stable release if preferred (with "built-in" export templates).
2. Clone the project repository:

```
git clone https://github.com/GDennis01/Project-Aether
```

3. Open Godot and import the cloned repository as a new project.
4. Build your custom version using the Godot editor's export menu.

Note: Detailed compilation instructions for Godot, including how to enable custom flags, are available in the official documentation.[2]

[1] `https://github.com/GDennis01/Project-Aether/releases`

[2] `https://docs.godotengine.org/en/4.4/contributing/development/compiling/index.html`

4.3.2 User Interfaces

The Graphical User Interface (GUI) of the application is organized into two primary tabs: **Settings** and **Model**.

Settings Tab

The **Settings** tab, shown in Figure 4.4, is responsible for fetching ephemeris data from NASA JPL Horizons service and for configuring the parameters required to compute the scale of the model.

The tab is divided into two main panels. The left panel is dedicated to parameter input, while the right panel displays the retrieved ephemeris data.

Figure 4.4: The Settings tab, where it is possible to import ephemeris data from NASA JPL Horizons service and configure the input needed to compute the scale of the model.

The configuration panel on the left is further organized into four distinct sections:

JPL Horizons Query This section is used to request ephemeris data by specifying the following parameters:

- **Comet Name**: The comet name for which the ephemeris data are desired. The search button sends a query to the NASA JPL Horizons service.
- **Start/End Date**: Starting and (optional) ending date of the ephemeris data
- **Step Size**: Time increment, in hours, for the requested ephemeris data.

Loading the data from the Horizons service also sets up automatically some geometry data (e.g., sun distance and position, distance of the comet from Earth),

as well as data needed to run the model, in the **Model** tab (or override them if already present), so this section is crucial for the whole software! Manual input is still possible in the event of automatic import failure.

Orbital Elements Once the data has been successfully retrieved, this section displays the comet's key orbital elements (Section 2.2.2):

- **EC (e):** Eccentricity of the orbit.
- **QR (q):** Perihelion distance in AU.
- **TP (T):** Time of perihelion passage.
- **OM (Ω):** Longitude of the ascending node
- **W (ω):** Argument of perihelion.
- **IN (i):** Inclination.

Before importing, users can also customize which ephemeris parameters are displayed in the main table on the right; the parameters needed for the model are uploaded regardless.

Telescope Properties This section allows for the configuration of the reference telescope image, which determines the scale of the model:

- **Delta (AU):** The distance between the observer (the telescope) and the comet in Astronomical Units.
- **Charge-Coupled Device (CCD) image size (pixels):** The dimensions, in pixels, of the telescopic image. A square image is assumed (width equals height).
- **Image resolution (arcsec/pixel):** The angular resolution of the telescope image.
- **Image resolution (km/pixel):** The physical scale at the distance of the comet, calculated from the above parameters.
- **Field of View (FOV) (arcseconds and Km):** The field of view of the telescope image and of the main window, in both arcseconds and kilometers.

This final section provides an option to load a telescope image to be placed as a layer below the model, allowing for direct comparison.

Right panel This panel displays the retrieved ephemeris data and allows downloading the data in a **Comma Separated Values (CSV)** file format.

Model Tab

The **Model** tab, shown in Figure 4.5, is dedicated to defining the physical properties of the comet nucleus, of the emitted dust, and the dust jets, as well as controlling the model execution.

The tab is further divided into several distinct sections.

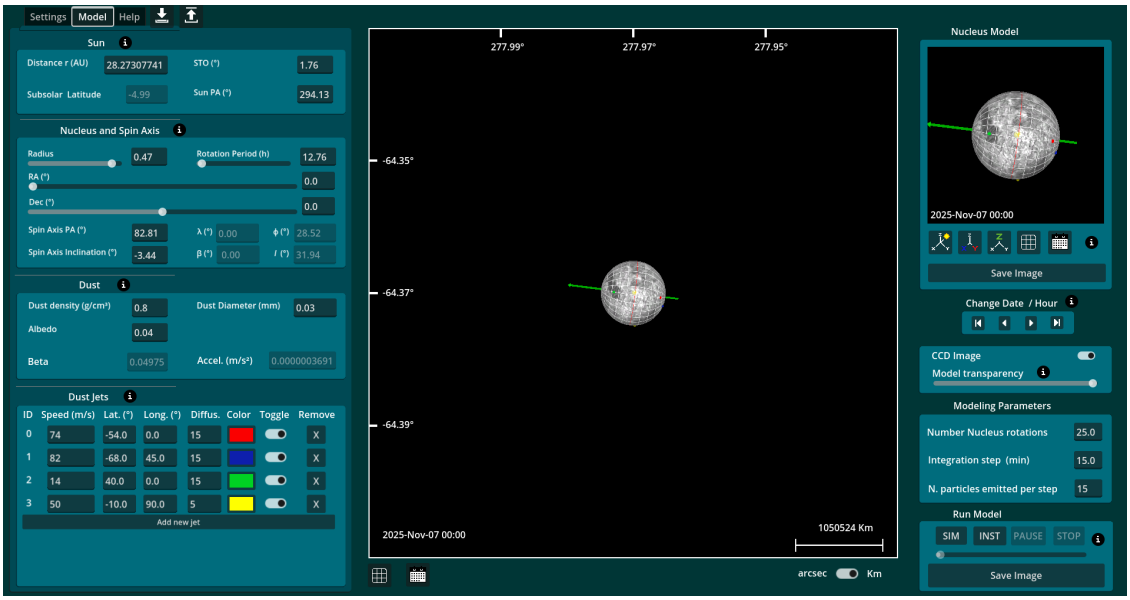


Figure 4.5: Model tab, where it's possible define the physical properties needed to start the modeling.

Sun This section displays key information about the Sun's position relative to the comet.

- **Distance r (AU):** The heliocentric distance of the comet.
- **STO (°):** The Sun-Target-Observer angle.
- **Subsolar Latitude:** The latitude of the point of maximum insolation on the nucleus. This field is automatically computed based on different orbital elements.
- **Sun PA (°):** The position angle of the Sun.

Nucleus and Spin Axis This section defines the physical characteristics and orientation of the comet's nucleus.

- **Radius:** The radius of the nucleus. This parameter is used for visualization purposes only and does not influence the dust dynamics model.
- **Rotation Period (h):** The time for one full rotation of the nucleus.
- **Spin Axis Orientation:** Defines the orientation of the nucleus spin axis.

  - **Right Ascension (RA) and Declination (DEC) (°):** The RA and DEC of the spin axis pole. These input values are estimated by the user and define the direction in which it is spinning (by convention, counterclockwise).
  - **PA and Inclination (°):** Define the direction of the spin axis and its inclination onto the sky plane. They are computed automatically from the Spin Axis RA and DEC coordinates, or can be entered manually.
  - **Important Distinction:** The RA and DEC values for the spin axis must not be confused with the ephemeris data in the **Settings** tab.
    * **Spin Axis RA/DEC (Model Tab):** A pair of coordinates defining the **orientation** of the comet's pole.
    * **Ephemeris RA/DEC (Settings Tab):** Define the **position** of the comet in the sky as seen from the observer.

- λ, β: Ecliptical coordinates of the pole. They are computed automatically based on estimated RA and DEC of the pole.
- $\phi, \boldsymbol{i}$: Orbital coordinates of the pole. They are computed automatically based on λ and β.

Dust Defines the properties of the dust particles.

- **Dust Density ($\mathbf{g/cm^3}$):** The bulk density of the dust emissions.
- **Dust Diameter (mm):** The size of the dust grains.
- **Albedo:** The reflectivity of the dust.
- **Beta:** A calculated, read-only value representing the ratio of solar radiation pressure force to solar gravity for the selected dust properties.
- **Accel. ($\mathbf{m/s^2}$):** The resulting acceleration of the dust particles due to solar radiation pressure. It is a read-only field.

Dust jets A table to manage active emission regions (jets) on the nucleus surface. Each jet is defined by its **ID**, **Speed**, **location** (latitude and longitude), **diffusion** factor, and **color**. Each jet can be toggled on/off and removed. A new jet can be added using the "Add new jet" button.

Change Date Once the ephemeris data has been imported, it is possible to switch to the previous or next date, or to the first or last date. This alters the orientation of the spin axis and the extent of isolation of the nuclear model in response to the changing geometric conditions.

Telescope Image Once a telescope image of the comet has been selected in the **Settings** tab, it is shown in a layer below the model; it is possible here to toggle it on/off and to regulate the transparency of the model to compare it to the telescopic image better.

Modeling Parameters Defines the parameters of the modeling (number of nucleus rotations, integration step, and number of particles emitted each step).

Running the model Once all the parameters have been configured, it is possible to run the model by pressing, in the bottom-right part of the tab, the **INST** button for instant modeling or the **SIM** button for automated step-by-step modeling. It is also possible to save the image of the modeling.

Nucleus model A dedicated 3D view in the upper-right of the tab shows a close-up of the nucleus, its orientation, the spin axis (green arrow, Y axis), and the direction to the Sun (yellow). It is possible to toggle each axis on or off, to display the grid pattern on the model, and to show the current date if the ephemeris data has been imported. This view also allows for saving an image of the nucleus model.

Model Viewport A dedicated viewport of the model of a size of 900x900 pixels. Here is where the modeling is displayed. It is possible to toggle the display of the grid showing RA/DEC coordinates and the date on or off. It is also possible to view the scale of the modeling in either arcseconds or kilometers.

Miscellaneous features

Help panel A dedicated help panel is available, providing a quick reference for navigating the **Model Viewport**. The primary controls are:

- **Right-click and drag**: Rotate the camera around the comet.
- **R key**: Reset the camera to the default Earth-based perspective.
- **Mouse Wheel Up**: Zoom in.
- **Mouse Wheel Down**: Zoom out.

Tooltips Most input fields are accompanied by an info button to enhance usability. Hovering the cursor over this button reveals a tooltip that provides a brief description of the parameter or its function.

Load and Save Configuration The interface includes buttons that allow users to save their current session settings to a configuration file. This file can be reloaded at a later time, allowing users to restore a previous modeling session easily.

Chapter 5

Experiments

In this chapter, a series of experiments performed with AETHER is presented. Each section is dedicated to a specific comet.

5.1 C/2025 A6 (Lemmon)

C/2025 A6 (Lemmon) is a recently discovered comet (as of the time of writing), first spotted in January 2025 by the Mount Lemmon Survey in Arizona. The comet follows a long, elliptical orbit that brings it close to the Sun approximately every 1˙300 years.

The comet reached its closest point to the Sun (perihelion) on November 8, 2025, at a distance of about 0.53 AU, and made its closest approach to Earth a few weeks earlier, on October 21, 2025, at roughly 0.60 AU. The nucleus of **Lemmon** has been modeled with five active emission regions: four located along the 0° meridian and one at 90°, with latitudes ranging from –40° to +60°.

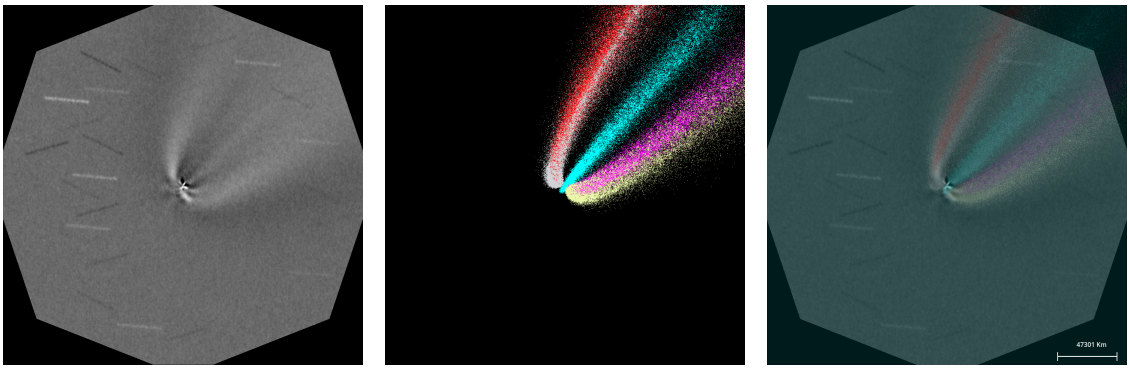
The experiments on **Lemmon** were carried out on two different dates: October 8 and October 11, 2025.

5.1.1 8th October 2025

On this date, the comet was located at a distance of ≈0.82 AU from Earth and ≈0.88 AU from the Sun. The experiment was conducted using an image taken with the Savonarola 0.4m telescope (Stazione Astronomica di Sozzago, Italy) IAU-MPC A12 (Figure 5.1). The overlay comparison shows that AETHER successfully reproduces the overall structure of the comet's coma observed on this date.

5.1.2 11th October 2025

At this time, the comet had moved closer to both Earth and the Sun, with distances of ≈ 0.74 AU and ≈0.83 AU, respectively. The experiment was based on an image taken with the Schmidt 67/92 telescope (Asiago Mount Ekar, Italy) INAF-OAPd (Figure 5.2). The overlay comparison confirms that AETHER continues to model the coma structure with high accuracy.

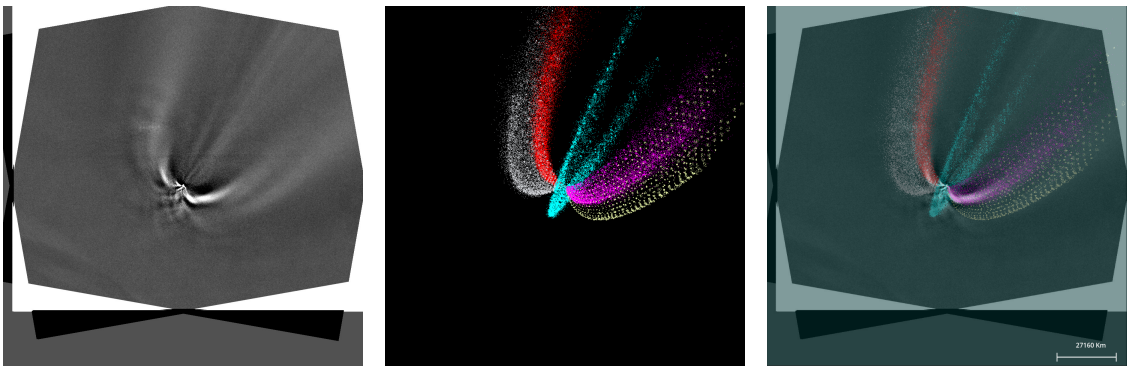


(a) Telescopic image of comet C/2025 A6 (Lemmon) captured by Federico Manzini on October 8, 2025.

(b) Simulation generated by AETHER for the same date.

(c) Overlay comparison between model and observation.

Figure 5.1: Observations and simulations of comet C/2025 A6 (Lemmon) on 8 October 2025.



(a) Telescopic image of comet C/2025 A6 (Lemmon) captured by Federico Manzini on October 11, 2025.

(b) Simulation generated by AETHER for the same date.

(c) Overlay comparison between model and observation.

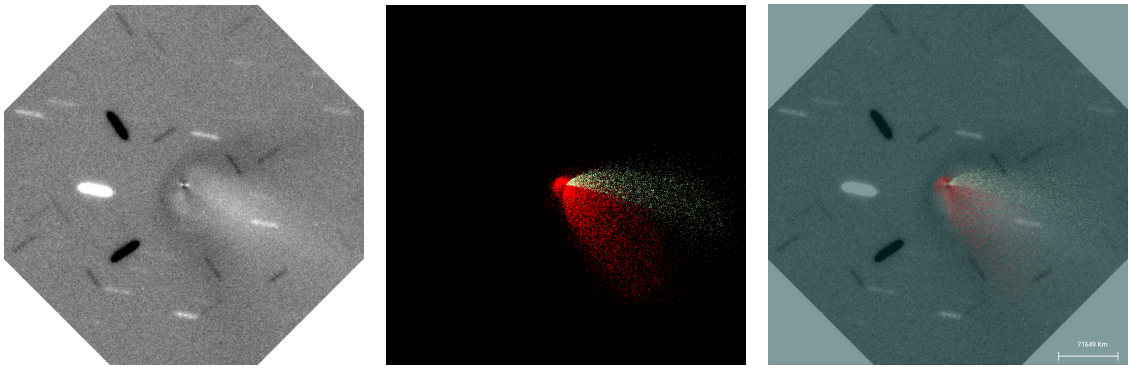
Figure 5.2: Observations and simulations of comet C/2025 A6 (Lemmon) on October 11, 2025.

5.2 C/2022 N2 (Pan-STARRS)

C/2022 N2 (Pan-STARRS) is a comet discovered by the Pan-STARRS 2 telescope located at the Haleakalā observatory on July 4, 2022. The comet has a nearly hyperbolic trajectory, suggesting it may have originated from the Oort cloud or beyond. The nucleus of **C/2022 N2** has been modeled with two active emission regions: both located at the 0° meridian with latitudes at -40° and 40° respectively.

The experiment on the **Pan-STARRS** was carried out on August 15, 2024, at a heliocentric distance of about 3.9 AU, using an image taken with the Copernico 1.82m telescope (Asiago Mount Ekar, Italy) INAF-OAPd (Figure 5.3).

Again, AETHER shows promising results in matching the coma structure of the comet.



(a) Telescopic image of comet C/2022 N2 (Pan-STARRS) captured by the Asiago Astrophysics Observatory on August 15, 2024.

(b) Simulation generated by AETHER for the same date.

(c) Overlay comparison between model and observation.

Figure 5.3: Observations and simulations of comet C/2022 N2 on August 15, 2024.

5.3 C/2013 R1 (Lovejoy)

C/2013 R1 (Lovejoy) is a long-period comet discovered on September 7, 2013, by Australian astronomer Terry Lovejoy using a 0.2 meter Schmidt–Cassegrain telescope. It became one of the most visible comets from the Northern Hemisphere in late 2013, reaching naked-eye visibility in November of that year. Its orbit has a period of approximately 7˙000 years, with a perihelion distance of 0.81 AU, which it reached on December 22, 2013 [5].

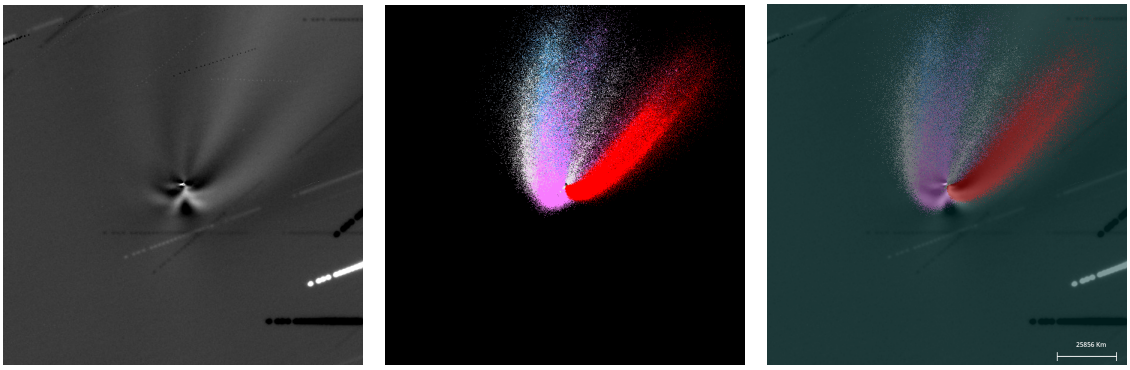
The nucleus of **Lovejoy** has been modeled with four active emission regions, all located close to the 0° meridian except one slightly shifted eastward. These regions are distributed across a wide range of latitudes: one in the southern hemisphere at –62°, another near the equator at –10°, a third in the northern mid-latitudes at +30°, and a fourth at +10° latitude and +3° longitude.

The experiment on the **Lovejoy** comet was carried out on December 3, 2023, at a heliocentric distance of 0.89 AU, using an observational image obtained by Federico Manzini (Figure 5.4).

5.4 67P/Churyumov-Gerasimenko

67P/Churyumov–Gerasimenko is a short-period comet with an orbital period of approximately 6.45 years. It was discovered in 1969 by Klim Churyumov and Svetlana Gerasimenko from the Kiev University Astronomical Observatory during a photographic survey of comet 32P/Comas Solà. The comet gained scientific interest as the primary target of the *ESA Rosetta* mission. [6]

The nucleus of **67P/Churyumov–Gerasimenko** has been modeled with four active emission regions derived from the observed jet morphology. These regions are located at latitudes and longitudes of (–52°, 0°), (–68°, 45°), and (40°, 90°), covering both hemispheres and spanning a wide longitudinal range.



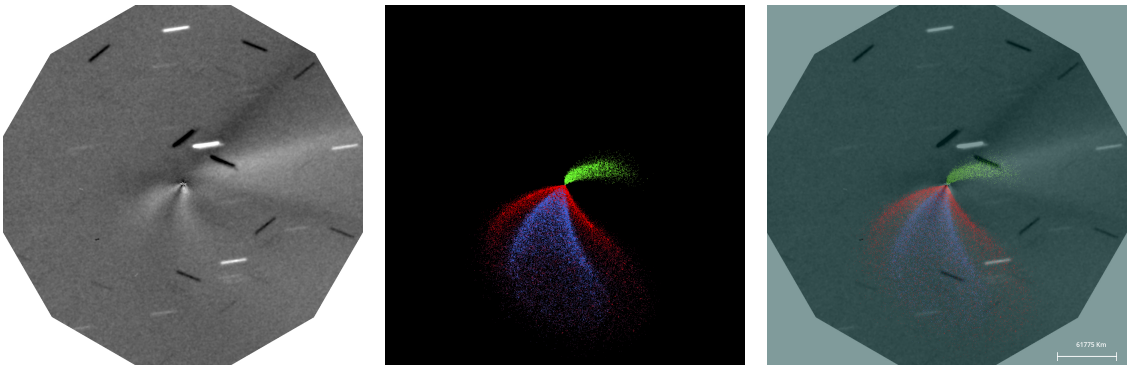
(a) Telescopic image of comet C/2013 R1 (Lovejoy) captured by Federico Manzini on December 3, 2023.

(b) Simulation generated by AETHER for the same date.

(c) Overlay comparison between model and observation.

Figure 5.4: Observations and simulations of comet C/2013 R1 (Lovejoy) on 03 December 2023.

The experiment on the **67P** comet was carried out on January 11, 2022 at a heliocentric distance of ≈ 1.48 AU, using an observational image obtained by Virginio Oldani (Figure 5.5).



(a) Telescopic image of comet 67P/Churyumov-Gerasimenko captured by Virginio Oldani on January 11, 2022.

(b) Simulation generated by AETHER for the same date.

(c) Overlay comparison between model and observation.

Figure 5.5: Observations and simulations of comet 67P/Churyumov-Gerasimenko on January 11, 2022.

Chapter 6

Conclusion

6.1 Conclusion

While the system successfully models dust particle dynamics in a 3D environment, the experimental phase highlighted specific limitations in the current implementation.

Notably, the algorithm for computing the model scale requires refinement to ensure mathematical consistency with real-world dimensions. Furthermore, the rendering pipeline currently produces a greenish hue in the particle model; this artifact lowers the contrast and makes direct visual comparison with grayscale telescopic images less efficient. Addressing these flaws is a priority for future development.

6.2 Future Updates

6.2.1 GPU Particle Simulation with Compute Shaders

The current CPU-based particle simulation is a performance bottleneck for high particle counts. A key future update is to migrate the physics calculations to a **compute shader**. Offloading the particle update loop to the GPU would allow for massively parallel processing. This would involve storing particle data (position, velocity, etc.) in GPU buffers and using a compute shader to update their states each frame, dramatically increasing the scale and density of the simulated coma.

6.2.2 Ephemeris-Driven Dynamic Simulation

Currently, the application uses imported JPL Horizons data to set the initial conditions for a simulation at a single point in time. A key future enhancement is to allow the simulation to run across the entire imported date range.

This would involve stepping through each ephemeris entry, automatically updating the orbital parameters of the comet, such as its distance from the sun, position, and orientation, at each time step. This feature will enable the modeling of the

dust environment over extended periods, showing how the coma and tail evolve as the comet travels along its orbit and providing a more dynamic and scientifically accurate simulation.

6.2.3 Modeling Rotational Precession

The current model assumes the comet rotates about a fixed axis. Implementing **rotational precession** would be a significant enhancement. For a non-spherical nucleus subject to torques from asymmetric outgassing, the spin axis will not remain fixed in space but will precess over time.

This motion is critical for accurately modeling the long-term illumination conditions on the surface of the comet. By simulating precession, the model will account for the evolving orientation of active emission regions relative to the Sun.

6.2.4 UI Visual Redesign

The current UI is built with a focus on functionality, utilizing Godot's default UI components. A future update will involve a complete visual redesign to create a more polished and professional interface. This would include the development of a custom theme with a proper color palette and fonts appropriate for a scientific tool. Additionally, custom icons would be designed to replace standard buttons, improving visual clarity and providing a better user experience.

Bibliography

[1] F. Manzini, V. Oldani, R. Behrend, P. Ochner, O. Baransky, and D. Starkey, «Comet C/2013 X1 (PanSTARRS): Spin axis and rotation period», *Planetary and Space Science*, vol. 129, 2016.

[2] J.-B. Vincent, «From observations and measurements to realistic modeling of cometary nuclei», Ph.D. dissertation, Technische Universität Braunschweig, 2010. [Online]. Available: `https://www.mps.mpg.de/phd/theses/from-observations-and-measurements-to-realistic-modeling-of-cometary-nuclei.pdf`.

[3] J.-B. Vincent, H. Böhnhardt, and L. M. Lara, «A numerical model of cometary dust coma structures», *Astronomy & Astrophysics*, vol. 512, A60, Apr. 2010, Planets and planetary systems. DOI: `10.1051/0004-6361/200913418`.

[4] A. L. Lesage and G. Wiedemann, «Determination of the position angle of stellar spin axes», *Astronomy & Astrophysics*, vol. 563, A86, 2014. DOI: `10.1051/0004-6361/201322964`.

[5] G. W. Kronk, *C/2013 R1 (Lovejoy)*, Gary W. Kronk's Cometography, Archived from the original on 10 November 2013., 2013. [Online]. Available: `https://web.archive.org/web/20131110072605/http://cometography.com/lcomets/2013r1.html`.

[6] International Astronomical Union, *Klim ivanovich churyumov*, International Astronomical Union.