A Real Time N-Body Gravitational Simulation

Shamil M

Abstract

This document details the design and implementation of a real time, three-dimensional N-body simulation. The project's core is a modular physics engine featuring a fourth-order Runge-Kutta (RK4) integrator for orbital mechanics. This is coupled with a renderer built on OpenGL. Key graphics features include a multi pass post-processing pipeline, bloom, a dynamic particle system, and image based lighting (IBL). The system's flexibility is demonstrated through the successful simulation of both a stable, hierarchical solar system analogue and a classic, chaotic three-body figure-eight orbit.

Table of Contents

1. Introduction

1.1 The N-Body Problem

The N body problem is one of the most fundamental and enduring challenges in the fields of physics, mathematics, and celestial mechanics. It seeks to predict the individual motions of a group of objects interacting with each other gravitationally, such as stars in a galaxy or planets in a solar system. While Sir Isaac Newton provided a perfect, closed form solution for the two body problem, which allows us to calculate the precise orbits like that of the Earth around the Sun, the problem becomes extraordinarily complex when a third body is introduced.

For three or more bodies ($N \geq 3$), a general analytical solution does not exist. This lack of a general solution gives rise to the phenomenon of chaos. In this context, chaos does not mean randomness, but rather an extreme sensitivity to initial conditions, famously known as the "Butterfly Effect." An infinitesimally small change in the starting position or velocity of one body can lead to a drastically different and unpredictable outcome over time. This makes long-term

prediction of such systems impossible through simple formulas, necessitating the use of computational methods. Therefore, to explore the beautiful and often counter intuitive choreographies of these systems, one must turn to **numerical simulation**.

1.2 Project Goals

The primary goal of this project was to design and implement a versatile, real-time, three-dimensional N-body simulation tool capable of demonstrating these complex gravitational interactions. The project was guided by the following key objectives:

- **Physical Accuracy:** To implement and verify a high-accuracy numerical integrator, the fourth-order Runge-Kutta (RK4) method. The aim was to ensure the simulation conserves total system energy over long durations.
- **High-Fidelity Graphics:** To build a rendering engine using OpenGL, moving beyond simple representations to achieve a high degree of visual realism. This included implementing a lighting model (initially tried PBR, then Blinn-Phong), a multi-pass post-processing pipeline for effects like bloom, and IBL.
- **Informative Visualization:** To create visual tools that make the underlying physics intuitive and aesthetically compelling. The primary tool developed for this was a dynamic particle system to trace the orbital paths of the celestial bodies.
- **Flexibility and Experimentation:** To develop a modular system capable of loading and simulating distinct N-body scenarios. This demonstrates the tool's flexibility by showcasing both a stable, hierarchical system (a solar system analogue) and a classic chaotic system (the three-body figure-eight orbit).

1.3 Technology Stack

The simulation was developed from the ground up to ensure a deep understanding of all underlying systems. The following tech & libraries were used:

- **Programming Language: C++**, chosen for its high performance and low level memory control, which are essential for both intensive physics calculations and real-time graphics rendering.
- **Graphics API: OpenGL**, providing direct, cross-platform access to the GPU for hardware-accelerated 3D rendering.
- **Windowing & Context: GLFW**, a lightweight library used to create the window, handle user input (keyboard and mouse), and manage the OpenGL context.
- **OpenGL Loading: GLAD**, used to load pointers to the required OpenGL functions at runtime.
- **Mathematics: GLM (OpenGL Mathematics)**, a header-only C++ mathematics library based on the GLSL shading language specifications, used for all vector and matrix operations.
- **Texture Loading: stb_image**, a popular single-header image-loading library used to load skybox textures from disk.

2. Physics Engine Implementation

The core of this project is a physics engine capable of accurately modeling the gravitational interactions between multiple bodies. The development of this engine was approached by first establishing the foundational physical principles and then evaluating multiple numerical integration techniques to find a method that helped with the requirement of long term stability.

2.1 Core Principles

The simulation is governed by Sir Isaac Newton's Law of Universal Gravitation. This principle states that every particle in the universe attracts every other particle with a force that is directly proportional to the product of their masses and inversely proportional to the square of the distance between their centers.

The force (F) exerted on a body of mass m1 by a body of mass m2 is calculated using the formula:

$$F = G\frac{m1m2}{r^2}$$

Where:

- **G** is the gravitational constant.
- $\mathbf{m}_1$ and $\mathbf{m}_2$ are the masses of the two interacting bodies.
- **r** is the distance between the centers of the two bodies.

This force calculation is performed for every pair of bodies in the simulation in each time step. The resulting net force on each body is then used to determine its acceleration ($a = \frac{F}{m}$), which in turn updates its velocity and position.

2.2 Numerical Integration: The Challenge of Motion

While the force calculation is straightforward, determining a body's position over time is not. As the bodies move, the forces they exert on each other constantly change, creating a complex, dynamic system for which no simple analytical solution exists for N > 2. Therefore, the trajectory of each body must be calculated using a **numerical integrator**, which is an algorithm that approximates the solution by stepping forward in small increments of time (Δt).

The choice of integrator is one of the most critical design decisions, as it directly impacts the simulation's accuracy and stability. Three distinct methods were implemented and evaluated to find the most suitable for this project.

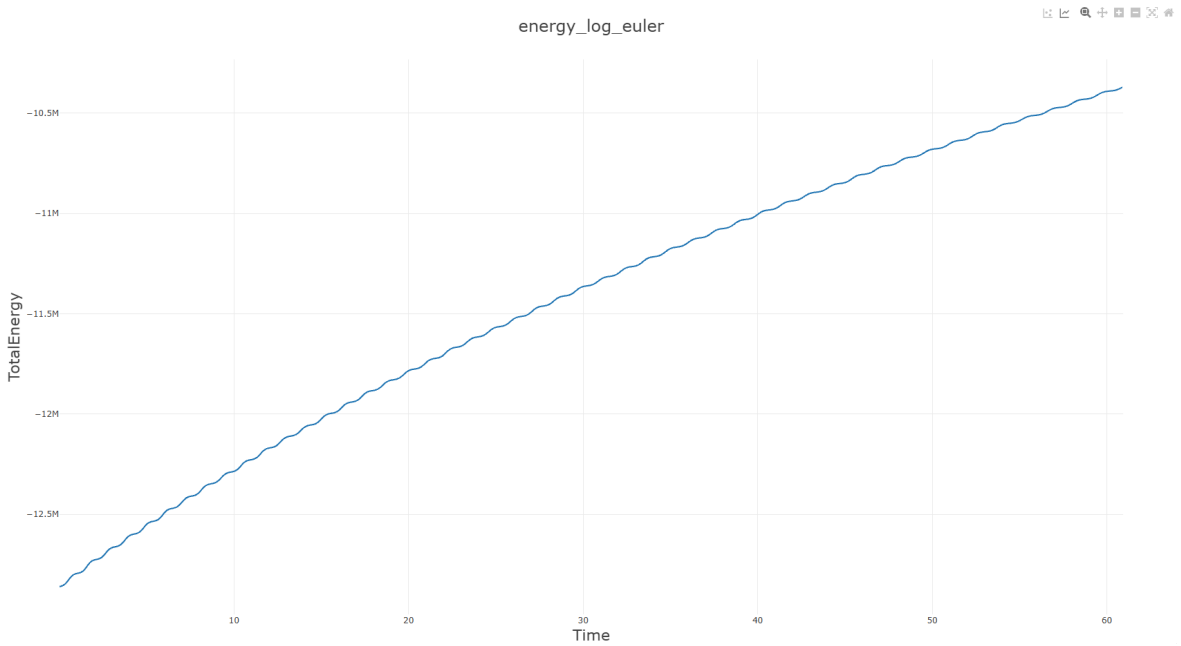
- **Method 1: The Euler Method** The simplest numerical integrator is the Euler method. It's a first order method that approximates the next position of a body by assuming its velocity is constant over a small time step. While fast to compute, its simplicity leads to a rapid accumulation of error, causing the total energy of the simulated system to drift over time. This makes it unsuitable for any long-term orbital simulation.
- **Method 2: The Leapfrog Method** The second method evaluated was the Leapfrog integrator. This is a second order method that is particularly well-suited for orbital mechanics. It achieves higher accuracy by "leaping" the position and velocity calculations over each other: it updates the velocity at a half time step, uses that new velocity to update the position for the full time step, and then completes the velocity update for the full time step. This structure makes it a **symplectic integrator**, meaning it has excellent properties for conserving the total energy of the system over very long periods, often showing only small, bounded oscillations around the true value.
- **Method 3: The Runge-Kutta 4th Order (RK4) Method** The final and most complex method implemented was the fourth-order Runge-Kutta (RK4) integrator. RK4 takes four "test" samples of the forces within each time step - at the beginning, two in the middle, and one at the end and calculates a weighted average. This process significantly reduces the error *per step*, making it exceptionally accurate for a wide variety of problems, including those with complex or non-conservative forces.

2.3 Results & Verification: Conservation of Energy

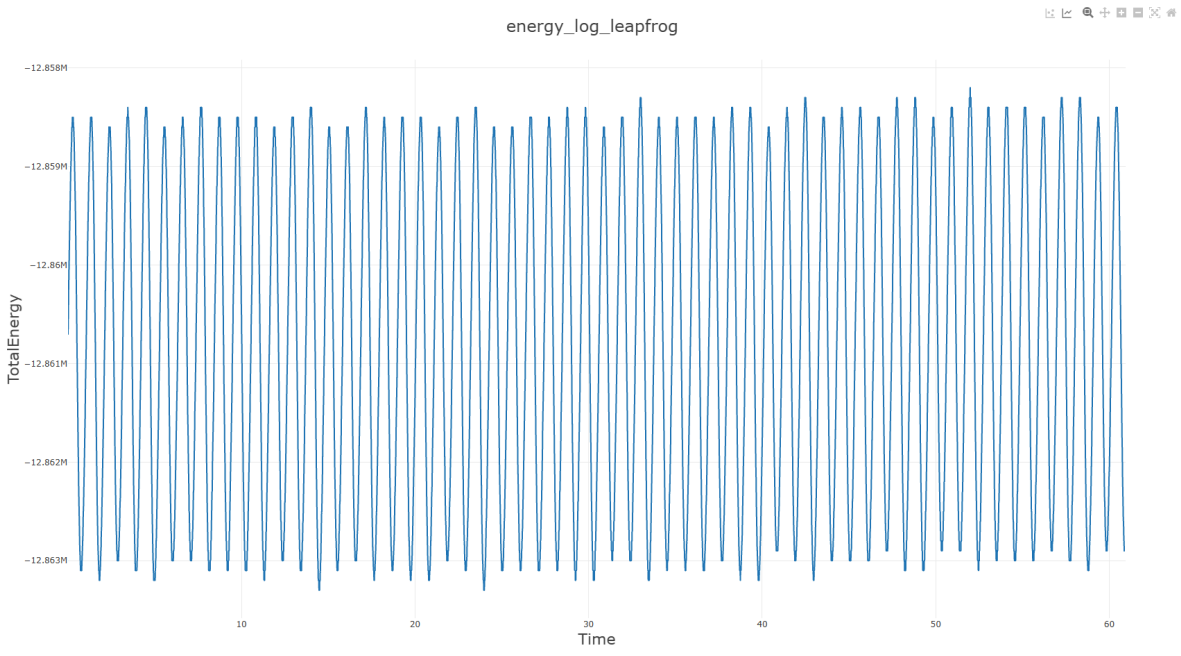
To be considered physically plausible, a gravitational simulation must adhere to the principle of conservation of energy. The total energy of the system (the sum of all bodies kinetic and potential energy) should remain constant over time. This became the primary benchmark for verifying the accuracy of the implemented physics engine.

An experiment was conducted to plot the total system energy over t = 60 seconds for all three integrators using the same initial conditions. The results are as such:

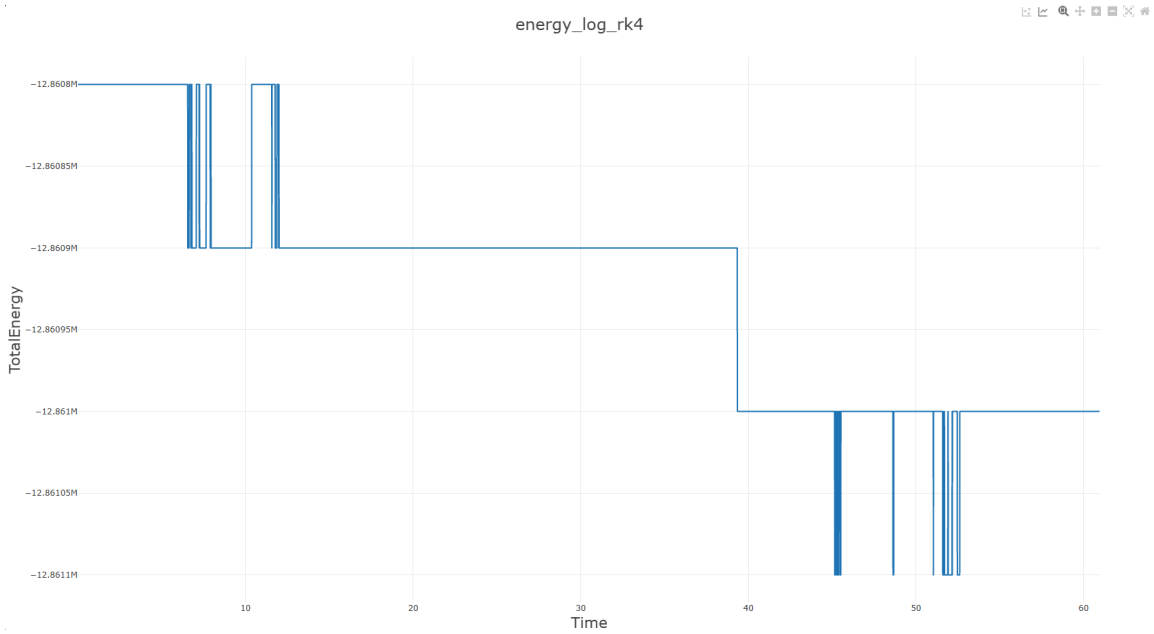
- The **Euler Method** showed a significant and steady drift in total energy, confirming its instability.



- The **Leapfrog Method** performed exceptionally well, showing almost no long-term energy drift, only minor high-frequency oscillations. This demonstrates its strength for long-term orbital simulations.



- The **RK4 Method** also demonstrated excellent performance, with the lowest error per step and a nearly perfectly flat energy conservation plot.



While the Leapfrog method is a great choice for purely gravitational systems, the RK4 method was chosen as the final implementation for this project due to its higher per step accuracy and its general purpose robustness, which would be beneficial for future extensions to the simulation.

All the CSV files are available:
Euler
Leapfrog
RK4

3. Rendering Engine & Visual Enhancements

While the physics engine dictates the motion of the celestial bodies, the rendering engine is responsible for bringing that motion to life in a visually compelling and informative way. The goal was to move beyond simple spheres and create a high fidelity, real time visualization. This was achieved through a multi-pass rendering pipeline that allowed for lighting and postprocessing effects.

3.1 The Rendering Pipeline

The renderer was built around a multi-pass, deferred shading style architecture. Instead of drawing directly to the screen, the main 3D scene is first rendered to an off-screen framebuffer object (FBO). This FBO was configured to use a High Dynamic Range (HDR) floating-point color buffer. This allows color values to exceed the standard 1.0 limit, enabling the storage of true light intensity. This capability is the foundation for post-processing effects like realistic bloom. After the initial scene is rendered, a series of subsequent passes apply effects before the final image is composed and drawn to the screen.

3.2 Environmental Rendering

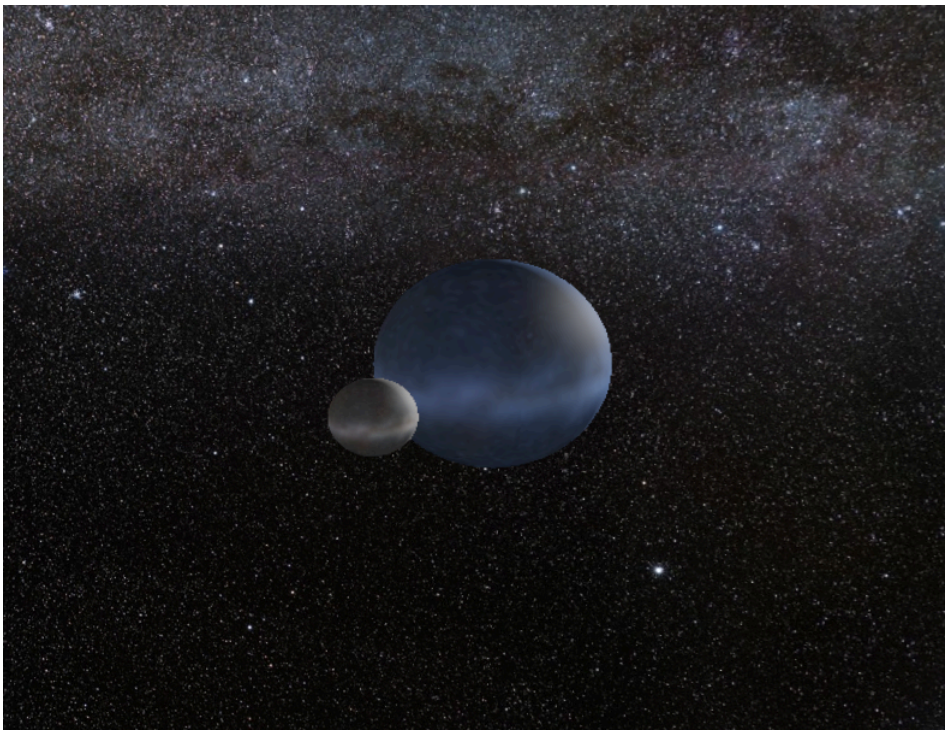
To create an immersive setting, two key techniques were used to render the space environment and utilize it for lighting.

- **3.2.1 Skybox** A skybox provides the backdrop for the simulation. This is implemented by mapping a cubemap texture, which is a collection of six 2D textures forming the faces of a cube, to the inside of a large cube that surrounds the entire scene. By keeping the camera centered within this cube, the skybox appears infinitely distant, effectively

simulating the vastness of space.

- **3.2.2 Image-Based Lighting (IBL)** To make the objects feel truly integrated into the environment, the skybox was used as a light source itself. This technique, known as Image Based Lighting, replaces a simple, uniform ambient light with realistic, soft lighting from the surrounding starfield. This was achieved through a one time pre computation step at application startup:
    1. The six sided skybox cubemap is loaded.
    2. A special "convolution" shader processes the skybox, sampling it from many directions for every point on a hemisphere.

3. The result, the total integrated light for any given direction, is baked into a new, low-resolution, blurry cubemap called an **irradiance map**.



- In the final render, the ambient light contribution for any point on a planet's surface is determined by a single, fast texture lookup into this irradiance map using the surface's normal vector. This results in the "dark side" of planets being subtly lit with the correct color and intensity from the part of the skybox they are facing.

3.3 Direct Lighting: An Investigation and Final Model

A significant part of the project involved investigating the best method for rendering direct light from the emissive bodies.

- **3.3.1 Investigation into Physically Based Rendering (PBR)** The initial goal was to implement a full Physically Based Rendering (PBR) model based on the Cook Torrance BRDF. PBR aims to simulate the physical properties of light interaction, using realistic material properties like *metallic* and *roughness*. During implementation, a persistent bug was encountered where the direct lighting component consistently evaluated to zero for non-emissive bodies, resulting in unlit planets. Extensive debugging was performed, including various shader tests (for e.g., visualizing normals & base color) which proved all input data and uniforms were being passed correctly. The conclusion was that a subtle mathematical failure within the BRDF equations was causing the lighting to fail.
- **3.3.2 Final Implementation: The Blinn-Phong Model** A decision to ensure a working and visually correct result, the lighting model was reverted to the classic and reliable **Blinn-Phong** model.

This model, while an approximation of light rather than a physical simulation, is computationally efficient and produces excellent results. It calculates the final color based on three components:

- **Ambient:** Handled by our IBL system.
- **Diffuse:** The base color and shading, calculated from the angle between the light and the surface normal. This creates the "day side" for planets.
- **Specular:** A glossy highlight calculated from the angle between the viewer, the light, and the surface, controlled by a *shininess* property.

3.4 Visual Effects

- **3.4.1 Bloom** To convey the intense energy of the emissive stars, a bloom post-processing effect was implemented. This effect mimics how a real camera lens scatters very bright light.
    1. **Bright Pass:** When the main scene is rendered, a second output texture is generated containing only the pixels whose brightness exceeds a certain threshold (i.e., "brighter than white").
    2. **Blur:** This bright-pass texture is then blurred using a separable Gaussian blur, which is applied iteratively in horizontal and vertical passes using a pair of "ping-pong" framebuffers.
    3. **Composite:** In the final rendering pass, this blurred glow texture is additively blended on top of the main scene, creating a soft, realistic aura around the stars.
- **3.4.2 Particle System** To make the orbital paths visible, a particle system was created. For each body, an emitter spawns a continuous stream of particles.

Each particle is a vertex with properties like position, velocity, and a lifetime. The system is updated on the CPU each frame and rendered efficiently to the GPU as *GL_POINTS*. The vertex shader controls the particle's size over its lifetime, while the fragment shader controls its opacity, causing the trails to fade out smoothly. This visualization is critical for understanding the orbits of the bodies visually and clearly.

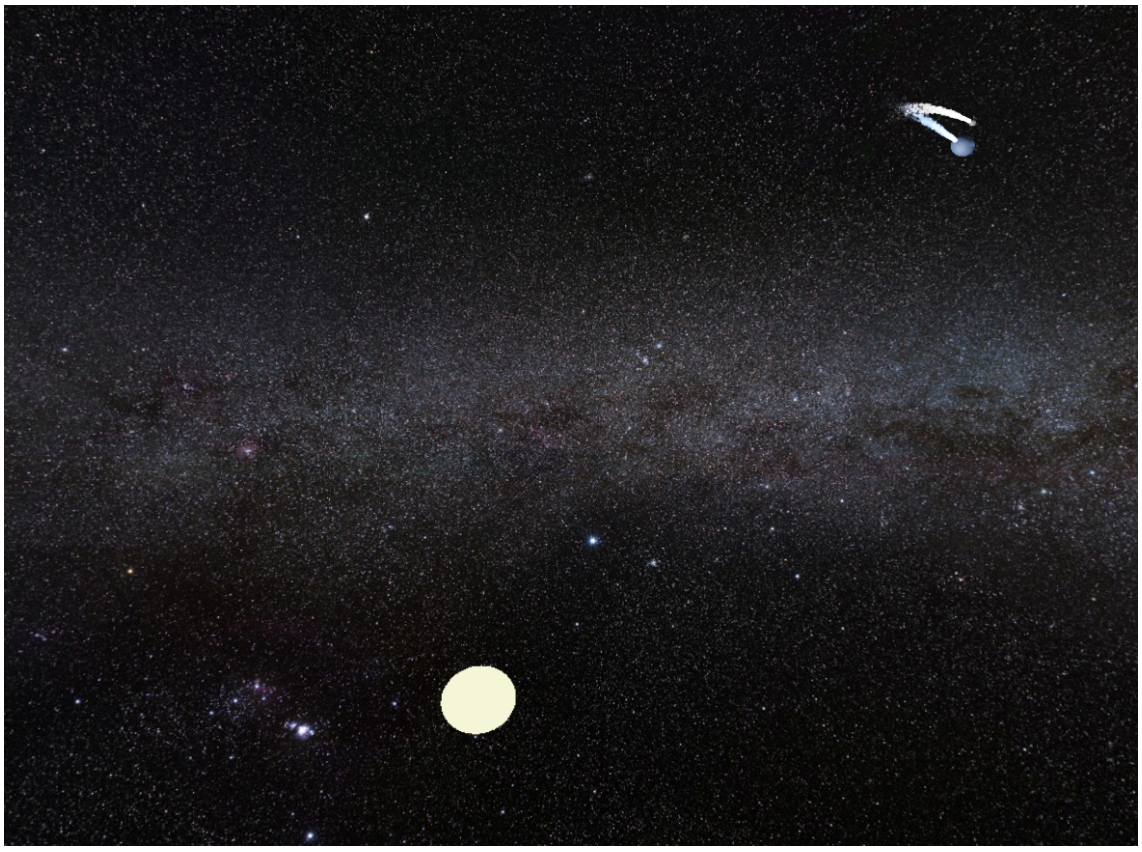
4. Scenarios & Results

The main goal of this project was to create a simulation tool capable of modeling different N-body systems. To demonstrate the engine's physical accuracy and the effectiveness of the visualization techniques, two distinct scenarios were implemented. These scenarios were chosen specifically to highlight the fundamental differences between a stable hierarchical system and a chaotic system.

4.1 The Hierarchical System (Solar System)

This scenario models a gravitationally stable system analogous to a star, a planet, and a moon. It is a "hierarchical" system, meaning the gravitational forces are dominated by a single, massive central body.

- **Initial Conditions:** The system was initialized with one body (the "Sun") possessing a significantly larger mass than the other two. The "Earth" was given an initial velocity to place it in a stable orbit around the Sun, and the "Moon" was given a velocity relative to the Earth to establish its own stable orbit.



- **Results:** As expected, the simulation shows stability. The particle trails trace clear, predictable, and nearly-elliptical orbits. The Moon orbits the Earth, and the combined Earth-Moon system orbits the central Sun in a non-intersecting path. This scenario validates the engine's ability to model the stable, long term orbital mechanics that are characteristic of many realworld systems, including our own solar system.

4.2 The Chaotic System (Figure-Eight Orbit)

This scenario models the classic three-body problem with three bodies of equal mass. While most initial conditions for such a system lead to chaotic and unstable behavior, a few rare, stable solutions exist. This simulation implements one of the most famous: the figure-eight orbit.

- **Initial Conditions:** The three bodies were initialized with equal mass and given the precise, non-intuitive positions and velocities first discovered numerically by C. Moore in 1993.
- **Results:** The simulation successfully reproduces this delicate choreography. The particle trails reveal a beautiful and complex pattern where the three bodies perpetually chase each other along a single, shared, figure-eight-shaped path. Successfully

modeling this intricate and sensitive orbit serves as a powerful validation of the RK4 integrator's high accuracy.

- **Observed Instability:** An important result observed during long-term simulation is the eventual disintegration of the figure-eight pattern. This is not a bug in the physics engine, but rather an accurate demonstration of the orbit's nature as an unstable equilibrium. The mathematical solution can only exist with infinite precision. In a computer simulation, minuscule floating point rounding errors are introduced in every step of the RK4 integration. Because the system is chaotic, these tiny errors will be amplified exponentially over time, eventually acting as a sufficient perturbation to break the delicate gravitational balance. Once this balance is lost, the system rapidly devolves, typically resulting in the ejection of one or more bodies. Observing this eventual collapse is, therefore, a successful outcome, highlighting the practical difference between a perfect mathematical solution and a real world numerical simulation of a chaotic system.

4.3 Demonstrating Chaos: Sensitivity to Initial Conditions

The figure-eight orbit, while periodic, is a chaotic system. This means it exhibits an extreme sensitivity to its initial conditions. To visually demonstrate this, a simple experiment was conducted.

- **The Experiment:** The simulation was run twice. First, with the exact, perfect initial conditions for the figure-eight orbit. Second, an infinitesimally small perturbation was introduced, the starting velocity of a single body was changed by less than 0.01%.
- **Results:** The results are a stark visual demonstration of chaos.
    - **Run 1 (First Condition):** The system remains in a figure-eight pattern indefinitely.
    - **Run 2 (Perturbed Conditions):** For the first few moments, the simulation appears identical to the first. However, the tiny initial error is quickly and exponentially magnified. After a short period, the stable pattern completely disintegrates, and the bodies are thrown into wild, tangled, and unpredictable trajectories.

This experiment successfully visualizes the fundamental difference between a deterministic system (it will always produce the same result from the same input) and a predictable one. The chaotic three-body problem is deterministic but not predictable in the long term, and this visualization makes that abstract concept immediately apparent.

5. The Problem-Solving Process

Beyond the technical implementation, this project represented a significant transition from simply "coding" to trying to engage in a more structured, research oriented development process suitable for academic and professional work. A key personal goal was to adopt and document a formal problem solving methodology, making this report itself an integral part of the learning experience. To that end, the project's development was guided by an iterative "Why, What, How" cycle for each new feature and every significant bug, moving beyond mere implementation to include research, design, and critical analysis.

5.1 The "Why, What, How" Cycle

This cycle is a framework for turning a vague goal into a concrete, well designed feature.

- **Why? (Problem Definition & Research):** This is the foundational step. It begins not with a solution, but with an observation or a goal. Why is the current state insufficient? Why do we need this new feature? This phase involves critical analysis of the project's shortcomings and research into established theories and techniques that could provide a solution.
- **What? (Design & Planning):** This is the architectural phase. Once the "why" is understood, the goal is to design a specific, concrete blueprint. What components are needed? What are the potential approaches and their trade-offs (e.g., performance vs. quality)?
- **How? (Implementation & Iteration):** This is the engineering phase where code is written. This phase is not linear. It involves constant testing, debugging, and analysis of the result. When a bug is found or the result isn't as expected, the process loops back to the "Why?" phase on a smaller scale to debug the specific issue, creating a micro cycle of problem solving.

5.2 Case Study: The Rendering Engine

The evolution of the rendering engine is a prime example of this cycle in action.

- **Why?** The initial simulation, while physically accurate, was visually unconvincing. The uniformly colored spheres looked like simple basic bowling balls and failed to convey a sense of 3D form, material, or realistic lighting. The goal was to create a high fidelity rendering that felt immersive and physically plausible.
- **What?** Research into modern rendering led to the goal of implementing an advanced lighting model. The initial approach was to build a full Physically Based Rendering (PBR) shader based on the Cook- Torrance BRDF. The design included new GLSL shaders, a multi-pass HDR pipeline for post-processing, and new material properties.
- **How & Iteration:** The PBR shader was implemented. However, this led to a persistent and difficult to diagnose bug where non-emissive bodies rendered as unlit. This triggered an extensive iterative debugging phase. Systematic tests were conducted to isolate the point of failure by visualizing intermediate data (normals, base color, light direction).

While these tests proved all input data from the C++ application was correct, the final complex BRDF calculation continued to fail.
Faced with this intractable issue, a pragmatic engineering decision was made to pivot. The **"Why"** was re-evaluated: the core goal was a realistic lighting model, not necessarily a strict PBR implementation. The **"What"** became implementing the classic Blinn-Phong model, but augmenting it with Image-Based Lighting (IBL) to achieve the desired environmental realism. The **"How"** was a successful implementation of this new, more stable system, which finally produced the desired visual result. This journey demonstrated a pragmatic approach to engineering, the importance of systematic debugging, and the ability to adapt a solution when faced with unforeseen complexity.

5.3 Case Study: Visualizing Orbital Dynamics

A simpler example of the cycle was the implementation of the particle trails.

- **Why?** The motion of the bodies was accurate, but their orbital paths were invisible. This made it difficult to appreciate the path & long term dynamics of the system.
- **What?** The designed solution was a dynamic particle system to create trails. The plan was to create a class that would emit particles at each body's position every frame. These particles needed a limited lifetime and should fade out to create a "trail" effect, and the system had to be efficient enough to handle thousands of particles in real time.
- **How?** A *ParticleSystem* class was implemented to manage particle properties. For rendering, the active particles' data is streamed to the GPU each frame and rendered as *GL_POINTS*. The *gl_PointSize* and fragment alpha are controlled within the GLSL shaders to create the size and fade-out effects.

6. Conclusion & Future Work

6.1 Conclusion

This project successfully achieved its primary goal: the design and implementation of a flexible, real time N-body simulation. The final application serves as a tool for visualizing the dynamics of celestial mechanics, bridging the gap between abstract physical laws and intuitive visual results.

The key accomplishments of this project are threefold. First, a high accuracy physics engine was developed, centered on a fourth order Runge Kutta (RK4) integrator. Its stability and physical plausibility will be empirically verified through energy conservation tests. Second, a multi-pass rendering engine was built in OpenGL. This engine leverages a classic Blinn-Phong lighting model, with Image-Based Lighting (IBL) and post-processing effects like bloom, to create a visually rich and immersive environment. Third, the system's modular design was proven effective through the successful simulation of two fundamentally different scenarios: a stable, hierarchical solar system and the chaotic figure-eight three-body problem.

Ultimately, this project was a successful exercise in both scientific simulation and graphical implementation. The development journey, particularly the challenges faced during the investigation of different lighting models, underscored the importance of a structured, iterative problem solving process. The resulting application is not just a physics calculator, but a visual laboratory for exploring the beauty and complexity of gravitational dynamics.

6.2 Future Work

The current simulation provides a strong foundation upon which many exciting features could be built. Future development would focus on enhancing both physical realism and visual fidelity.

- **Revisiting Physically Based Rendering (PBR):** The next logical step in visual fidelity would be to successfully implement a full PBR workflow. While the initial attempt was reverted in favor of the more robust Blinn-Phong model, a renewed effort would allow for a much greater diversity of realistic materials. This would enable the rendering of distinct surfaces like metallic asteroids, icy moons, and gas giants, each with unique roughness and metalness properties that interact with light in a physically plausible way.
- **Procedural Planet Generation:** To move beyond simple, smooth spheres, a procedural generation system for planet surfaces could be implemented. By using noise functions like Perlin or Simplex noise, a height map could be generated for each body. This data could then be used for **normal mapping** to create the illusion of detailed surface features like mountains and craters at a low performance cost, or for a more advanced **tessellation and displacement mapping** approach to generate real, dynamic 3D geometry for the terrain.
- **Scaling the Simulation (N > 3):** The current physics engine uses a direct, brute-force calculation where every body interacts with every other body, a process with O(N2) complexity. To simulate a larger number of bodies (e.g., a star cluster or small galaxy), this would become too slow. Future work could involve researching and implementing a

more efficient, approximate algorithm like the **Barnes-Hut simulation**, which improves performance to O(NlogN) by grouping distant bodies into single nodes.

- **Investigating Advanced Integrators:** While the RK4 method provides excellent accuracy, further research into computational physics could explore other specialized integrators. For example, investigating higher-order **symplectic integrators** would be a valuable academic exercise. These methods are specifically designed to conserve energy and momentum in gravitational systems over extremely long (astronomical) timescales, and comparing their long-term stability against RK4 would be a meaningful extension of the physics engine verification.

7. Appendix: Codebase Architecture

This appendix provides an overview of the project's codebase.

C++ Overview

- **main.cpp - The Orchestrator**
  - **Role:** This is the main entry point and the central hub of the application.
  - **How it Works:** It is responsible for all initialization, the main render loop, and final cleanup.
    - **Initialization:** It initializes GLFW to create a window and OpenGL context, and it initializes GLAD to load the necessary OpenGL functions.
    - **Asset & Object Creation:** It loads all GLSL shader programs into Shader objects and creates the primary Simulation and Camera objects.
    - **IBL Pre-computation:** It runs the one-time, startup process to generate the irradianceMap from the skybox texture. This involves setting up a special framebuffer and using a convolution shader to "bake" the ambient lighting.
    - **The Render Loop:** This is the heart of the program. In each frame, it orchestrates the entire simulation and rendering pipeline in a precise order:
      - It calculates deltaTime for smooth, frame-rate-independent physics and camera movement.
      - It calls processInput to handle user keyboard and mouse events.
      - It runs the physics simulation using a fixed-timestep loop for stability.
      - It executes the multi-pass rendering pipeline: rendering the full 3D scene to an HDR framebuffer, performing a blur pass for the bloom effect, and finally compositing the results to the screen.
- **Simulation.h / .cpp - The Physics Engine**
  - **Role:** This class manages the state of the N-body system and executes the physics calculations.
  - **How it Works:**
    - It contains a std::vector<Body> which holds all the celestial objects in the current scene.
    - The LoadScenario() function clears the current simulation and populates the bodies vector with new initial conditions (mass, position, velocity, etc.) by reading from the scenarioData map in simulation_config.h.
    - The update() method contains the core physics logic. It implements the fourth-order Runge-Kutta (RK4) numerical integrator to accurately calculate the new position and velocity of every body for a given time step, based on their mutual gravitational interactions.
    - It also calls the Emit() function on each body's particle system to generate the visual trails.
- **Body.h / .cpp - The Celestial Object**

    - **Role:** This class represents a single celestial body.
    - **How it Works:**
        - It acts primarily as a data container, storing physical properties (mass, radius, position, velocity) and visual properties (color, isEmissive flag).
        - In its constructor, it calls a private setupSphereMesh() function. This function procedurally generates the vertex data (positions and normals) for a 3D sphere and uploads it to the GPU, storing the resulting VAO and VBO handles. This mesh data is created only once.
        - The draw() function is a simple command called from the main render loop. Its only job is to set the model matrix (which is unique to this body's position) and issue the OpenGL glDrawElements call to render its mesh.
- **ParticleSystem.h / .cpp - The Visual Trail**
    - **Role:** This class manages the glowing trail of particles for a single Body.
    - **How it Works:**
        - It maintains a pre-allocated "pool" of Particle structs to avoid creating and destroying objects every frame.
        - The Emit() function is called by the Simulation. It finds an inactive particle in the pool and "launches" it at the body's current position with an initial velocity and a set lifetime.
        - The Update() function is called every physics step. It iterates through all active particles, updates their position based on their velocity, and decreases their remaining life.
        - The Render() function is called every frame. It builds a fresh vertex buffer containing only the currently active particles and draws them to the screen efficiently using GL_POINTS.
- **Camera.h / .cpp - The Viewer**
    - **Role:** Manages the user's viewpoint and movement within the 3D scene.
    - **How it Works:** It tracks the camera's position, orientation (yaw and pitch angles), and movement speed. Its most important function is GetViewMatrix(), which uses these properties to calculate the **view matrix** via glm::lookAt. This matrix transforms the world from 3D world space into the camera's view space. It also contains ProcessKeyboard() and ProcessMouseMovement() to handle user input.
- **Shader.h - The Shader Utility**
    - **Role:** A helper class that abstracts away the repetitive OpenGL code needed to load, compile, and use GLSL shaders.
    - **How it Works:** The constructor reads shader source code from text files, calls the OpenGL functions to compile the vertex and fragment shaders, checks for compilation errors, and links them into a final shader program. It provides simple helper functions like .use() to activate the shader and .setFloat(), .setVec3(), etc., to send data (uniforms) to the GPU.
- **simulation_config.h - The Central Configuration**
    - **Role:** A single, centralized header file for all simulation parameters, constants, and scenario data.

    - **How it Works:** It uses const variables and a std::map to define global settings (like FIXED_TIMESTEP) and the specific initial conditions (masses, velocities, colors, etc.) for all available scenarios. This design cleanly separates the simulation's data from its logic, making it very easy to tune parameters or add new scenarios without changing the C++ code.

Shaders Overview

- **lighting.vs & lighting.frag:** This is the main shader pair for non-emissive objects. The vertex shader transforms the sphere's vertices and normals into world space. The fragment shader implements the full Blinn-Phong lighting model, using the irradianceMap for ambient light and calculating direct diffuse and specular light from the Sun.
- **emissive.frag:** A very simple fragment shader used for glowing objects like the Sun. It ignores all lighting and simply outputs the object's base color multiplied by an intensity factor, which feeds into the bloom post-processing effect.
- **skybox.vs & skybox.frag:** Renders the 3D cube for the background and samples the cubemapTexture to draw the starfield.
- **particle.vs & particle.frag:** Renders the particles. The vertex shader controls the particle's on-screen size (gl_PointSize), and the fragment shader controls its color and fade-out effect based on its remaining life.
- **blur.frag & bloom_final.frag:** The pair of post-processing shaders that create the bloom effect. blur.frag performs a Gaussian blur, and bloom_final.frag combines the original scene with the blurred result.