**Physics Simulation Engine**

**CONTENTS**

[1. Introduction 2](#_Toc1182709759)

[Team Details 2](#_Toc1545106839)

[Project Overview 2](#_Toc931412774)

[Scope 3](#_Toc664800328)

[2. Objectives 3](#_Toc1745181200)

[3. System Overview 3](#_Toc1034199789)

[Technical Specifications 4](#_Toc787291456)

[Languages: 4](#_Toc415015400)

[Console-Based Command-Line Interface: 4](#_Toc735840634)

[JavaFX Visualization: 4](#_Toc1428688560)

[Backend: 4](#_Toc2123772838)

[Middleware: 5](#_Toc1586463284)

[Input/Output Requirements 5](#_Toc1664860839)

[Input: 5](#_Toc840628410)

[From Console (via Java): 5](#_Toc1238537484)

[From Java to C++: 5](#_Toc1931504611)

[Output: 5](#_Toc1256493685)

[From C++ to Java: 6](#_Toc1906875666)

[From Java to JavaFX: 6](#_Toc1302946309)

[4. Functional Requirements 6](#_Toc186054823)

[Detailed Features 6](#_Toc339323516)

[Object Movement: 6](#_Toc1231268286)

[Gravity Simulation: 6](#_Toc1894213378)

[Friction Simulation: 7](#_Toc516961147)

[Collision Detection: 7](#_Toc1210607322)

[Collision Response: 7](#_Toc1875949927)

[User-Defined Forces: 7](#_Toc1475785797)

[Use Cases 8](#_Toc468284483)

[Adding an Object: 8](#_Toc1648133325)

[Applying Gravity: 8](#_Toc2055319516)

[Collision Between Objects: 8](#_Toc956772399)

[Applying Custom Force: 9](#_Toc1632549679)

[Simulating Friction: 9](#_Toc310917731)

[Pausing and Resetting the Simulation: 9](#_Toc764065785)

[Real-Time Interaction: 9](#_Toc722225132)

[5. Non-Functional Requirements 10](#_Toc1955164546)

[5.1 Performance Requirements 10](#_Toc1146205398)

[5.2 Usability 10](#_Toc324418654)

[6. Development Setup 11](#_Toc1231666365)

[Required Tools 11](#_Toc1518356156)

[Dependencies 11](#_Toc637486396)

[Setup Instructions 11](#_Toc369956073)

[7. UML Diagrams 12](#_Toc662272023)

[8. Workflow 13](#_Toc1632526106)

[Backend (C++ Physics Engine, Java) 13](#_Toc2141196300)

[Middleware (Java with JNI) 14](#_Toc1895954917)

[Frontend (JavaFX GUI and Console-Based Interaction): 14](#_Toc1947281329)

[Real-time Interaction: 15](#_Toc2061972225)

[9. Important Files & Folders 15](#_Toc1936316932)

[10. Testing & Logging 18](#_Toc283375845)

[Testing Strategy 18](#_Toc1551798054)

[Logging Mechanisms 19](#_Toc1395263128)

[11. Conclusion 21](#_Toc1293341611)

## **1. Introduction**

### **Team Details**

|  |  |  |
| --- | --- | --- |
| **Name** | **Roll No** | **Email** |
| Nihit Reddy | IMT2023584 | nihit.reddy@iiitb.ac.in |
| Anish Kantheti | IMT2023536 | kantheti.anish@iiitb.ac.in |
| Satyaram Mangena | IMT2023576 | satyaram.mangena@iiitb.ac.in |
| Undi Trivedh Venkata Sai | IMT2023002 | trivedh.undi@iiitb.ac.in |
| Mitilesh Sai Yechuri | IMT2023507 | mitilesh.sai@iiitb.ac.in |
| Sreekantam Sai Venkat | IMT2023501 | venkat.sreekantam@iiitb.ac.in |

### **Project Overview**

This project is a Physics Simulation Engine that simulates object movement, collisions, and forces in a 2D environment. The system supports real-time interaction through a console-driven command-line interface for simulation configuration and a JavaFX-based GUI for visualization. The purpose of this engine is to provide accurate and real-time simulations for applications such as video games, physics research, and educational tools. The system simulates rigid body dynamics, applies forces, detects collisions between objects, and provides realistic collision responses.

### **Scope**

The simulation will support:

* Rigid body simulation with customizable objects and forces.
* Basic force effects including gravity, friction, and user-defined forces.
* Collision detection for axis-aligned bounding boxes (AABB) and circle colliders.
* Real-time updates of objects in response to forces and collisions.

The system will **not** support:

* Deformable body physics.
* Advanced collision optimization techniques like spatial partitioning (e.g., quadtrees).
* Complex fluid dynamics or particle systems.

## **2. Objectives**

The key objectives of this project include:

* Developing a core physics engine to handle object movement and interaction.
* Implementing rigid body dynamics, including force application and collision response.
* Handling collisions using basic AABB and circle collider techniques.
* Providing a simple interface for users to add and simulate objects in a virtual environment.
* Building a flexible system that can be expanded with additional features like rotation and advanced collision detection in the future.

## **3. System Overview**

### **Technical Specifications**

### **Languages:**

* **C++**: Core physics engine, handling simulation logic such as object movement, force application, and collision detection/response.
* **Java**: Middleware and backend, handling user input, simulation configuration, and communication with the C++ physics engine via Java Native Interface (JNI). Java also forwards real-time simulation data to the JavaFX GUI for visualization.
* **JavaFX**: Provides the graphical user interface (GUI) to visualize the simulation in real time.

### **Console-Based Command-Line Interface:**

* Users interact with the system through a console-based interface powered by Java.
* Key features include:
  + Selecting simulation modes (e.g., Block Simulation or Collision Simulation).
  + Inputting simulation parameters such as object properties, forces, and environment settings.
  + Starting, pausing, or resetting the simulation.

### **JavaFX Visualization:**

* The GUI will visualize the simulation dynamically, showing:
* Object movements, forces, and collisions in real time.
* User-applied actions like toggling gravity or applying custom forces.
* Controls for viewing the simulation, such as play, pause, and clear, are integrated directly into the visualization window.

### **Backend:**

* The core simulation logic is implemented in C++, responsible for:
* **Object Management**: Tracking properties like position, velocity, mass, and shape.
* **Force Calculations**: Handling various forces such as gravity, friction, and user-defined inputs.
* **Collision Detection and Response**: Detecting AABB and circle collider collisions and calculating the resulting changes in motion and properties.
* The C++ engine runs independently of the user interface, receiving input and sending output through the Java middleware.

### **Middleware:**

The Java middleware leverages JNI for seamless communication between Java and C++.

* Input Handling: Takes user-provided parameters from the console and passes them to the C++ engine.
* Simulation Updates: Relays real-time simulation data from C++ to the JavaFX visualization for rendering.
* Command Handling: Synchronizes user commands (e.g., play, pause, clear) with the simulation state in the physics engine.

### **Input/Output Requirements**

### **Input:**

#### From Console (via Java):

* **Object Parameters:**
  + Position, velocity, mass, and shape entered by the user through the command-line interface.
* **Force Parameters:**
  + Gravity, friction, and user-defined forces specified during simulation setup.
* **Simulation Control Parameters:**
  + Commands for starting, pausing, or resetting the simulation.
  + Toggles for enabling/disabling gravity or applying specific forces.

#### From Java to C++:

* The Java middleware will package input parameters from the console and send them to the C++ physics engine via JNI.
* Parameters include object properties, force settings, and simulation controls for processing in the physics engine.

### **Output:**

#### From C++ to Java:

* **Simulation Data:**
  + Updated positions, velocities, and states of objects after each simulation step.
* **Collision Information:**
  + Data about collisions, including the objects involved and their updated states (e.g., velocities, positions).

#### From Java to JavaFX:

* Real-time simulation updates are sent to the JavaFX GUI for visualization:
  + **Graphical Visualization:**
    - Animations showing object movements, forces, and collision interactions.
    - Dynamic rendering of the simulation environment based on the C++ backend's output.
  + **Interactive Controls:**
    - Reflect changes like toggled gravity or paused/resumed simulations directly in the visualization.

## **4. Functional Requirements**

### **Detailed Features**

#### Object Movement:

* The system continuously updates each object’s position, velocity, and acceleration based on forces acting on it, including user-defined forces (applied via the console) and internal forces like gravity and friction.
* Objects have properties such as mass, initial velocity, and initial position, which determine their movement.
* The physics engine, built in C++, uses a time-stepping loop to update the state of each object at regular intervals.
* The JavaFX GUI visualizes these real-time updates, reflecting the calculated movements and interactions from the backend.

#### Gravity Simulation:

* Gravity is applied globally to all objects in the scene unless toggled off by the user through the command-line interface.
* Objects experience a constant downward force proportional to their mass, based on a user-defined or default gravitational constant (e.g., 9.8 m/s² for Earth's gravity).
* Users can toggle gravity on/off via console commands, and the system will apply or remove this force accordingly in real time.

#### Friction Simulation:

* Friction opposes the motion of objects sliding or moving across surfaces, dependent on the object's velocity and surface type.
* **Two types of friction are simulated:**
  + **Static Friction:** Prevents motion when an object is at rest until an applied force overcomes a threshold.
  + **Kinetic Friction:** Reduces velocity when objects are in motion.
* Users can input the friction coefficient for different materials or objects in the console, and the simulation dynamically adjusts object behavior in response.

#### Collision Detection:

* **AABB Collider:** Handles collisions for rectangular or box-like objects by checking if their boundaries overlap.
* **Circle Collider:** Handles collisions for round objects by checking if the distance between their centers is less than the sum of their radii.
* These methods ensure realistic object interactions. Collision results are displayed in real-time through the JavaFX GUI.

#### Collision Response:

* Upon detecting a collision, the system calculates reactions based on physics principles such as conservation of momentum.
* Objects may bounce off each other or lose energy in inelastic collisions if damping is applied.
* Updated velocities, positions, and collision states are visualized in the JavaFX GUI in real time.

#### User-Defined Forces:

* Users can apply custom forces to specific objects during runtime, such as pushing, pulling, or accelerating them in a chosen direction.
* Through the command-line interface, users select an object and input a force’s magnitude and direction, dynamically affecting its movement.
* These forces influence the object’s trajectory or speed until overridden by other forces like gravity or friction, with real-time updates reflected in the GUI.

### **Use Cases**

#### Adding an Object:

User Action:

The user adds an object to the simulation via the console, specifying parameters such as position, velocity, mass, and shape.

System Response:

* The Java middleware sends these parameters to the C++ engine.
* The physics engine initializes the object and places it in the simulation environment.
* The object’s state is continuously updated based on applied forces, and its movement is visualized in real-time in the JavaFX GUI.

#### Applying Gravity:

User Action:

* The user toggles gravity via a console command for all or specific objects.

System Response:

* The system updates the simulation by applying gravitational force to the relevant objects.
* Objects accelerate downward, with the results visualized in real time in the JavaFX GUI.
* If collisions occur (e.g., with the ground or other objects), the system handles collision detection and response.

#### Collision Between Objects:

User Action:

* Two objects move toward each other due to applied forces or initial velocity.

System Response:

* The system detects the collision using AABB or Circle Collider methods.
* Collision response is calculated based on the mass and velocity of each object.
* The updated velocities and positions are displayed in real time via the JavaFX GUI.

#### Applying Custom Force:

User Action:

* The user selects an object and applies a custom force through console inputs, specifying magnitude and direction.

System Response:

* The Java middleware sends the input to the C++ engine.
* The engine updates the object’s trajectory, and the effect is visualized in real time in the JavaFX GUI.

#### Simulating Friction:

User Action:

* The user enables friction or adjusts the friction coefficient via the console.

System Response:

* The system applies friction to slow down objects based on their velocity and surface type.
* The effects are visualized in the JavaFX GUI as objects decelerate over time in the simulation.

#### Pausing and Resetting the Simulation:

User Action:

* The user pauses or resets the simulation using console commands.

System Response:

* On pause, the system halts all object movement and force calculations.
* On reset, objects return to their initial states, and the simulation starts over.

#### Real-Time Interaction:

User Action:

* The user interacts with objects by selecting them in the console to apply forces or alter parameters during the simulation.

System Response:

* Changes are applied instantly to the simulation, updating the object’s state accordingly.
* The JavaFX GUI reflects these changes in real-time, showing updated movement or force application.

## **5. Non-Functional Requirements**

### 5.1 Performance Requirements

1. **Real-Time Simulation:**

The engine will maintain at least **60 FPS** for typical use cases with up to **50 objects**, ensuring smooth real-time performance. The simulation will be visually updated in real time via the JavaFX GUI.

1. **Low Latency:**

User interactions (e.g., adding objects, applying forces) through the **console** or JavaFX interface should have a response time of under **100 milliseconds**. Communication between the **console, Java middleware, and C++ engine** will be optimized to ensure minimal latency.

1. **Efficient Collision Detection:**

The C++ engine will handle up to **100 active colliders** without noticeable performance degradation. The JavaFX GUI will efficiently reflect these collisions visually without slowing down.

1. **Scalability:**

The engine will allow users to adjust simulation parameters, such as the number of active objects or the simulation’s time step, to handle larger object counts. Performance will be maintained by optimizing force and collision calculations in the C++ backend.

### 5.2 Usability

1. **Simple Interface:**

The **console** will offer an easy-to-use text-based input system to set up simulations and parameters interactively. The **JavaFX GUI** will provide a visual interface with intuitive controls for visualizing the simulation in real time. Preset options for quick setup and default scenarios will also be available.

1. **Error Handling:**

Clear and user-friendly error messages will be displayed for incorrect inputs via the **console**. Validation will occur in Java before sending commands to the C++ backend, ensuring that only valid parameters are processed.

1. **Interactive Controls:**

Users can adjust simulation settings in real time through **console commands** or the **JavaFX GUI**. Features like pausing, resuming, resetting, and adjusting forces or gravity will be accessible through simple console inputs or buttons/sliders in the GUI.

## **6. Development Setup**

### **Required Tools**

* **C++ Compiler:** GCC or Clang.
* **Java Development Kit (JDK):** Java SE Development Kit (JDK 11 or higher).
* **JavaFX SDK:** Required for building the GUI.
* **JNI Setup:** To connect Java with the C++ simulation engine.
* **IDE:** Visual Studio Code (for C++), or any IDE supporting these languages.

### **Dependencies**

* **JavaFX Libraries:** For creating the GUI interface in Java.
* **JNI:** To facilitate communication between Java middleware and the C++ backend.
* **Standard Template Library (STL):** Utilized in the C++ engine.

### **Setup Instructions**

1. **Clone the Repository**:

git clone https://github.com/ani3h/physics-engine.git

cd physics-engine

1. **Set Up C++ Environment**:

Navigate to the src/cpp/ directory:

cd src/cpp

Compile the C++ physics engine:

g++ -fPIC -shared -o ../../lib/libPhysicsEngine.so main.cpp object.cpp physics\_world.cpp collider.cpp

1. **Set Up Java Environment**:

Navigate to the src/java/ directory:

cd ../../src/java

Compile the Java middleware and JNI integration:

javac -d . PhysicsSimulation.java jni/PhysicsEngineJNI.java

If the JNI header (jni.h) is not already generated, run:

javah -jni jni.PhysicsEngineJNI

1. **Setup JavaFX GUI:**

Ensure the JavaFX SDK is downloaded and properly configured.

Run the JavaFX application with the correct --module-path for the JavaFX SDK:

java --module-path /path/to/javafx-sdk/lib --add-modules javafx.controls,javafx.fxml -cp . PhysicsSimulation

1. **Run the Simulation:**

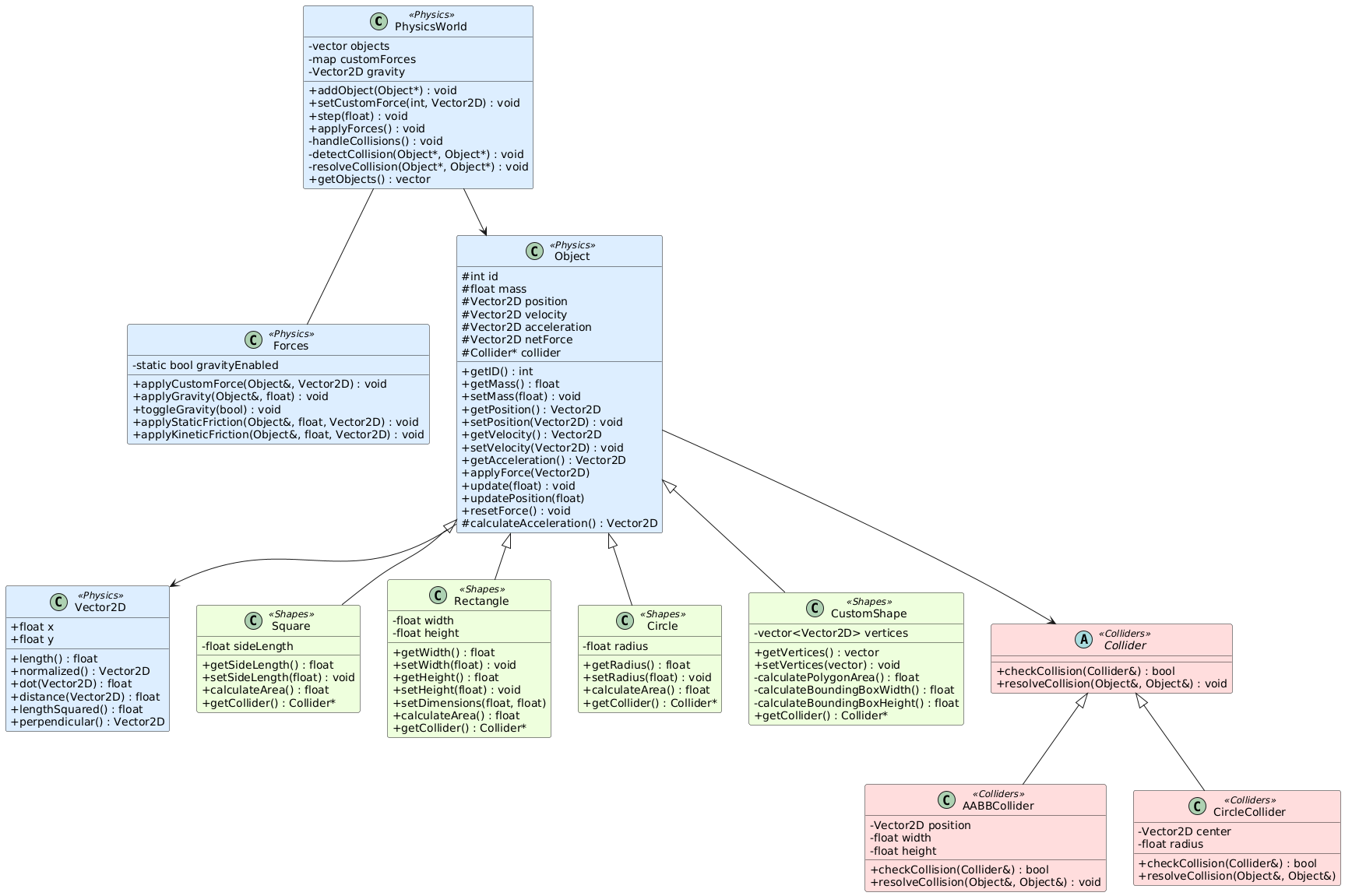
Ensure the compiled libPhysicsEngine.so file is in the correct library path:

export LD\_LIBRARY\_PATH=../../lib:$LD\_LIBRARY\_PATH

Execute the Java middleware to connect the **console-based input** or **JavaFX GUI** with the C++ physics engine:

java PhysicsSimulation

## **7. UML Diagrams**



## **8. Workflow**

### **Backend (C++ Physics Engine, Java)**

1. Object Management:

The C++ engine manages the core simulation, including:

* Object creation, with properties such as position, velocity, mass, and shape.
* Real-time updates to object states based on forces (gravity, friction, user-defined).

**2. Collision Handling:**

Collision detection and response are handled entirely in C++.

* Algorithms like AABB for rectangles and radius-based checks for circles detect collisions.
* Responses (e.g., velocity changes, positional adjustments) follow physics principles like momentum conservation.

**3. JNI Communication:**

The C++ functions are exposed to Java via JNI, enabling:

* Simulation control (start, pause, reset) from Java.
* Updates to object states and responses to collision events.

### **Middleware (Java with JNI)**

1. Java Middleware:

Acts as the intermediary between the console commands (or JavaFX GUI) and the C++ physics engine:

* Receives simulation parameters or commands (e.g., add object, apply force) via the console or GUI.
* Invokes the C++ physics engine functions via JNI.
* Processes simulation results (updated positions, velocities, collisions) and forwards them to the JavaFX GUI for rendering.

2. JavaFX GUI:

A user-friendly visualization of the simulation:

* Displays objects moving and interacting in real time based on C++ engine calculations.
* Allows users to pause, reset, or apply forces via interactive GUI elements (buttons, sliders).

3. JNI Integration:

The Java layer uses JNI to:

* Send object properties and force parameters to C++ for computation.
* Receive updated simulation states for rendering in the JavaFX GUI.

### **Frontend (JavaFX GUI and Console-Based Interaction):**

1. User Interface (JavaFX):

JavaFX replaces the web-based Next.js GUI for:

* Adding objects to the simulation (position, velocity, mass, shape).
* Configuring simulation parameters like gravity and friction via sliders or checkboxes.
* Applying user-defined forces through interactive input elements.

2. Real-time Rendering:

* JavaFX visualizes object movement, forces, and collisions using JavaFX Canvas or Scene Graphs.
* Simulation updates from the C++ backend are rendered in real time, ensuring smooth interactions.

### **Real-time Interaction:**

1. User Input (Console/JavaFX):

* Users add objects, apply forces, or adjust simulation parameters.
* Input can be provided via a console-driven interface or directly in the JavaFX GUI.

1. Request (Java Middleware):

* The Java layer receives the input and formats it into calls to the C++ engine via JNI.

1. Physics Calculation (C++):

* The C++ physics engine processes the input:
* Updates object states based on applied forces.
* Detects and resolves collisions.

1. Update Backend (C++ to Java):

* The C++ engine sends updated object states (position, velocity, collisions) back to Java.

1. Response to Frontend (JavaFX):

* Java forwards the updated simulation states to the JavaFX GUI for rendering.

1. Real-time Rendering (JavaFX):

* The JavaFX GUI updates visuals in real time, ensuring smooth animations and immediate feedback on user interactions.

This implementation streamlines communication, optimizes performance with native C++ physics calculations, and delivers a responsive and interactive simulation experience.

## **9. Important Files & Folders**

docs/

* Contains project documentation, including:
* High-level design and implementation details of the Physics Simulation Engine.
* Instructions for setting up the JavaFX GUI, Java middleware, and JNI integration with the C++ backend.

include/jni.h

* The Java Native Interface (JNI) header file generated from Java class definitions.
* Facilitates communication between the Java middleware and the C++ physics engine.
* Essential for invoking C++ functions from the Java layer.

lib/libPhysicsEngine.so

* The shared library for the physics engine, compiled from C++ source files.
* Used by the Java middleware via JNI for executing core simulation logic like force application and collision handling.

resources/temp/

* Directory for resource files, including:
* Configuration files for simulation settings.
* Temporary or cache files created during simulations.
* JSON files for storing object properties or user-defined simulation parameters.

src/cpp/

1. main.cpp

* The main entry point for the C++ physics engine.
* Initializes the simulation environment, processes updates to object states, and handles the physics loop.

1. object.cpp/.h

* Defines the base object class and derived classes for physical entities (e.g., squares, rectangles, circles).
* Manages attributes such as position, velocity, mass, and force application.

1. physics\_world.cpp/.h

* Implements the physics simulation world, handling:
* Application of forces (gravity, friction, custom).
* Integration of motion and interaction between objects.

1. collider.cpp/.h

* Contains collision detection and resolution logic:
* AABB and circle collision methods.
* Support for expanding to other shapes like polygons in the future.

src/java/

PhysicsSimulation.java

* The main entry point for the Java middleware.
* Acts as a bridge between user input (from console or JavaFX GUI) and the physics engine via JNI.
* Manages the flow of simulation data:
* Sends user inputs to the C++ engine.
* Retrieves updated simulation states for rendering in JavaFX.

jni/PhysicsEngineJNI.java

* A Java class wrapping native C++ methods.
* Enables Java to call the C++ functions for managing object interactions, force applications, and collision handling.

src/frontend/ (JavaFX)

JavaFX GUI:

Main.java: Entry point for the JavaFX-based simulation visualization.

controllers/:

* Contains JavaFX controller classes managing user interactions like adding objects, toggling gravity, or applying forces.

views/:

* FXML files defining the structure of the GUI, such as layouts for simulation controls and object visualization.

tests/

Unit and integration tests for:

* The C++ physics engine (e.g., collision detection, force application).
* Java middleware and JNI communication.
* JavaFX GUI functionality (e.g., user interactions, simulation rendering).

Ensures robust integration between components and validation of simulation results.

.gitignore

Specifies files and directories to exclude from version control, including:

* Compiled libraries (e.g., .so, .dll files).
* Temporary or intermediate build files for C++, Java, and JavaFX.
* Logs and other runtime-generated files.

README.md

* The primary project documentation file.
* Includes setup instructions for:
* Compiling and running the C++ physics engine.
* Configuring and using the Java middleware and JavaFX GUI.
* Details on simulation features, file structure, and common troubleshooting steps.

## **10. Testing & Logging**

### **Testing Strategy**

1. **Unit Testing (C++):**

* All classes in the C++ physics engine, including **Object**, **Collider**, and **PhysicsWorld**, will have dedicated unit tests using a framework like **Google Test**.
* Core functionalities to be verified include:
  + Accurate force application (gravity, friction, and user-defined forces).
  + Object state updates (position, velocity, acceleration).
  + Collision detection and resolution algorithms for AABB and circles.

1. **Unit Testing (Java Middleware):**

* The Java middleware interacting with the C++ backend via JNI will undergo rigorous unit tests.
* Focus areas:
  + Correct invocation of JNI calls to ensure seamless communication between Java and C++.
  + Validation of inputs sent from Java to the C++ backend (e.g., object parameters, simulation settings).
  + Handling of exceptions during JNI calls and accurate propagation of error messages.

1. Integration Testing (Next.js Frontend, Java, & C++):

* Integration tests will ensure that all layers—JavaFX frontend, Java middleware, and the C++ backend—communicate seamlessly.
* Scenarios to test:
  + Adding objects with varying properties to the simulation.
  + Applying forces and toggling physics settings like gravity or friction.
  + Collision handling and object response validation.
  + Simulation controls such as Play, Pause, and Reset.
* Testing will confirm that the JavaFX GUI accurately reflects real-time updates from the backend.

1. Performance Testing:

* Ensures that the simulation maintains at least **60 FPS** for typical scenarios with up to 50 objects.
* Stress tests will simulate:
  + Real-time updates for large object counts (100+).
  + Handling of simultaneous collisions and dynamic forces.
* Performance metrics like FPS, memory usage, and CPU load will be monitored during tests.

1. **User Interface Testing (JavaFX):**

* The JavaFX interface will be tested for usability and responsiveness.
* Key tests include:
  + Accurate rendering of objects and forces based on backend updates.
  + Real-time responsiveness to user interactions such as adding objects, applying forces, and toggling settings.
  + Compatibility testing across different screen sizes and resolutions.

### **Logging Mechanisms**

**1. Error Logging:**

* C++ Backend:
  + Logs simulation errors, such as invalid object states, failed collision calculations, or force application errors, into a dedicated file.
  + Exception handling mechanisms will write detailed stack traces to assist debugging.
* Java Middleware:
  + Logs JNI-specific errors, including:
  + Failed function calls to the C++ backend.
  + Invalid responses or corrupted data received from the physics engine.
  + Middleware errors will be tagged and timestamped for traceability.
* JavaFX Frontend:
  + Logs issues related to GUI rendering, simulation control errors, or invalid user inputs.
  + Errors will appear in a user-friendly message while being recorded in detailed logs for debugging.

**2. Simulation Logging:**

* Optional logging for tracking simulation states, including:
  + Object properties like position, velocity, and applied forces over time.
  + Collision events, including object IDs and resolved velocities.
  + Logs can be toggled on or off for debugging or performance monitoring.

3. API Call Logging (JavaFX to Java Middleware):

* Logs all API requests between the JavaFX interface and the Java middleware, covering:
  + Object creation and simulation parameter updates.
  + Commands like play, pause, reset, and force application.
* Logs include request payloads, response times, and error details for failed calls.

**4. Performance Logging:**

* Tracks real-time metrics, including:
  + Frames per second (FPS).
  + Memory and CPU usage of the simulation engine.
  + Latency between JavaFX user actions and updates received from the C++ backend.
* Performance logs will help fine-tune system efficiency and identify bottlenecks during stress testing.

## **11. Conclusion**

By integrating JavaFX, Java, and C++, the physics simulation engine achieves a blend of high-performance computation and an interactive, user-friendly interface. The C++ backend serves as the core, delivering efficient real-time physics calculations for object movements, forces, and collisions. The Java middleware, enabled by the Java Native Interface (JNI), acts as a bridge, ensuring seamless communication between the backend and the frontend while managing user inputs and simulation updates. The JavaFX frontend provides an intuitive and visually engaging interface, allowing users to interact with the simulation dynamically by placing objects, applying forces, and adjusting simulation parameters.

This layered architecture combines computational efficiency, interactivity, and real-time performance, offering a scalable and flexible platform for physics simulations. The system’s modular design ensures that each component is optimized for its role, making the simulation engine suitable for applications in education, gaming, and research.