## **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 **web-based GUI**. The purpose of this engine is to provide accurate and real-time simulation for applications like video games, physics research, or educational tools. The system will simulate rigid body dynamics, handle force application, detect collisions between objects, and provide 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 between the frontend and the C++ physics engine. Java will handle communication between the Next.js frontend and the C++ engine via Java Native Interface (JNI). Java will forward input commands from the frontend and retrieve output from the C++ engine.
* **Next.js (JavaScript):** Frontend framework for building the graphical user interface (GUI) on a localhost server, allowing users to interact with the simulation.

### **Frontend:**

* **Next.js-based GUI:** The frontend will be developed using **Next.js** to provide a real-time user interface where users can:
  + Place objects in the simulation environment (input object properties like position, velocity, mass, shape, etc.).
  + Apply forces to objects (e.g., gravity, friction, or user-defined forces).
  + Toggle gravity on or off.
  + Control the simulation with **Play**, **Pause**, and **Clear** buttons.
  + Visualize the object movements, collisions, and interactions dynamically in a graphical format.

### **Backend:**

* The core simulation logic will be implemented in C++, responsible for:
* Object Management: Tracking object properties such as position, velocity, mass, and shape.
* Force Calculations: Handling various forces like gravity, friction, and custom forces input by the user.
* Collision Detection/Response: Managing object collisions and calculating the resulting changes in velocity, position, and other properties post-collision.
* The C++ engine will operate independently of the GUI and only receive input/output requests through the Java middleware, which ensures that the two languages (C++ and Java) are properly connected via JNI.

### **Middleware:**

The middleware for this project will be **Java Native Interface (JNI)**, enabling communication between Java and C++. Java will serve as the intermediary, passing simulation commands from the Next.js frontend to the C++ physics engine and relaying the simulation results back to the frontend for visualization.

### **Input/Output Requirements**

### **Input:**

* **From Next.js (via Java):**
  + Object parameters (position, velocity, mass, shape, etc.) that users can input through the Next.js GUI.
  + Force parameters such as gravity, friction, and user-defined forces.
  + Simulation control parameters (time steps, start/stop commands, toggling gravity).
* **From C++ (via Java):**
  + The C++ physics engine will handle simulation inputs such as object properties and force applications passed through JNI calls from Java.

### **Output:**

* **From C++ to Next.js (via Java):**
  + Updated positions, velocities, and states of objects after every simulation step.
  + Collision information and response data (e.g., updated velocities and positions after collisions).
* Java will pass these outputs to the Next.js frontend for real-time visualization:
* **Graphical Interface:** The Next.js frontend will dynamically display the results of the simulation, including object movements, forces acting on them, and collision responses.
* **Console Outputs (Optional):** For debugging or detailed simulation logs, outputs can be displayed in the backend.

## **4. Functional Requirements**

### **Detailed Features**

#### **Object Movement:**

1. The system continuously updates each object’s position, velocity, and acceleration based on forces acting on it, including user-defined forces (applied through the Next.js frontend) and internal forces like gravity and friction.
2. Objects have properties such as mass, initial velocity, and initial position, which determine their movement.
3. The physics engine, built in **C++**, uses a time-stepping loop to update the state of each object at regular intervals (e.g., each frame of the simulation).
4. The **Next.js GUI** allows users to place objects, specify these parameters, and visualize real-time updates.

#### **Gravity Simulation:**

1. Gravity is applied globally to all objects in the scene unless toggled off by the user through the frontend.
2. 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).
3. Users can toggle gravity on/off via the **Next.js frontend**, and the system will update the simulation to apply or remove this force accordingly.

#### **Friction Simulation:**

1. Friction opposes the motion of objects sliding or moving across surfaces, and it is dependent on the object's velocity and surface type.
2. Two types of friction are simulated:
   1. **Static Friction:** Prevents motion when an object is at rest until an applied force overcomes a threshold.
   2. **Kinetic Friction:** Reduces velocity when objects are in motion.
3. Users can adjust the friction coefficient in the frontend for different materials or objects, and the simulation will respond by slowing down objects accordingly.

#### **Collision Detection:**

1. **AABB Collider:** Used for rectangular or box-like objects; the system checks if their boundaries overlap, indicating a collision.
2. **Circle Collider:** For round or spherical objects, the system checks if the distance between their centers is less than the sum of their radii.
3. These collision methods ensure that objects interact realistically in the environment. Users can see the results of collisions in real-time through the **Next.js frontend**.

#### **Collision Response:**

1. After detecting a collision, the system applies the laws of physics (e.g., conservation of momentum) to calculate how objects react.
2. Objects may bounce off each other or, if damping is applied, lose energy in inelastic collisions.
3. The updated velocities and positions of the objects are displayed in the frontend in real-time.

#### **User-defined Forces:**

1. Users can apply custom forces to specific objects during runtime (e.g., pushing, pulling, or accelerating them in a chosen direction).
2. Through the **Next.js frontend**, users can select an object and input a force’s magnitude and direction, affecting the object’s movement.
3. These forces dynamically change the object’s trajectory or speed until stopped or overridden by other forces like gravity or friction.

### **Use Cases**

#### **Adding an Object:**

1. **User Action:** The user adds an object to the simulation via the Next.js GUI, specifying parameters such as position, velocity, mass, and shape.
2. **System Response:**

* The system, via 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 frontend.

#### **Applying Gravity:**

1. **User Action:** The user toggles gravity in the frontend for all or specific objects.
2. **System Response:**

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

#### **Collision Between Objects:**

1. **User Action:** Two objects move toward each other due to applied forces or initial velocity.
2. **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 frontend.

#### **Applying Custom Force:**

1. **User Action:** The user selects an object and applies a custom force through the Next.js GUI, specifying magnitude and direction.
2. **System Response:**

* The system, via Java, sends the input to the C++ engine.
* The engine updates the object’s trajectory, and the effect is visualized in the frontend.

#### **Simulating Friction:**

1. **User Action:** The user enables friction or adjusts the friction coefficient through the frontend.
2. **System Response:**

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

#### **Pausing and Resetting the Simulation:**

1. **User Action:** The user pauses or resets the simulation using buttons in the frontend.
2. **System Response:**

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

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

1. **User Action:** The user interacts with objects by dragging or clicking to apply forces via the **Next.js** interface.
2. **System Response:**

* Changes are instantly applied to the simulation, and the object’s state is updated accordingly.
* The GUI reflects the changes in real-time, showing the 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 **Next.js frontend**.
2. **Low Latency:** User interactions (e.g., adding objects, applying forces) via the **Next.js interface** should have a response time of under 100 milliseconds. Communication between the **Next.js frontend**, **Java middleware**, and **C++ engine** will be optimized to ensure minimal latency.
3. **Efficient Collision Detection:** The C++ engine will handle up to 100 active colliders without noticeable performance degradation. The **Next.js frontend** will efficiently reflect these collisions visually without slowing down.
4. **Scalability:** The engine should handle larger object counts by allowing users to adjust simulation parameters such as the number of active objects or the simulation’s time step. The system should maintain performance even with higher object counts by optimizing force and collision calculations in the **C++ backend**.

### 5.2 Usability

1. **Simple Interface:** The **Next.js frontend** will offer a user-friendly, web-based interface with intuitive controls. Users will be able to interact with the simulation easily, using drag-and-drop functionality, buttons for adding forces, and toggling settings (e.g., gravity, friction). Preset options will be provided for easy interaction and quick setup.
2. **Error Handling:** Clear and user-friendly error messages will be displayed for incorrect inputs via the **Next.js interface**. Errors (such as invalid object parameters) will be validated in the frontend before sending requests to the backend.
3. **Interactive Controls:** Users can adjust simulation settings in real time through the **Next.js GUI** without restarting the simulation. Features like pausing, resuming, resetting, and adjusting forces or gravity will be accessible through simple buttons and 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).
* **Node.js:** Required for running the Next.js frontend (v14 or higher).
* **JNI Setup:** Required to connect Java with the C++ simulation engine.
* **IDE:** Visual Studio Code (recommended for Next.js and C++), IntelliJ IDEA (for Java), or any IDE that supports these languages.

### **Dependencies**

* **SDL or OpenGL libraries** for C++ (optional for graphical rendering, if required).
* **Next.js:** Web-based frontend for interacting with the simulation.
* **React** and **GSAP** for animations and frontend interactivity in the Next.js project.
* **JNI:** To facilitate communication between the Java middleware and the C++ backend.

### **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 frontend and JNI integration:

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

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

javah -jni jni.PhysicsEngineJNI

1. **Setup Next.js Frontend:**

Navigate to the src/frontend/ directory (for the Next.js frontend):

cd ../../src/frontend

Install the necessary dependencies for the Next.js application:

npm install

1. **Run the Next.js Frontend:**

After setting up the frontend, run the Next.js development server:

npm run dev

This will launch the frontend interface on a local server, typically accessible at <http://localhost:3000>

1. Run the Simulation (Integrating C++ and Java)**:**

Ensure that the compiled libPhysicsEngine.so is available in your library path:

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

Execute the Java middleware, which will handle interaction between the **Next.js frontend** and the **C++ physics engine** via JNI:

java PhysicsSimulation

1. Interaction Flow:

* The Next.js frontend sends requests to the Java middleware via REST or WebSockets.
* The Java middleware communicates with the C++ physics engine through JNI for handling the simulation logic.
* Simulation results are displayed in real time on the Next.js interface.

## **7. Workflow**

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

1. **Object Management:**

* The C++ engine will handle core simulation logic, including object creation, state updates (position, velocity, mass, etc.), and force application (gravity, friction, and user-defined forces).

1. **Collision Handling:**

* The C++ engine will manage collision detection and response, determining how objects interact and resolve collisions based on physics principles like momentum and force.

1. **JNI Communication:**

* The C++ functions will be exposed to Java through JNI, allowing the Java layer to invoke the physics engine for processing simulation updates and managing object states.

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

1. **Java Middleware:**

Java will act as the bridge between the **Next.js frontend** and the **C++ backend**, managing simulation inputs and outputs. It will:

* Receive user inputs (e.g., object parameters, forces) from the frontend.
* Call corresponding C++ functions via JNI to process these inputs within the physics engine.
* Retrieve updated object states and simulation results from the C++ backend.

1. **JNI Integration:**

* JNI (Java Native Interface) will be used to bridge the communication between Java and C++, allowing Java to invoke C++ methods for handling object movements, collisions, and force applications.

### **Frontend (Next.js)**

1. **User Interface:**

The **Next.js frontend** will allow users to:

* Add objects to the simulation environment by specifying properties like position, velocity, mass, and shape (e.g., rectangles or circles).
* Configure simulation parameters such as gravity, friction, and custom forces.
* Control the simulation (e.g., play, pause, reset) and apply user-defined forces to objects.
* Visualize the simulation in real time, showing object movement and interactions based on the C++ physics engine's calculations.

1. **Communication with Java Middleware:**

* The Next.js frontend will send user input (e.g., object creation, force application) to the Java middleware via REST APIs or WebSockets.
* The frontend will also receive real-time simulation updates from the Java middleware and reflect these updates in the user interface (using React and GSAP for animations).

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

1. **Frontend (Next.js):**

* The user interacts with the Next.js UI to add objects, modify parameters (e.g., gravity), and start/pause the simulation.
* User actions trigger requests sent to the Java middleware.

1. **Middleware (Java):**

* The Java layer receives these requests, processes the input, and communicates with the C++ engine via JNI.
* It handles the execution of the simulation by invoking C++ functions.

1. **Backend (C++):**

* The C++ engine computes object movements, force applications, and collision responses.
* The simulation state is updated continuously as per the time steps, and the results are returned to the Java middleware.

1. **Return Flow:**

* The Java middleware sends updated simulation states (e.g., object positions, velocities) back to the Next.js frontend for real-time visualization.

## **8. Important Files & Folders**

* docs/

Contains project documentation, including high-level design, implementation details, and an overview of the Physics Simulation Engine, as well as instructions on setting up the Next.js frontend and Java-JNI integration.

* include/jni.h

The generated Java Native Interface (JNI) header file. It facilitates communication between the Java middleware and the C++ physics engine. This file is essential for calling C++ functions from the Java side.

* lib/libPhysicsEngine.so

The shared library for the physics engine, compiled from the C++ source code. This file is linked with the Java middleware via JNI for executing core physics simulation functionality.

* resources/temp/

Directory for resource files such as configuration files, images, or temporary data required during the simulation. Could also contain JSON files for storing object or simulation parameters.

* src/cpp/main.cpp

The main entry point for the C++ physics engine. This file initializes the physics world and runs the core simulation loop, managing object states, forces, and collision detection.

* src/cpp/object.cpp/.h

Implements the object properties and methods, defining the behavior and attributes of physical objects in the simulation (e.g., position, velocity, mass, and shape).

* src/cpp/physics\_world.cpp/.h

Contains the logic for the physics simulation, handling forces (gravity, friction, and custom forces), motions, and interactions between objects in the physics world.

* src/cpp/collider.cpp/.h

Handles collision detection between objects, including algorithms for Axis-Aligned Bounding Box (AABB) and circle colliders, and potentially other shapes like polygons.

* src/java/PhysicsSimulation.java

The main entry point for the Java middleware. It connects the Next.js frontend to the C++ physics engine via JNI, receiving user inputs from the frontend and calling C++ functions for simulation updates.

* src/java/jni/PhysicsEngineJNI.java

A Java class that wraps the native C++ methods, allowing Java to call C++ functions using JNI. It facilitates communication between the frontend and the physics engine.

* src/frontend/ (Next.js)

Contains the code for the Next.js frontend, which provides the graphical user interface (GUI) for interacting with the physics engine.

* components/:
  + Contains React components for rendering the user interface (e.g., buttons to add objects, sliders to adjust gravity, and play/pause controls).
* pages/:
  + Includes the main Next.js pages for the simulation interface (e.g., /simulation page where users can interact with the physics engine).
* api/:
  + Provides the API endpoints for sending user commands to the Java middleware and receiving simulation updates in real time.
* tests/

Contains unit and integration tests for the C++ physics engine, Java middleware, and Next.js frontend components. It ensures that the simulation logic (collision detection, force application) and user interface interactions work as expected.

* .gitignore

A configuration file that specifies files and directories to ignore in version control (e.g., compiled libraries, build artifacts, temporary files from Next.js, and logs).

* README.md

The main documentation file for the project. It contains setup instructions for the Next.js frontend, Java-JNI integration, and C++ physics engine. It also provides an overview of the system, describing how to run and interact with the simulation.

## **9. Testing & Logging**

### **Testing Strategy**

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

Each class in the C++ physics engine (such as Object, Force, and Collider) will have dedicated unit tests to ensure their correctness. Unit tests will be conducted using testing frameworks like **Google Test**. These tests will verify core functionalities, such as force application, motion updates, and collision detection.

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

The Java middleware responsible for interacting with the physics engine via JNI will have unit tests to ensure that the communication between Java and C++ is working correctly. These tests will cover JNI function calls, ensuring that the Java methods properly invoke the corresponding C++ functions.

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

Integration tests will ensure smooth communication across the entire stack—starting from the **Next.js frontend** (user input), through the **Java middleware**, and into the **C++ physics engine**. Scenarios tested will include:

* 1. Adding objects to the simulation.
  2. Applying forces (like gravity or custom user-defined forces).
  3. Collision detection and response.
  4. Simulation control actions (e.g., play, pause, and reset).

These tests will verify that user inputs from the Next.js frontend correctly trigger actions in the C++ physics engine, with appropriate results sent back for real-time visualization.

1. Performance Testing:

The system will undergo performance testing to ensure it can maintain at least 60 frames per second (FPS) for a typical simulation with up to 50 objects. Testing will include:

* Object creation and real-time updates.
* Efficient handling of multiple simultaneous collisions.
* Stress-testing with a larger number of objects to monitor system scalability.

1. **User Interface Testing (Next.js Frontend):**

The Next.js interface will be tested for responsiveness and ease of use. Testing will ensure that users can easily interact with the simulation, such as placing objects, applying forces, and toggling settings. Compatibility across different browsers will also be verified.

### **Logging Mechanisms**

1. **Error Logging:**
   1. **C++:** Any errors or unexpected behavior in the physics engine will be logged to an external file for debugging. This includes issues such as invalid object states, miscalculations, or exceptions during the simulation.
   2. **Java:** The Java middleware will log any issues related to JNI communication, such as errors during the invocation of C++ functions or invalid JNI calls.
   3. **Next.js (Frontend):** The frontend will log errors related to user interactions, API requests to the Java backend, or display issues. These logs will be useful for tracking issues with rendering or user commands.
2. **Simulation Logging:**

Optional logging of simulation details (such as object positions, velocities, and forces) will be enabled to analyze performance. These logs can be generated in both C++ and Java to track the progression of the simulation over time, which is useful for debugging complex interactions between objects.

1. **API Call Logging (Next.js to Java):**

Logging of API requests between the Next.js frontend and Java middleware will help debug issues related to user commands, such as object creation, force applications, or simulation controls (play/pause/reset).

1. **Performance Logging:**

The system will log performance metrics, such as the current FPS, memory usage, and CPU load, to ensure real-time simulation efficiency. These logs will assist in performance tuning and optimization, especially for larger simulations.

## **10. Conclusion**

By incorporating **Next.js**, **Java**, and **C++**, the physics simulation engine will combine high-performance simulations with a modern, user-friendly interface. The **C++** engine will provide the computational power needed for efficient real-time physics calculations, handling complex object movements, forces, and collisions. **Java** will serve as the middleware, ensuring smooth communication between the frontend and the backend via the **Java Native Interface (JNI)**. The **Next.js** frontend will offer a responsive, intuitive interface that allows users to interact with the simulation in real-time, enabling object placement, force application, and control over simulation parameters. This multi-layered architecture will deliver a powerful, scalable, and flexible platform for physics simulation, with seamless integration across all components.