## **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**

The **Physics Simulation Engine** is designed to simulate the movement and interaction of objects under various forces such as gravity, friction, and collisions. 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.o

### **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**: Main interface and application logic. Java will handle input management, system interaction, and visualization. It will call C++ functionalities through Java Native Interface (JNI) for executing the physics simulation.
* **Frontend**:
  + A **Java-based GUI** (e.g., using JavaFX or Swing) for users to interact with the physics engine, allowing them to add objects, define forces, and visualize simulation results.
  + Optional visualization through **SDL or OpenGL** (called from C++ or Java) to render real-time graphical representations of object movements and collisions.
* **Backend**:
  + The **core simulation logic** will be implemented in C++, responsible for:
    - Managing the objects in the simulation (tracking position, velocity, mass, etc.).
    - Handling force calculations like gravity, friction, and custom forces.
    - Detecting collisions and managing the collision response between objects.
  + Java will interact with the C++ engine through **JNI**, enabling the seamless integration of the two languages, where Java serves as the interface and C++ performs the heavy computational tasks.
* **Middleware**:
  + The middleware for this project will be the **Java Native Interface (JNI)**, enabling the communication between Java and C++ for executing physics engine methods from the Java application.

### **Input/Output Requirements**

* **Input**:
  + From Java:
    - Object parameters (position, velocity, mass, shape, etc.), which users can input through the Java GUI.
    - Force parameters (gravity, friction, user-defined forces).
    - Simulation control parameters (time steps, start/stop commands, etc.).
  + From C++:
    - The physics engine will handle simulation inputs like object states and force application through JNI calls from Java.
* **Output**:
  + The C++ engine will return:
    - Updated positions, velocities, and states of objects.
    - Information about collisions, including the results of collision response (e.g., updated velocities after a collision).
  + Java will display these outputs in real time through:
    - A graphical interface for visualizing the simulation (JavaFX, Swing, or SDL/OpenGL).
    - Console outputs for detailed logging of simulation data if needed.

## **4. Functional Requirements**

### **Detailed Features**

* **Object Movement**: The system will update object positions based on velocity and acceleration, influenced by applied forces.
* **Gravity Simulation**: All objects will experience gravitational force, which will accelerate them downward.
* **Friction Simulation**: Objects will experience friction forces based on their velocity and a predefined friction coefficient.
* **Collision Detection**:
  + **AABB Collider**: Axis-aligned bounding box collision detection for rectangular objects.
  + **Circle Collider**: Simple circular collision detection for spherical objects.
* **Collision Response**: After a collision, objects will respond by exchanging velocities based on their mass and impact force.
* **User-defined Forces**: Users can apply custom forces to objects in real-time.

### **Use Cases**

1. **Adding an Object**:
   1. User adds an object with specific parameters (position, velocity, mass).
   2. System updates the object's position in each time step based on the forces applied.
2. **Applying Gravity**:
   1. User enables gravity for the simulation.
   2. System automatically applies gravitational force to all objects.
3. **Collision Between Objects**:
   1. Two objects collide.
   2. The system detects the collision and applies a response, adjusting velocities and positions accordingly.

## **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.
2. **Low Latency**: User interactions (e.g., adding objects, applying forces) should have a response time of under 100 milliseconds.
3. **Efficient Collision Detection**: The system will efficiently handle up to 100 active colliders without noticeable performance issues.
4. **Scalability**: The engine should handle larger object counts by allowing users to adjust simulation parameters to maintain performance.

### 5.2 Usability

**1. Simple Interface**: The Java GUI will be user-friendly, with intuitive controls and preset options for easy interaction.

**2. Error Handling**: Clear, user-friendly error messages will be provided for incorrect inputs.

**3. Interactive Controls**: Users can adjust simulation settings in real-time through the GUI without restarting the simulation.

## **6. Development Setup**

### **Required Tools**

* **Languages**: C++ and Java.
* **C++ Compiler**: GCC or Clang.
* **Java Development Kit (JDK)**: Java SE Development Kit (JDK 11 or higher).
* **IDE**: Visual Studio Code (or IntelliJ for Java), or any preferred IDE that supports both C++ and Java.
* **JNI Setup**: JNI will be required to connect Java with the C++ simulation engine.
* **Dependencies**:
  + SDL or OpenGL libraries for C++ (optional for graphical rendering).
  + JavaFX or Swing for the Java GUI (optional but recommended for visualization).

### **Setup Instructions**

1. **Clone the Repository**:

git clone <repository\_url>

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

Compile the C++ physics engine:

g++ -o physics\_engine main.cpp object.cpp physics\_world.cpp -shared -o libPhysicsEngine.so

1. **Set Up Java Environment**:

Compile the Java frontend and ensure JNI integration:

javac -cp . -d . PhysicsSimulation.java

1. **Run the Simulation**:

Execute the Java program, which calls the C++ engine:

java PhysicsSimulation

## **7. Workflow**

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

* **Object Management**: The C++ engine will handle object creation, state updates (position, velocity, etc.), and force application.
* **Collision Handling**: Collision detection and response will be calculated in C++.
* **JNI Communication**: The C++ functions will be exposed to Java through JNI.

### **Frontend (Java)**

* **User Interface**: The Java frontend will allow users to:
  + Add objects, configure simulation parameters (gravity, friction, etc.), and start the simulation.
  + Visualize the simulation using a graphical interface.
* **JNI Integration**: Java will invoke C++ engine methods through JNI to handle object movements, collisions, and force applications.

### **Middleware (JNI)**

* **JNI (Java Native Interface)** will bridge the communication between the Java frontend and the C++ backend, ensuring that Java can call the physics engine functions.

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

|  |  |
| --- | --- |
| **File/Folder** | **Description** |
| main.cpp | Entry point for the C++ physics engine. |
| object.cpp/.h | C++ implementation of the Object class, handling physical properties of objects. |
| physics\_world.cpp/.h | Core physics simulation logic (forces, collision detection) implemented in C++. |
| PhysicsSimulation.java | Java entry point and GUI handler, managing user input and calling the C++ engine via JNI. |
| libPhysicsEngine.so | The compiled C++ library that Java will link to using JNI. |

## **9. Testing & Logging**

### **Testing Strategy**

* **Unit Testing (C++)**: Each class (Object, Force, Collider) in C++ will have unit tests to verify its functionality, using tools like Google Test.
* **Integration Testing (Java & C++)**: Integration tests will ensure proper communication between Java and C++ through JNI. Scenarios like force application, collision handling, and object creation will be tested.
* **Performance Testing**: The system will be tested to ensure it runs efficiently in real-time, maintaining at least 60 frames per second (FPS).

### **Logging Mechanisms**

* **Error Logging**:
  + Both Java and C++ will log errors or unexpected behavior to a file for debugging.
  + Any JNI-related issues will be logged in both Java and C++ to help troubleshoot integration issues.
* **Simulation Logging**: Optional logging of simulation steps (positions, velocities) for performance analysis.

## **10. Conclusion**

By incorporating both **Java** and **C++**, the physics simulation engine will leverage the computational power of C++ for high-performance simulations while maintaining flexibility and user interaction through Java. The JNI bridge will enable seamless communication between the two languages, providing a solid framework for simulating object movement, forces, and collisions in a flexible and extendable architecture.