# Simulating physics in C++ with Verlet Integration

**Agastya Singh**

Student
Sanskriti School

***Abstract*- This paper presents a physics simulation program that uses the Verlet integration method and the C++ programming. The program can simulate various physical phenomena, such as gravity, collisions, and pendulums. The program also allows the user to interact with the simulation by changing parameters, adding or removing objects, and applying forces. The paper explains the design and implementation of the program, the choice of programming language and libraries, the concepts and formulas of Verlet integration, and the features and limitations of the physics engine. The paper also discusses the significance and applications of physics simulation in science, engineering, education, and entertainment5. The paper concludes with some future avenues for improving and extending the physics simulation program.**

***Keywords*: physics, c++, verlet integration, simulation, physics engine**

## Introduction

### Aim

The aim of this project is to design and implement a physics simulation program using the Rust programming language. The program will be able to simulate various physical phenomena, such as gravity, collisions, springs, pendulums, and fluid dynamics. The program will also allow the user to interact with the simulation by changing parameters, adding or removing objects, and applying forces.

### Significance of Physics Simulation

Physics simulation is an important tool for studying and understanding the natural world. It can help us explore physical systems that are too complex, too large, too small, or too dangerous to observe directly. It can also help us test hypotheses, verify theories, and discover new phenomena. Physics simulation can also have practical applications in engineering, education, entertainment, and art.

### Choice of Programming Language

One possible paragraph of “choice on programming language” in context of making a simple physics simulation engine in C++ using verlet integration is:

C++ is a widely used programming language that offers many advantages for developing a simple physics simulation engine using verlet integration. Some of these advantages are:

- C++ is a compiled language, which means it can run faster and more efficiently than interpreted languages. This is important for physics simulations, which often involve complex calculations and large data sets.
- C++ supports multiple paradigms, such as object-oriented, procedural, and generic programming. This allows the developer to choose the best approach for modeling different aspects of the physics simulation, such as objects, forces, collisions, constraints, etc.
- C++ has a rich set of libraries and frameworks that can help with various tasks related to physics simulation, such as graphics, user interface, math, networking, etc. For example, OpenGL is a popular library for rendering 2D and 3D graphics, while SFML is a framework for creating cross-platform applications with audio, input, and window management features.
- C++ is compatible with other languages, such as C and Python. This means the developer can use existing code or libraries written in these languages, or extend the functionality of the physics simulation engine with scripting or embedding features.

## Tools and Prerequisites

### Toolchains

1. CMake: This cross-platform build system simplifies the process of managing your project's compilation and build. Ensure you have CMake installed on your system. If not, you can download it from https://cmake.org/download/.
2. Clang: We recommend using Clang as your C/C++ compiler due to its robust support for modern C++ standards and optimizations. Verify that Clang is installed and accessible on your system. To obtain Clang, you can refer to https://clang.llvm.org/.

### Libraries

1. SFML (Simple and Fast Multimedia Library): I utilize SFML, specifically the "sfml-graphics" module, for handling graphics and display. You can acquire the SFML library from https://www.sfml-dev.org/download.php.
2. Eigen3: Eigen3 is a versatile C++ template library for linear algebra that will be instrumental in performing vector mathematics efficiently. To access Eigen3, you can obtain it from http://eigen.tuxfamily.org/dox/GettingStarted.html.

## Concepts

### Verlet Integration

Verlet integration is a numerical method that can be used to solve Newton’s equations of motion for physical systems, such as the motion of particles or planets. It is based on the idea of using the position and acceleration at the current time step to estimate the

position at the next time step, without explicitly calculating the velocity. This makes it more stable and accurate than other methods that use velocity, such as the Euler method. Verlet integration also has some desirable properties for physical systems, such as time reversibility and conservation of energy and momentum.

Verlet integration can be derived from a Taylor series expansion of the position function around the current time step, and then adding or subtracting the previous or next time step to eliminate higher-order terms. The basic formula for Verlet integration is:

$$\vec{r}_{t+h} = 2\vec{r}_t - \vec{r}_{t-h} + \vec{a}_t h^2$$

A simplified version of the equation, that I'll be using is:

$$\vec{r}_t = \vec{r}_{t-h} + \vec{v}_t \cdot \Delta t + \vec{a}_t \cdot \Delta t^2$$

In layman's terms,

$$position\_current = position\_old + velocity + acceleration * delta * delta$$

Where *delta* is the chosen time difference.

Building the Engine

Rigid bodies

All the objects in our engine are treated as 'rigid bodies', ie, they don't undergo any kind of compression. For this project, the rigid_body class represents all the objects in the world.

The class definition is as follows:

```
#pragma once
#include <Eigen/Dense>
#include <iostream>

namespace engine {

class rigid_body {
 private:
  Eigen::Vector3d position_current;
  Eigen::Vector3d position_old;
  // Eigen::Vector3d velocity;
  Eigen::Vector3d acceleration;

 public:
  rigid_body(float x, float y, float z);

  Eigen::Vector3d get_position();
  Eigen::Vector3d get_velocity();
  Eigen::Vector3d get_acceleration();

  void update_position(float delta);
  void accelerate(Eigen::Vector3d acc);
};
} // namespace engine
```

The implementations of the methods are as follows:

```
#include <Eigen/Dense>
#include <rigid_body.hpp>

namespace engine {

rigid_body::rigid_body(float x, float y, float z) {
  position_old = {x, y, z};
  position_current = {x, y, z};
}

Eigen::Vector3d rigid_body::get_position() { return position_current; }
// Eigen::Vector3d rigid_body::get_velocity() { return velocity; }
Eigen::Vector3d rigid_body::get_acceleration() { return acceleration; }

void rigid_body::update_position(float delta) {
  const Eigen::Vector3d velocity = position_current - position_old;
  position_old = position_current;
```

```
    position_current = position_old + velocity + acceleration * delta * delta;
    acceleration = {};
}

void rigid_body::accelerate(Eigen::Vector3d acc) { acceleration += acc; }

} // namespace engine
```

The world

All the objects in our world are encapsulated in a class named ‘scene’. This class handles the simulation, applying gravity to objects, and updating the objects positions. The class definition is as follows:

```
#pragma once

#include <iostream>
#include <vector>

#include <rigid_body.hpp>

namespace engine {

class scene {
  private:
    std::vector<rigid_body *> rigid_bodies;
    Eigen::Vector3d gravity = {0, 0, -9.8};

    float delta;

    void update_positions(void);
    void update_gravity(void);

  public:
    explicit scene(float delta) : delta(delta) {}

    void update();

    void add_body(rigid_body *rb);
    const std::vector<rigid_body *> &get_bodies();
};
} // namespace engine
```

The implementation of the methods is as follows:

```
#include <scene.hpp>

namespace engine {

void scene::update() {
    update_positions();
    update_gravity();
}

void scene::add_body(rigid_body *rb) { rigid_bodies.push_back(rb); }

const std::vector<rigid_body *> &scene::get_bodies() { return rigid_bodies; }

void scene::update_positions() {
    for (auto &rb : rigid_bodies) {
        rb->update_position(delta);
    }
}
```

```
void scene::update_gravity() {
  for (auto &rb : rigid_bodies) {
    rb->accelerate(gravity);
  }
}

} // namespace engine
```

**Conclusion**

Key takeaways

- Physics Engine: We have learned what a physics engine is and how it can be used to simulate physical phenomena, such as gravity, collisions, and constraints.
- Verlet Integration: We have explored the Verlet integration method, a simple but effective technique for updating the positions and velocities of objects in a simulation.
- C++ Programming: We have gained valuable programming skills in C++, such as using classes, vectors, pointers, and libraries.

**Future Avenues in Physics Simulation**

- Advanced Simulations: We can extend our physics engine to simulate more complex systems, such as fluids, cloth, or soft bodies.
- Game Development: We can use our physics engine to create games that feature realistic physics and interactivity.
- Scientific Research: We can apply our physics engine to model and study real-world problems, such as climate change, biomechanics, or robotics.

**Acknowledgements**

I would like to express our gratitude to the following individuals and resources that have supported and inspired us throughout this research project:

- Teachers and Mentors: I thank our teachers and mentors for providing us with guidance, feedback, and encouragement.
- Open-Source Community: I acknowledge the open-source community, in particular the maintainers of the Eigen3 project, for offering us invaluable resources, libraries, and tools for physics simulation and programming.

**REFERENCES:**

1. Bjarne Stroustrup. The C++ Programming Language (4th edition). Addison-Wesley Professional, 2013. ISBN: 978-0321563842.
2. Eigen3. Eigen: A C++ template library for linear algebra. Online documentation. Accessed on October 26, 2023.
3. Stack Overflow. C++ Eigen Matrix clarifications. Question and answer. Posted on August 9, 2019. Accessed on October 26, 2023.
4. Verlet integration. (2022, May 28). In Wikipedia. Article. Accessed on October 26, 2023.