Department of Computer Science

Project Name: Luna2D - Physics Engine

Student Names: Michael Kogan, Itay Ifrah

Project Supervisor: Mr. Shtekel Amit

Date of Submission: 07.07.2024

Table of contents

Introduction

This is an academic project as part of a bachelor's degree in computer science studies at Afeka - The Academic College of Engineering in Tel Aviv.
Physics is the science of motion, and understanding how objects move and interact in two-dimensional space is essential for many applications, such as video games, simulations, and animations. However, creating realistic and accurate physics simulations in 2D is not a trivial task. It requires a lot of mathematical and computational skills, as well as a suitable framework for rendering and displaying the results.

The purpose of this project is to develop a 2D Physics Engine called Luna2D that can handle basic physical phenomena, such as gravity, collision, friction, and rigid body dynamics, using C++ and OpenGL.

The objectives of the project are to:

- Design and implement the core components of the physics engine, such as the world, the physics objects, and the physics solver.
- Integrate the physics engine with OpenGL for visualization and user interaction.
- Evaluate the performance and accuracy of the physics engine using the various test cases and scenarios.

This project will demonstrate the feasibility and usefulness of creating a 2D physics engine from scratch using C++ and OpenGL. It will also provide a valuable learning experience for those involved in the development, as well as a potential tool for future projects and research in the field of 2D physics simulation.

Project Goals

Core Components Development

- Design and implement the foundational elements of Luna2D, including the world representation, physics objects, and the physics solver.
- Ensure robustness, efficiency, and accuracy in simulating key physical phenomena such as gravity, collision, friction, and rigid body dynamics.

OpenGL Integration

- Seamlessly integrate Luna2D with the OpenGL framework to enable visualization of physics simulations in a two-dimensional space.
- Implement user interaction features for dynamics and engaging experiences in simulated environments.

Demonstration of Physics Principles

- Showcase the fundamental principles of physics, including gravity, effects, frictional forces, rigid body dynamics, and collision behavior.
- Use clear visualization to highlight these principles for educational purposes.

Educational Goal

- Provide an interactive learning experience for users interested in exploring Engine in particular Physics Engines for educational purposes.
- Allow users to freely run the program (limited to Windows) experiment with variables, and observe the real-time impact on physics simulations.
- Luna2D will be Open source and well-commented so that the learner can understand the inner workings of the engine.

Measurements and Evaluation Criteria

Demonstration of Physics Principles

Objective: To showcase fundamental physics principles through the simulation of physical interactions in a virtual environment by creating and validating all necessary mathematical functions.

Measurement Criteria:

- **Task**: Develop and implement essential mathematical functions such as vector multiplication and collision detection.
- **Question**: Do the objects on the screen react logically and by physical laws?

Implementation of Elastic Collision

Objective: To achieve a realistic simulation of elastic collisions, ensuring that objects bounce off each other accurately.

Measurement Criteria:

- **Task**: Implement the elastic collision model to ensure natural and physically correct interactions.
- **Question**: Does the collision behavior on the screen make physical sense and appear natural?

Core Components Development

Objective: To ensure the engine operates smoothly on the Windows operating system, providing a stable and bug-free user experience.

Measurement Criteria:

- **Task**: Test the engine's performance and stability during a ten-minute runtime.
- **Question**: How many bugs or issues does the user encounter during a ten-minute session?

OpenGL Integration

Objective: To ensure that OpenGL is integrated properly, allowing for smooth and accurate rendering of objects.

Measurement Criteria:

- **Task**: Verify that OpenGL functions as intended, ensuring smooth object rendering and visual consistency.
- **Question**: Does OpenGL run without issues and do the objects appear and behave correctly on screen?

Measurements and Results for Evaluating the System's Performance

Demonstration of Physics Principles

Objective: The primary goal is to showcase fundamental physics principles through the simulation of physical interactions in a virtual environment. This involves creating and validating all necessary mathematical functions that underpin the physical behaviors observed on the screen.

Measurement Criteria:

- **Task**: Create and implement essential mathematical functions such as vector multiplication, collision detection, and other physical calculations.
- **Question**: Do the objects on the screen react in a manner that aligns with physical expectations?

Detailed Analysis:

- **Process**: We developed and tested mathematical functions to ensure they accurately represent physical principles such as momentum conservation and force interactions. This involved rigorous testing of vector operations, force calculations, and object interactions.
- **Result**: The objects on the screen demonstrate logical and realistic behavior. They move and interact in ways that are consistent with the expected physical laws. All mathematical functions have been thoroughly checked and verified for correctness through tests and simulations, ensuring that the physical phenomena are represented accurately.

Implementation of Elastic Collision

Objective: To achieve a realistic simulation of elastic collisions, ensuring that objects bounce off each other in a manner that mimics real-world physics.

Measurement Criteria:

- **Task**: Implement the elastic collision model and ensure that it behaves in a physically accurate and natural manner.
- **Question**: Does the collision behavior on the screen accurately reflect the principles of elastic collisions, where objects bounce off each other without losing kinetic energy?

Detailed Analysis:

- **Process**: The implementation focused on the conservation of momentum and kinetic energy during collisions, ensuring that the objects bounced off one another correctly.
- **Result**: The collision mechanics are mostly accurate, with objects displaying the expected bouncing behavior. However, there is a minor visual issue where objects do not visually contract upon impact as one might expect in a more detailed simulation. Despite this, the physical interactions such as the direction and magnitude of the bounce align with theoretical predictions.

Core Components Development

Objective: To ensure that the core components of the engine operate smoothly on the Windows operating system, delivering a stable and bug-free user experience.

Measurement Criteria:

- **Task**: Test the engine's performance and stability during a ten-minute runtime on Windows.
- **Question**: How many bugs or issues does the user encounter during a standard ten-minute test session?

Detailed Analysis:

- **Process**: We conducted a ten-minute test session, during which we observed and documented any bugs or issues encountered. This included monitoring for crashes, performance slowdowns, or other unexpected behaviors.
- **Result**: The engine performed exceptionally well during the test session. The program ran smoothly without any Windows-related issues, and the user experience was flawless, with no bugs or malfunctions detected throughout the entire ten-minute period.

OpenGL Integration

Objective: To ensure that OpenGL is integrated properly, allowing for smooth and accurate rendering of objects in the simulation environment.

Measurement Criteria:

- **Task**: Verify that OpenGL functions as intended, ensuring that objects render smoothly and interactions on the screen appear as expected.
- **Question**: Does OpenGL integration support smooth object rendering and visual consistency on screen?

Detailed Analysis:

- **Process**: We tested OpenGL through a variety of scenarios to assess rendering quality, performance, and visual consistency. This included examining object movements, visual effects, and rendering stability.
- **Result**: The OpenGL integration is largely successful, with most aspects of the rendering process functioning smoothly and effectively. However, an issue was identified where spamming object creation clicks could cause temporary visual inconsistencies, such as objects appearing inside one another before the system corrects the arrangement. This issue is minor and does not significantly impact the overall functionality.

Summary of Results

- **Demonstration of Physics Principles**: The mathematical functions and physical interactions are accurately represented on screen, with realistic object movements and interactions.
- **Elastic Collision Implementation**: The collision mechanics are mostly correct, though there is a minor visual inconsistency with object deformation upon impact.
- **Core Components Development**: The engine runs smoothly on Windows, with no bugs or performance issues encountered during the test period.
- **OpenGL Integration**: OpenGL integration is effective for most scenarios, though there is a minor visual issue when rapidly creating objectS.

These measurements and results illustrate the effectiveness of the system in meeting its design goals and provide a foundation for future improvements and refinements.

Literature Review

Gravity

This is the fundamental force in physics, it is essential for object interactions. The implementation of gravity in physics engines depends on different models, each of which has different trade-offs. Approaches include the uniform, point, spherical, height-map, and voxel models. The uniform model is simple but lacks realism, while the voxel model is computationally intensive. Millington's book[1] explores uniform gravity and alternatives, along with optimization methods, Chrono[2] written by a group of researchers and developers uses the voxel model for complex objects. Other papers focus on 2D physics engines, they use the uniform gravity model and Euler method, dealing with customization and optimization challenges.
An article on the Sulfur[3] engine shows a height-map gravity model, showing improved realism and stability.
Common challenges include performance, accuracy, stability, and flexibility, with proposed solutions including alternative models and optimization techniques.

Rigid Body

They are objects that maintain their shape under external physical forces, which are essential in physics simulations. The application of rigid body dynamics consists of different approaches:

- ***Lagrangian Approach*** - Derives equations of motion from the principle of least action, using generalized coordinates. A very elegant but complex and nonlinear approach.
- ***Newton-Euler Approach*** - Derives equations of motion from Newton's laws and Euler's rotation equations, using Cartesian coordinates. Simple, linear, but limited and dependent on reference frames.
- ***Featherstone Approach*** - Combines advantages of Lagrangian and Newton-Euler approaches, using spatial coordinates. Efficient and compact, but less intuitive.

Millington's book[1] reviews the Newton-Euler approach and alternatives, along with numerical methods and optimization techniques. The developers of Chrono[2] which is a library are using the Featherstone approach for articulated rigid bodies. Other papers discuss 2D rigid body physics engines, collision 2D engines, and the Sulfur[3] physics engine using various approaches.
Proposed solutions are hybrid approaches, appropriate numerical methods, and optimization strategies.

Collision

The interaction of objects resulting in energy and momentum exchange, is crucial in physics engines. Implementing collision simulation is complex, involving detection, resolution, and handling errors like penetration or jitter.
Various approaches exist:

- ***Discrete Approach*** - Checks collision at discrete time steps, applying impulses or forces to separate objects. Simple and fast but can miss collisions or cause penetration with large time steps or fast objects.
- ***Continuous Approach*** - Checks collisions along object trajectories, calculating the exact time and point of collision. Accurate and stable but can be slow, and complex, especially for curved or deformable objects.
- ***Hybrid Approach*** - Combines discrete and continuous methods, using different approaches for different objects or collisions. Balances performance and accuracy but introduces complexities and inconsistencies.

Millington's book[1] covers the discrete approach and alternatives, discussing numerical methods and optimization techniques for collision simulation.
Chrono's paper[2] is a parallel multi-physics library using the hybrid approach for collisions between rigid bodies, soft bodies, and fluids. The paper details parallelization and optimization strategies, including GPU computing and adaptive mesh refinement. The Hoyos-Cadavid paper[5] describes the creation of a 2D rigid body physics engine for cross-platform using the discrete approach for collisions between simple 2D shapes. It explains the customization and optimization of collision settings for various 2D scenarios.
Mulley's paper[4] about predictive collision focuses on the nature of 2D game engines using the discrete approach for collisions between simple 2D shapes. The paper discusses challenges and limitations, proposing solutions like variable time steps and predictor-corrector schemes.
The paper about Sulfur[3] introduces a physics engine unifying rigid and soft bodies, utilizing the continuous approach for collisions between complex 2D meshes. The paper compares the performance and accuracy of the discrete and hybrid approaches, showing advantages in realism and stability with less computational cost.

● *Friction*

A crucial element in physics engines resists relative motion between objects in contact. Simulating friction poses challenges, particularly in modeling and solving the nonlinear complementarity problem with second-order cone constraints. Different approaches to friction modeling include the widely used Coulomb friction model, the viscous friction model, and the more complex state-based friction model, which captures velocity and history dependence.
Millington's book[1] offers an extensive overview of the Coulomb friction model and its implementation in 3D physics engines, along with alternative models like viscous and state-based friction. The book explores numerical methods and optimization techniques for friction simulation.
The Chrono engine[2] uses the state-based friction model to handle velocity and history dependence, including the stick-slip effect and frictional lag. The paper details parallelization and optimization strategies, incorporating GPU computing and adaptive mesh refinement.
The Sulfur engine[3] uses the viscous friction model for simulating friction between complex 2D meshes. The paper compares the performance and accuracy of the viscous model with the Coulomb and state-based models, demonstrating the former's stability and robustness with reduced computational cost.
Mulley's predictive collision paper[4] centers on this in 2D game engines using the Coulomb friction

model for simulating friction between simple 2D shapes. The paper addresses challenges and limitations, proposing solutions like variable time steps and predictor-corrector schemes.
Hoyos-Cadavid's paper[5] focuses on a 2D rigid body physics engine for cross-platform app development, utilizing the Coulomb friction model for simulating friction between simple 2D shapes. The paper discusses customizing and optimizing friction settings for various 2D scenarios.

- ***Conclusion***

The literature review extensively examines the design and implementation of 2D physics engines, focusing on fundamental components such as gravity, rigid body, collision, and friction.
It identifies key challenges, and compromises in the development of the engines, including considerations of performance, precision, stability, and flexibility.
The review compares existing 2D physics engines such as Chrono[2] and Sulfur[3], and reveals that no single approach dominates and each has specific strengths and weaknesses spending on application requirements.
The review highlights common challenges in simulating basic physics components, paying attention to the need for solutions and improvements such as alternative or hybrid models, appropriate numerical methods, optimization, and parallelization techniques. It highlights limitations in existing research, particularly the lack of systematic comparisons, prediction, and accessible 2D physics engine development tools.

System Architecture

There are several main components in the architecture of our system, we will focus on each of them and explain about it and why we chose it.

- ## OpenGL

  OpenGL is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering. We chose to use it because it is the leading tech in this field + we are also familiar with this technology.

- ## C++

  C++ is an object-oriented programming language that gives a clear structure to programs and allows code to be reused, lowering development costs. C++ is portable and can be used to develop applications that can be adapted to multiple platforms. C++ offers the benefits of a low-level language (i.e. having a lot of control over memory allocation) but at the same time offers the convenience of a high-level language (i.e. being an object-oriented programming language).

- ## Vector Class

  In mathematics and physics, vector is a term that refers informally to some quantities that cannot be expressed by a single number (a scalar), or to elements of some vector spaces. Vectors are the" building blocks" that need to be manipulated mathematically to showcase physics phenomena. In our project, we have implemented manipulations (i.e. vector multiplication, etc.) that are necessary to create the phenomenon that we chose.

- ## Rigid Body Class

  This class will represent a hard body in space. His attributes are speed, position, force, mass, and more. In general rigid body is an idealization of a body that does not deform or change shape. Formally it is defined as a collection of particles with the property that the distance between particles remains unchanged during motions of the body.

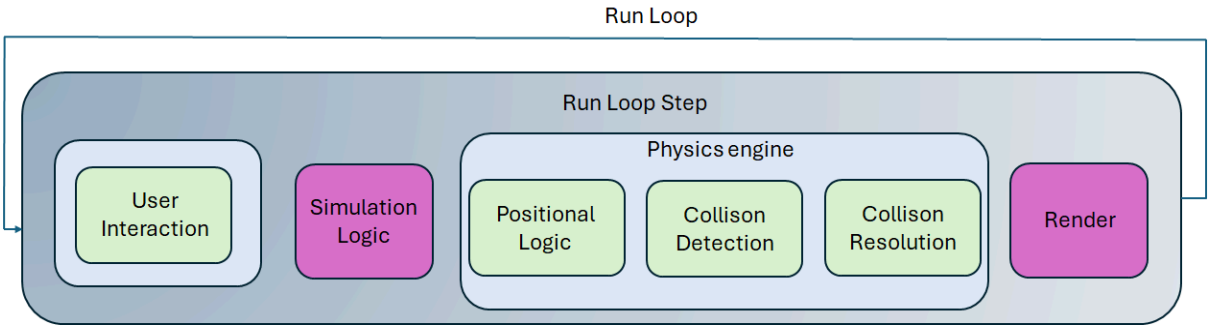
- ## Collision Detection and Response

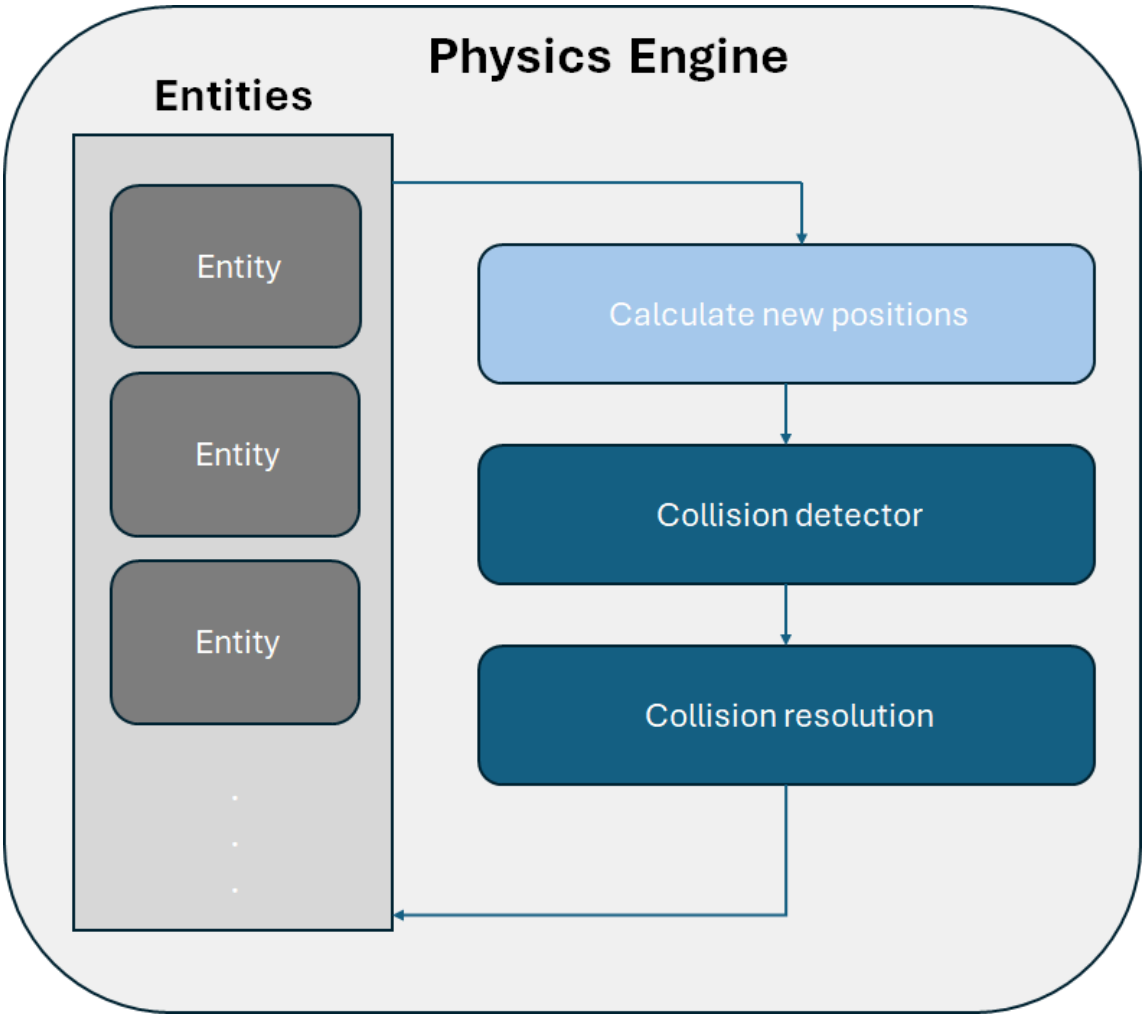
  This component handles the detection of collisions between different shapes (e.g., circles and polygons) and computes the appropriate physical responses. It uses a 2D array of collision detection function pointers to manage interactions between various shapes. The system ensures that interactions between objects are computed accurately, enhancing the realism and physical accuracy of the simulations.

- ***Manifold class***

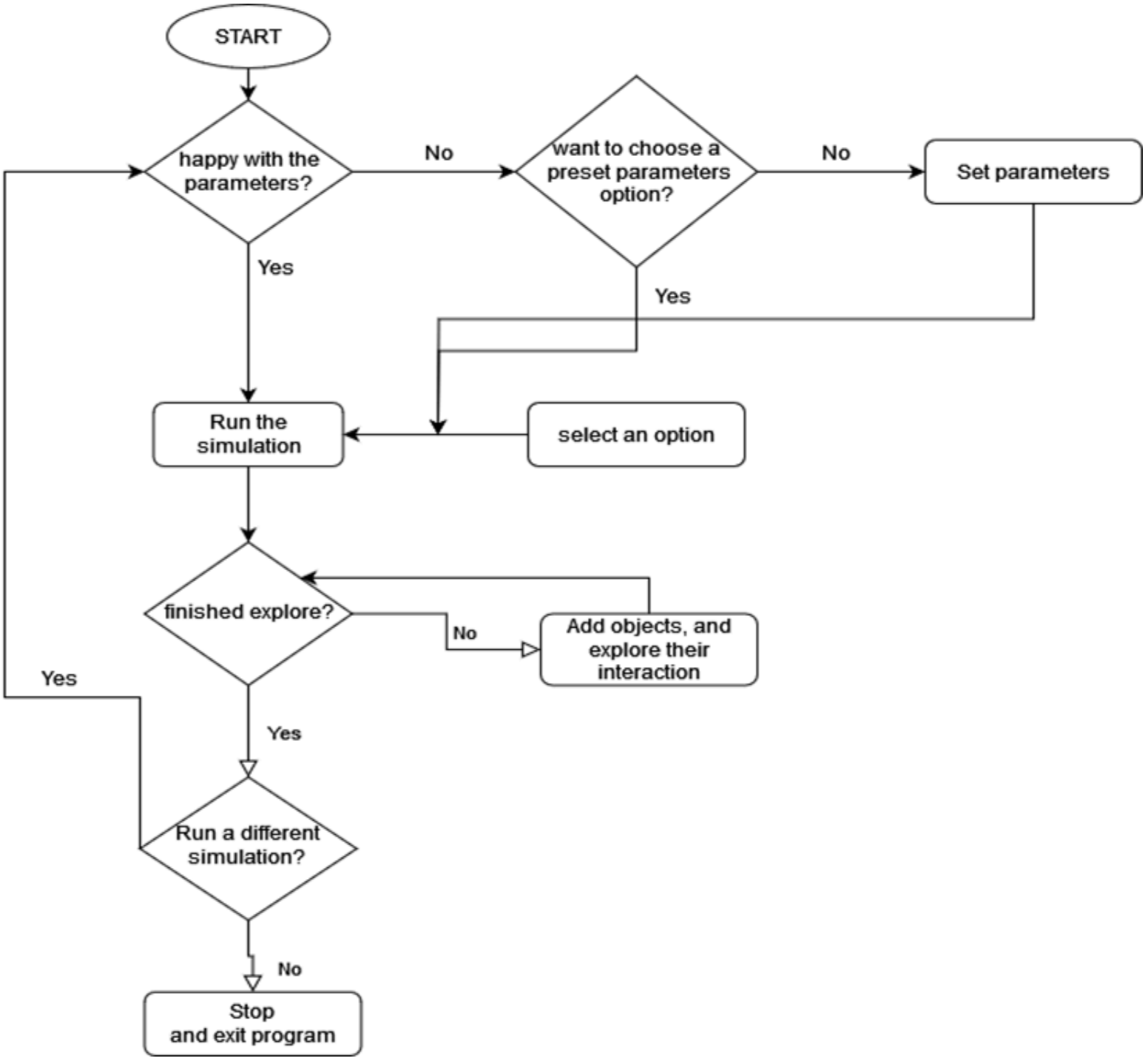
  The Manifold class is responsible for managing and resolving collisions between rigid bodies in the simulation. It calculates the physical response to collisions, including the forces and impulses that result from the interactions between bodies. This class ensures that collisions are handled accurately, preserving the physical realism of the simulation. It initializes collision properties, applies impulses to resolve collisions, and corrects positional errors to maintain stability in the system. The class includes methods for solving collisions, initializing collision properties, applying impulses, and correcting positional errors due to penetrations.

Architecture Diagrams *(true for the development alternative as well)*

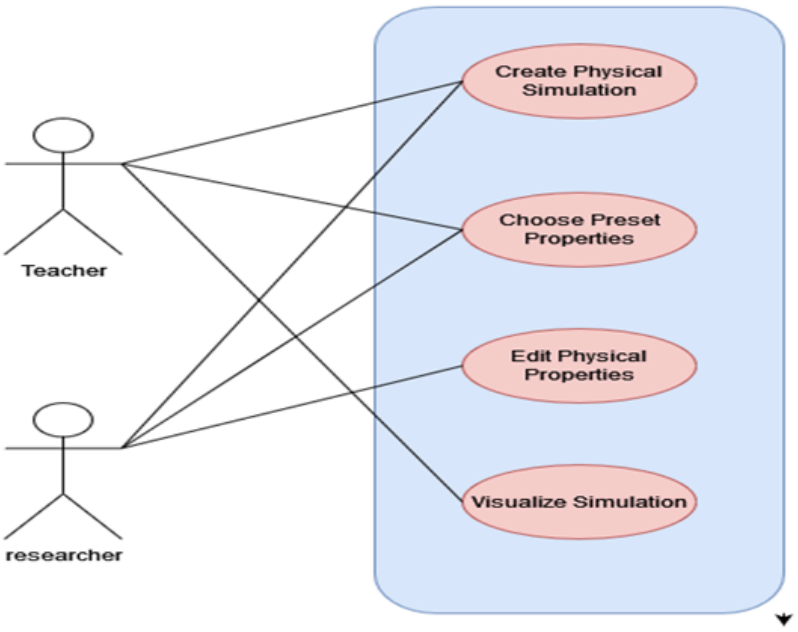




Flow Diagram



Use Cases Diagram



Development Alternative

- ## C#

C# is a versatile, high-level programming language developed by Microsoft as part of its .NET framework. It is known for its simplicity, robustness, and strong typing, making it an excellent choice for developing a wide range of applications, from desktop software to web and mobile apps. C# supports object-oriented programming, which helps in creating modular and maintainable code. The language integrates seamlessly with the .NET ecosystem, providing access to a vast library of pre-built functionalities. This integration simplifies development processes, enhances productivity, and enables the creation of scalable and efficient applications.

- ## Vulkan

Vulkan is a modern graphics and compute API designed to provide high-efficiency, cross-platform access to GPUs. Unlike OpenGL, which is older and sometimes considered less efficient, Vulkan offers lower-level control over hardware, leading to better performance and more optimized resource management. Vulkan's explicit design means developers have more control over the graphics pipeline and can achieve higher levels of performance, especially in complex rendering tasks. It also supports multithreading, allowing for more efficient use of multi-core processors. By using Vulkan in C#, developers can leverage these performance benefits while writing code in a more user-friendly and managed environment.

Recommendations For Further Development:

Our suggestions for further development concern a few areas in this project, on the user side(make it more comfortable for the users, to add more features and improve them.

- ## ***User side***

  On the user side, we suggest making the system more accessible by creating a website that will be able to run the system. That will make the system easier to use for users and open a lot of possibilities(i.e. running more complex experiments because the servers will have more computing power than the user computer etc..).

- Making a better and more interactive UI for the system. The interface is ok now but with further research, we could make the UI more interactive and intuitive for the users, and that will make the system more approachable.

- Comprehensive Tutorials and Documentation: Develop detailed tutorials and documentation to guide users through the system's features and functionalities. This includes step-by-step guides, video tutorials, and FAQs.

- Community Forum and Support: Establish an online community forum where users can ask questions, share tips, and collaborate on projects. Provide dedicated support through live chat or a helpdesk to assist users with technical issues and inquiries.

- Customizable Dashboards: Allow users to create personalized dashboards where they can arrange and view the data and visualizations most relevant to their needs. This customization can enhance user experience by providing tailored insights.

- ## ***System side***

  On the system side, we suggest a few improvements as well, to make it more attractive and more functional for our users.

- Creating an API for the system - we suggest further development to create an API for the system, which will be useful for other developers to exploit the data that the system gives for their uses and advantages. Also, this will make the system more attractive for users because the number of features that other developers will add on top of the system will be useful for more cases and as a result for more users.

- Another suggestion will be to add a way to see the data behind the phenomena that are happening in our physics engine (i.e the system will give a paragraph for each phenomenon with the data about it like a force of impact, numbers of impacts, the object top speed, etc..). We believe that this suggestion will be crucial information for a part of our future users.

- One more suggestion that we have is to run the program on a very powerful server with a lot of computing power. After a few tests and improvements in the code, we believe that the benefit could be significant, as it would allow the system to handle more complex simulations and process data more quickly, providing users with faster and more detailed results.

- Another suggestion is to add an option for the user to control the force of gravity during the simulation. The user will be able to set the gravity force from a drop-down menu that will suggest the gravity force of other planets as options, also the user will be able to set a custom force as he likes to explore. Also, the system will have to show the deformation of soft-bodied objects (such as trampolines) under the impact of stronger gravity.

- Another suggestion is to add an option to add air resistance to the simulation. To make the system as realistic as we can, air resistance has a lot of impact on objects falling and moving in the space of the simulation. To take it a step further we can even add a mode with realistic weather (in the real world, at different heights and places, the wind blows in different directions and with different forces).

- Another suggestion is to switch the use of OpenGL in to a different more powerful library. We chose to use OpenGL in this project because we already know how to use it and that was crucial to the work plan that we planned for. OpenGL has its limitations, we think that if we want to be a real competitor in the market we need to use better and more powerful libraries and use them to their best.

- Another suggestion is to fix a problem during the creation of a new object. For now, you can click inside other objects and create a different object “inside of them, the system corrects itself in a fraction of a second, but we want to disable this possibility to make the system more realistic.

- Another suggestion is to add an option to switch to a 3D simulation environment. The leading simulation competitors are simulating in 3D because it's more realistic. Also having another dimension in your simulation means more data an making a lot of adjustments in the existing code, but we think that if we want to be a real competitor in the market we have to add this option to our physics engine.

- Another suggestion is to add an option to cross-platform. We want the users to be able and run the program on different platforms (an option that will make our program more popular among developers and developers). This suggestion requires a lot of changes in the program but we believe that it is necessary to be a big competitor in this field.

Competitor Analysis Survey

Engine name	Creator	Primary Use	Real-time Physics computation	Collision Detection	Rigid Body Dynamics	Soft Body Dynamics	Performance Optimization	Physics algorithms specialize in	Platform (s)	Language (s)
Box2D	Erin Catto	2D games and simulations	8	9	9	N/A	7	Sequential impulse solver	Cross-platform	C++
Bullet Physics Library	Erwin Coumans	3D games and simulations	9	9	9	8	8	Projected Gauss-Seidel solver	Cross-platform	C++
Chipmunk Physics Engine	Scott Lembcke and Howling Moon Software	2D games and simulations	7	8	8	N/A	8	Sequential impulse solver	Cross-platform	C
Newton Game Dynamics	Julio Jerez	3D games and simulations	8	8	9	7	7	Projected Gauss-Seidel solver	Cross-platform	C++
PhysX	NVIDIA	3D games and simulations	10	10	10	9	9	Projected Gauss-Seidel solver	Cross-platform	C++
Havok Technology Suites	Havok	3D games and simulations	10	10	10	9	10	Projected Gauss-Seidel solver	Cross-platform	C++
Physics Abstraction Layer	Adrian Boeing	3D games and simulations	6	7	7	6	6	Various (Depends on the Backend)	Cross-platform	C++
Open Dynamics Engine	Russell Smith	3D games and simulations	7	7	8	7	7	Quickstep solver	Cross-platform	C++
Project Chrono	Radu Serban and Dan Negrut	3D games and simulations	8	8	9	8	8	Various (Depends on the Module)	Cross-platform	C++
Chaos	Epic Games	3D games and simulations	9	9	9	9	9	Projected Gauss-Seidel solver	Cross-platform	C++

Advanced Simulation Library	**Roman Lygin**	**3D games and simulations**	**8**	**8**	**8**	**8**	**8**	**Lattice Boltzmann method**	**Cross-platform**	**C++**
MonoGame	**MonoGame Team**	**2D and 3D games and simulations**	**7**	**7**	**7**	**6**	**7**	**Various (Depending on the Library)**	**Cross-platform**	**C#**
Farseer Physics Engine	**Ian Qvist**	**2D games and simulations**	**7**	**8**	**8**	**N/A**	**7**	**Sequential impulse solver**	**Windows, Xbox, Windows Phone**	**C#**
Aether.Physics2D	**Matt Bettcher**	**2D games and simulations**	**7**	**8**	**8**	**N/A**	**7**	**Sequential impulse solver**	**Cross-platform**	**C#**
LiquidFun	**Google**	**2D games and simulations**	**8**	**9**	**9**	**N/A**	**8**	**Sequential impulse solver**	**Cross-platform**	**C++**

Review References

[1] Millington, I. (2007). Game Physics Engine Development

[2] Mazhar, H., Heyn, T., Pazouki, A., Melanz, D., Seidl, A., Bartholomew, A., Tasora, A., & Negrut, D. (2013). Chrono: a parallel multi-physics library for rigid-body, flexible-body, and fluid dynamics

[3] Maggiorini, D., Ripamonti, L. A., & Sauro, F. (2012). Unifying Rigid and Soft Bodies Representation: The Sulfur Physics Engine

[4] Mulley, G. (2009). The Construction of a Predictive Collision 2D Game Engine

[5] Hoyos Cadavid, S. (2020). Physics simulation in Mobile Applications: Creation of a two – dimensional rigid body Physics Engine for cross-platform App Development