Jeffrey Song

Building a Physics Engine From Scratch

Computer Science Tripos - Part II

Robinson College

May 10, 2024

Declaration of originality

I, Jeffrey Song of Robinson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University. Signed Jeffrey Song

Date May 10, 2024

Proforma

Candidate Number:	**2419E**
College:	**Robinson College**
Project Title:	**Building a Physics Engine From Scratch**
Examination:	**Computer Science Tripos – Part II, May 2024**
Word Count:	**TBA**[1]
Code Line Count:	**TBA**[2]
Project Originator:	The candidate
Project Supervisor:	Joseph March

Original Aims of the Project

This project takes interest in building a simplified but functional physics engine, primarily focusing on implementing rigid body simulation. With its core being a software engineering challenge, it also heavily involves some computational geometry and physics formulation. The physics engine in the end is expected to be able to simulate the movement of rigid bodies in 3D space, including their position, rotation and collision. These interactions should then be tested with simple experiments against more popular and available physics engines. As extensions, evaluation of its performance and other more in-depth mechanics like soft body simulation can be built on top.

Work Completed

This project has completed all success criteria. A physics engine is built to simulate rigid body dynamics, with collision handling in place. Then, I used a rendering software Blender to test out its APIs, and successfully generated examples of simple working experiments using rigid body simulation. Finally, I collected a few public physics engines and compared mine with them on some simple physical experiments to compare the performance. The result showed that my physics engine was able to generate comparable results.

[1]This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

[2]This line count was computed by TBA

Special Difficulties

None.

Contents

Chapter 1

Introduction

Physics Engine attempts to simulate real life physical properties through well-known physics laws. Starting as a branch of computer graphics, they quickly became commonly used in video games, since many games try to resemble what we already have in the real world. In general, this is done with a physics engine, encapsulating most physics simulation modules. They are then incorporated into the game engine itself, allowing scripts to easily control the movement and settings of important game objects. The most popular game engine for example, Unity[24], has its own implementation of physics engine.

It then becomes significantly easier for game developers to simulate realistic effects. For example, to create "Flappy Bird[8]" in unity, the bird could be attached with a rigid body physics component, which could then modify the movement of the bird according to gravity with response to player controls. The underlying physics engine sees the rigid body, and attempts to modify its position and rotation according to physics law. This engine integration has become an important tool of game development.

1.1 Motivation

Nowadays, many developers choose to rely on whatever comes as built-in for their choice of engine. Unity, for example, has an integration of the NVIDIA PhysX engine, which is provided under a freeware license. Other popular choices include many open-source physics engines, for example SOFA[23]. These ready-made tools are well-polished by big communities and have in-depth support for most functionalities the developers want. However, it's also not rare to see people create their own custom physics engine, as this allows for greater customization possibilities, and some ideas might eventually get contributed to the open-source community. I also take interest in setting up a physics engine from scratch of my own, additionally granting me learning experiences for underlying physics mechanics and software engineering problem solving.

While physics engine consists of many possible components that are considered ongoing research problems, a main building block of it - Rigid body dynamics[17], is more widely used and accepted, with many well-established implementations and experiments. Rigid body dynamics studies systems of interconnected bodies which do not deform under applied forces. Relative position is preserved inside each rigid body, making it easy to

calculate positions relative to the world. Despite the rigid assumptions imposed on the physical objects, rigid body simulation gives rise to quick use of well-known physical laws and common experiments, which would set the core fundamentals for this project.

1.2 Previous related work

There has been continuous efforts on physics simulation particularly in game industries. The development is marked by several big physics engine projects, with some of them still being frequently used to this day. Some milestones in this industry include:

- Havok Physics engine[10], as one of the pioneering physics engine in the early 2000s. Developed by the company Havok, related SDK saw stable release and development still to this day.
- Open Dynamics Engine (ODE)[22] provides a widely used open-source framework for rigid body dynamics. As another early 2000s project, it has a great impact on subsequent related researches.
- Bullet[5], provides a free and open-source software for soft and rigid body dynamics, currently hosted on github[6].
- PhysX[16], with full support from NVIDIA, acquired much popularity with its wide range of simulation features in real-time uses. Some of the more complicated features like fluid simulation were made easy to be use, attracting many developers.
- Unity and Unreal Engine[19] are modern game developing engines which have physics engines components incorporated along with other development utilities. Unity uses a version of PhysX while Unreal uses its in-house physics engine. As complete development tools these game engines are becoming widely used for many purposes.

Other new physics engines are being actively developed and published, for example brax[9], Jade[29], and so on.

Meanwhile, there are also many recent researches concerning some of the physics simulation mechanics both in theoretical level and in application level.

Many theoretical explorations focus on improvements on efficiency and precision. Rigid body simulation[21], soft body simulation[13], and fluid simulation[15], [12], [25] have all been receiving new methods to compute efficiently and accurately.

Application level researches demonstrate several science or real life applications assisted by relevant physics simulation. Examples include [26], [14] and [11].

1.3 Structure of the dissertation

The following Chapter 2 presents theoretical background maths for implementing rigid body simulation. Chapter 3 highlights some of the software engineering challenges along

with some extra relevant algorithms for tackling a few specific problems. In Chapter 4, some evaluations are conducted for the physics engine using simple experiments and comparisons. Finally, Chapter 5 discusses my personal lessons and success, while giving some ideas for potential future work.

Chapter 2

Preparation

2.1 Starting point

Knowledge

Relevant knowledge came from NST Maths in IA, Introduction to Graphics in IA, and Complexity Theory in IB, despite them being only helpful in providing concepts and basic ideas for manipulating computational geometry. I had previously briefly touched on computational geometry related algorithms in coding practicing sites like leetcode prior to the project.

Reference material

I did not read any of the tutorials and guides prior to the project, but I knew they existed in a somewhat plentiful quantity, which lent me some confidence. After starting with the project, I read many of them during the early stages of implementation, including [27], [28] and [1].

Codebase and libraries

I decided to use C++ as the main programming language for the project, which Programming in C and C++ in IB had partially covered, plus me having some experience with it from coding tutorials online. However, I do not have profound experience with building projects using C++, so I do have to learn the repository structure alongside with deploying automated build systems for C++.

As for the libraries, I had no experience with any of the relevant rendering libraries, so I had to learn from scratch on using Blender[2]. Its documentation page with scripting[3] helps a lot. Besides Blender, several third-party libraries provided support for utitilies like testing and plotting. I also tried importing some other libraries during development but later removed them due to them not being helpful.

Here is a list of all third-party software I have used in the current project:

Third-party software	Usage	License
numpy	Efficient arrays for data management	BSD License
matplotlib	Plotting diagrams for data	PSF License
CMake	C++ Build system	BSD License
Makefile	Build automation tool	-
scipy	Conversion of rotations	BSD License
Blender	3D modeling and animation	GPL License

Table 2.1: List of Third-Party Software Used

Development tools

I used Visual Studio Code on my laptop (specs) for engine implementation. Some plots and tests were done with Google Colab and Kaggle Notebook. Platforms including GitHub and Google Drive allowed for backups and simple version control for the project. For the writeup of this dissertation, Overleaf helped with LaTeXcompilation and file management.

2.2 Structure of the engine / Background theory

Physics engine here in my project specifically refers to Rigid body simulation, as it is the main component and the building block of any large-scale complete game engine. Other possible compoents are considered extensions. The main working loop of the engine could be described by a simple feedback loop framework in [figure]

[figure]

After the initial setup of the objects as rigid bodies, the engine begins a main feedback loop. Each iteration of the loop attempts to advance the simulation by a small time frame, and at the end of each time frame, the position and rotation data could be updated, and then serialized and exported to the rendering software, currently Blender.

In each iteration of the loop, all rigid bodies are assumed to start in a nonpenetration state, that is, no pairs of rigid bodies intersect with each other. But touching on some faces are allowed. Then, we perform the unconstrained simulation for a small timeframe dt. During this simulation it is assumed that no collisions happened. Finally, collision detection and resolution are performed, correcting mistakes from the unconstrained simulation. Some more details of these steps are described below.

2.2.1 Unconstrained simulation / Rigid body modelling

(How the linear/angular speed, rotation and translation defines the state of a rigid body.)

In general, the goal of unconstrained simulation is to first track the status of all rigid bodies as reasonable mathematical data structure, and then update them assuming no collisions ever happen.

The state of a rigid body entails the location x, rotation R, linear momentum p, and angular momentum L, relative to the centre of the entire simulation space. The location and rotation in particular encapsulates where to place them in the world, and need to be exported for rendering at the end of each iteration. The entire state can be

expressed as a tuple of these information, say $S = (x, R, p, L)$, and it gives the main moving characteristics of the rigid body. This state can be viewed as a function of time t. The derivative of the state with respect to time, $\frac{d}{dt}S$, can then be derived using variables from S, which would be used when advancing by the small time frame $dt \approx \frac{1}{30}s$ for example in 30 fps rendering.

$$\frac{d}{dt}S = (\frac{d}{dt}x, \frac{d}{dt}R, \frac{d}{dt}p, \frac{d}{dt}L) \tag{2.1}$$

Each element of the state needs to be defined then differentiated separately.

Location x

A 3-element vector containing the x y z coordinates of the center of the rigid body relative to the world center. The center of a rigid body is the center of mass, which is defined as

$$x = \frac{\sum m_i x_i}{M} \tag{2.2}$$

if we consider the body is made up of many small particles, the i-th one at position x_i in world space while having mass m_i. M is the total mass of the body

$$M = \sum m_i \tag{2.3}$$

Here x_i are vectors and masses are scalars, so the center of mass is a vector of the center of mass coordinates. For simplicity think of it as the geometric center of the body, and think of the density to be uniform across the body.

Differentiating x gives the linear velocity v.

$$v = \frac{d}{dt}x \tag{2.4}$$

Rotation R

A 3×3 matrix describing how the rigid body has rotated around its center. For any particle on the body, consider r_i to be its body space location, i.e. the coordinates if the body is placed upright, with its center of mass located at the origin $(0, 0, 0)$. Then, the world space location of the particle x_i can be computed with the help of the rotation matrix, then translated using the location of the center of mass

$$x_i = R \times r_i + x \tag{2.5}$$

R is a rotation matrix, which comes with many properties, for example if

$$R = \begin{pmatrix} R_{xx} & R_{xy} & R_{xz} \\ R_{yx} & R_{yy} & R_{yz} \\ R_{zx} & R_{zy} & R_{zz} \end{pmatrix} \tag{2.6}$$

then multiplying R by an axis-aligned unit vector, say $(1, 0, 0)$, gives the result of rotating the corresponding axis

$$R \times \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} R_{xx} \\ R_{yx} \\ R_{zx} \end{pmatrix} \tag{2.7}$$

Angular velocity ω is used to describe the speed of rotation. Slightly counterintuitively, ω is a vector, where its direction gives the rotating axis and its magnitude gives the speed.

Clearly differentiating the rotation matrix R does not give the angular velocity vector ω. Instead, with some careful analysis [appendix?], we would obtain

$$\frac{d}{dt}R = \omega^* \times R \tag{2.8}$$

where v^* is the star operation that transforms vector $v = (x, y, z)$ to a matrix:

$$v^* = \begin{pmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{pmatrix} \tag{2.9}$$

Note a useful property of this operation

$$a^* b = a \times b \tag{2.10}$$

Linear momentum p

The linear momentum p is defined as sum of products of mass and velocity of all particles comprising the body

$$p = \sum m_i v_i \tag{2.11}$$

making p a vector.

Velocity of a particle v_i is the derivative of its world space location x_i

$$\begin{aligned} v_i &= \frac{d}{dt}x_i \\ &= \frac{d}{dt}(Rr_i + x) && \text{(by equation xxx)} \\ &= \omega^* R r_i + v \\ &= \omega \times (Rr_i) + v && \text{(by equation xxx)} \\ &= \omega \times (Rr_i + x - x) + v \\ &= \omega \times (x_i - x) + v \end{aligned}$$

Now to derive p

$$\begin{aligned} p &= \sum m_i v_i \\ &= \sum m_i(\omega \times (x_i - x) + v) \\ &= \sum m_i v + \omega \times \sum m_i(x_i - x) \\ &= \sum m_i v + \omega \times (\sum m_i x_i - Mx) \\ &= \sum m_i v \qquad \text{(by equation xxx)} \\ &= (\sum m_i) v \\ &= Mv \end{aligned}$$

Differentiating p gives the total force F acting on the body

$$\frac{d}{dt}p = F \tag{2.12}$$

The force can be imagined as the sum of the forces on every individual particle of the body

$$F = \sum F_i \tag{2.13}$$

Angular momentum L

Angular momentum L is a vector defined by

$$L = I\omega \tag{2.14}$$

where I is a 3×3 matrix known as inertia.

$$I = RI_{body}R^T \tag{2.15}$$

where I_{body} is a constant matrix describing the shape of the rigid body, calculated in body space

$$I_{body} = \sum m_i((r_i^T r_i)\mathbf{1} - r_i r_i^T) \tag{2.16}$$

where $\mathbf{1}$ is the identity matrix.

Torque τ_i as a vector describes how the body would rotate if individual force F_i is applied to each particle. Total torque τ is the sum of individual torque.

$$\tau_i = (x_i - x) \times F_i \tag{2.17}$$

$$\tau = \sum \tau_i \tag{2.18}$$

Differentiating L gives the total torque τ, similar to how differentiating p gives the total force F.

$$\frac{d}{dt}L = \tau \tag{2.19}$$

2.2.2 Collision detection and resolution / Nonpenetration constraints

If at the end of the timeframe two rigid bodies end up clipping into each other, then there must have been a collision during the timeframe, meaning the states need to be corrected. This is done by first scanning for collision detection, and if detected, calculate relevant contact information, such as when and where two objects collide. Then the information will be used for collision resolution, where the velocity needs to be corrected before resuming the simulation.

Collision detection

Finding separating plane is a popular method for detecting collisions among convex polyhedrons. Concave objects are trickier to deal with, and luckily most concave objects are good enough when represented by a bounding convex object in practice. In rare cases convex decomposition is required, which is far more complicated and computationally intensive, with the current popular choice being V-HACD [reference], and its complexity should be outside the scope of this project.

Consider two convex polyhedrons A and B. They are considered to be collided if any point in A is also in B. It is actually easier to detect if the pair has not collided, in which case no part of the two objects intersect. Note that having some parts touched between them does not count as collision, so as to avoid repeatedly resolving collision when two touching objects happen to move at the same speed. Using the convex property, there must be a separating plane that separates the space into either only containing A or only containing B.

Furthermore, it is sufficient to only consider two cases. When both cases fail, it can deducted that no separating planes exist.

- Face-vertex collision. Consider the separating plane to be one of the faces of the original objects.
- Edge-edge collision. Consider the separating plane to be between one edge of A and one edge of B, with its normal equal to the cross product of these two edges.

Both cases can be checked by brute force over faces or edges of the two objects.

In the case a collision does happen, find the exact moment the contact takes place, and record the normal of the separating plane at that moment, say n, as well as the points on the two bodies that contact. Next, hand the information over to collision resolution.

Collision resolution

Only the velocities at direction n are altered after the collision takes place, so the first thing is to obtain component of the relative velocity in the n direction using dot product.

$$v_{AB} = n \cdot (v_B - v_A) \tag{2.20}$$

where v_{AB} measures the relative velocity as a scalar, while v_A and V_B are the velocity of the contacting points on object A and B respectively at the moment of collision.

The law of collisions states

$$v'_{AB} = -\epsilon v_{AB} \tag{2.21}$$

where v'_{AB} is the same quantity after the collision happens, and ϵ is the coefficient of resititution.

This coefficient ϵ satisfies $0 \leq \epsilon \leq 1$. At $\epsilon = 0$, two bodies will move together after collision, known as resting contact. At $\epsilon = 1$, two bodies perfectly bounce backwards. In practice, the user could modify this coefficient to achieve the desired effect.

In order to preserve this law, impulse needs to applied to both bodies in different directions. Impulse is the effect of very big force applied in a short duration of time, which encapsulates the interaction of a collision, denoted by a vector. The direction of this vector is of course n, and the magnitude denotes how hard the body gets pushed. The impulse on A J_A and impulse on B J_B satisfy

$$J_A = -J_B \tag{2.22}$$

$$J_A = k \cdot n \tag{2.23}$$

where k is some scalar derivable with contact information

$$k = \frac{-(1+\epsilon)v_{AB}}{\frac{1}{M_a} + \frac{1}{M_b} + n \cdot (I_A^{-1}(r_A \times n)) \times r_A + n \cdot (I_B^{-1}(r_b \times n)) \times r_B} \tag{2.24}$$

I_A is the inertia of A, M_A is the mass of A, r_A is the body space coordinates of the point of contact in A.

The velocity change is just impulse divided by mass, so say for A with mass M_A, the change of velocity Δv_A is just

$$\Delta v_A = \frac{J_A}{M_A} \tag{2.25}$$

Change in angular velocity is

$$\Delta \omega_A = I_A^{-1}(r_A \times J_A) \tag{2.26}$$

2.2.3 Rendering / API

In the grand scheme of the physics engine, rendering is needed for visualization and debugging, which is done by supplying coordinates data at the end of each time frame to the renderer Blender.

The engine should also supply APIs for the user to set up their own simulation environment. Customizations include adding new cuboid rigid bodies with different masses, locations and initial velocity, fixed bodies, time of experimentation and so on.

2.3 Requirement analysis

(Timeline, backup plan, development model) The requirements stayed consistent with the ones listed in the Project Proposal (Appendix []). The core project aims to provide a physics engine capable of modelling rigid bodies, collision detection, and collision resolution (either bouncy or resting). Then the engine will be evaluated with comparisons with existing physics engines under simple experimentations. For extensions adding supports for fluid dynamics, soft body dynamics and real time rendering, as well as evaluating the performance under different parameters.

Main list of deliverables with risk analysis, different modules with dependency

[List: deliverable Implement XXX, Risk Low/Medium/High, Priority Low/Medium/High]

[Modules, arrow meaning depends on, colorcoded]

2.3.1 Development model

The project mainly tackles software engineering challenges. The waterfall development model suits best for the reasons below.

- The separation into different stages gives a good indication on the progress of the project throughout.
- The project has well defined requirements, resources and technologies, allowing the separation to be clear.
- Balancing between the stages are very much needed, as for example spending too much time on researching is not desireable, and the implementations need to undergo testing. The division of stages reminds me to not get stuck at one step and not move on.
- Contrast with agile development, the project focuses on building one predetermined software, so reiterations are unecessary.

During the process of the development there has been some delays in some parts, so I spent some buffer time to catch up with some difficulties, particularly in the implementation stage and dissertation writeup stage. Gantt chart is also used for the planning of the project:

[Gantt chart of timeline]

Chapter 3

Implementation

3.1 Repository Overview

[Figure of repository structure]

[Figure of modules interaction]

[Figures] describes the structure of the repository. The main engine is put in the engine directory, being self-supportive on its own with encapsulated APIs for setting up experiments. The engine part is fully written in C++, and has its own structure, which will be described in [section XXX]. The rendering part is isolated from the main engine because it mainly uses third-party rendering library - Blender, to render the results provided by the APIs of the physics engine. However, the rendered videos were what I heavily relied on when testing and evaluating the engine. They will be described in [section XXX]. The report directory is for storing writeup files for the project.

3.2 Engine

As a reminder, the physics engine mainly involves first modelling objects as rigid bodies, then uses a collision detection-resolution feedback loop for simulation. In the repository, these core mechanics are in the core directory. I used a geometry directory to provide utilities for computational geometry. The core directory includes

3.2.1 Geometry

For better customizability I implemented my own geometry modules. Since the whole engine operates in 3D in every way, I could use just two classes, `Vector` and `Matrix`, in order to represent 3×1 and 3×3 tensors respectively. Some common operators are also supported for quick C++ style syntax. Some of the interfaces include

- `static Matrix star(const Vector& v)`: the star operation that creates a matrix $A = a^*$ for any vector a such that $A \cdot b = a \times b$.
- `Matrix inverse()`: finding the inverse of a matrix
- `Vector operator*(Vector rhs)`: multiplying matrices with vectors

3.2.2 Rigid body

The main class for individual rigid bodies is `RigidBody`, with each instance of it representing one unique rigid body. The states and auxilaries are also individually classed as `RigidBodyState` and `RigidBodyAuxilaries` respectively.

For each rigid body, the main fixed properties are its `mass` and `bodySpaceInertiaTensor`. I also implemented a prepared `CuboidRigidBody` that subclasses from `RigidBody`, and instead uses `length`, `width`, and `height` to automatically calculate the integral. The subclasses are also responsible for documenting shapes information when outputting.

In order to implement the later edge-edge or vertex-face collision detection, the collection of vertices, edges and faces are also saved and managed by the subclasses.

All the derivatives and computable auxilaries are encapsulated as methods like

```
RigidBodyState computeDerivative(RigidBodyState state,
RigidBodyForceAndTorque forceAndTorque)
```

3.2.3 Ordinary Differential Equations

Earlier in [section XXX], the state of a rigid body, S, and its derivative with regards to time t, $\frac{d}{dt}S$, are defined with variables from states S together with outside force and torque. Force and torque are assumed to be known ahead of time (they are usually a constant, for example, gravity constantly pulls everything downwards, or other user-defined constant force fields), but they also might change depending on the result of collision resolution. They can be treated as a known function of time, say $f(t)$. The derivative depends on the force and torque and the current state, so it can be treated as a known function of both of them, say $g(S, f(t))$. This helps establishing a differential equation for predicting the next states

$$\frac{d}{dt}S = g(S, f(t)) \tag{3.1}$$

which can now be processed as an ordinary differential equation (ODE), where the states at the current time frame t_0 are given, and the equation is used to find the states at the next time frame $t_0 + dt$.

I decided to use a naive approximation (known as Euler's method) by assuming the derivative remains constant across the duration of the small time frame dt. That is, I derived the derivative at t_0 as $\frac{d}{dt}S = g(S, f(t_0))$, and advance the state S by simply adding $dt \cdot \frac{d}{dt}S$ to the state S. This in fact works very well already, considering most of my experiments expand over several seconds, so it turns out to be very difficult if not impossible for humans to notice the tiny discrepancies. It is also possible to divide to time steps even more, and with smaller steps (smaller dt) the accuracy will be improved even further.

There are in fact more precise simulations using an actual ODE solver. For example, there are known ODE solvers such as the Boost.ODEInt library that deploys more complicated algorithms such as fourth-order Runge-Kutta method to integrate the ODE.

However, these are not considered in this project considering the original goal of the project is the ability to simulate rather than the ability to accurately approximate things. Of course, they are interesting candidates to consider as future improvement.

3.2.4 Unconstrained Simulation

Given the rigid bodies and simulation basics, the actual state advancement is as simple as going through all bodies recorded in the engine:

```
void advanceByTimeFrameUnderConstantForce(double dt) {
    for (auto body : bodies) {
        auto states = body.getStates();
        auto [time, state] = states.back();
        auto stateDerivative = body.computeDerivative(state, body.getForceAndTorque()
        states.emplace_back(std::make_pair(time + dt, state + stateDerivative * dt));
    }
}
```

All the state history along with timing information is recorded and saved in a `std::vector` in time order. This is helpful when we need to unroll the states to a previous time frame due to collision detection.

3.2.5 Collision Detection

This stage turns out to be one of the trickier modules to implement. I implemented them directly as methods of the main `PhysicsEngine` class. The collision detection has several steps.

Entry point

Collision detection methods are called immediately after advancing the previous unconstrained simulation. The problem of the previous unconstrained simulation is that it does not consider collision or interactions between rigid bodies in the engine. In fact, what we did is to advance the states no matter what happened, and check if any problems arise afterwards. The idea behind this is that collisions are rare in most scenerios, and it is common that only a couple of collisions happens throughout the experiment. Hence, we could be lazy and check for collision every 10 steps of regular unconstrained simulation, since checking for collisions are quite resource-intensive. However there is the issue of having objects clip through each other with longer time frames of checking. In practice I called the detection after every regular unconstrained simulation since it is enough for simple experiments.

Bounding boxes

As a preprocessing step, the bounding boxes of all existing rigid bodies are generated. Bounding box is the smallest box that encapsulates the object but is also aligned with the axis, which simplifies computations. [fig] shows an example of a bounding box.

[fig]

If collisions were to occur, the bounding boxes, which represent an even larger space, must collide first. So checking if two bounding boxes overlay is a good precondition for actual collisions, and does a good job of filtering out most of the non-colliding cases.

Checking if any pairs of bounding boxes overlap is much more efficient. Even a naive pairwise check between all pairs of bounding boxes are more efficient because no need to iterate over edges and faces, with a complexity of $O(n^2)$ where n is the number of rigid bodies. But it can be solved more efficiently.

- One-dimensional case: Consider what will happen if the bounding boxes are in only one dimension. In that case, the boxes are actually simply segments. Let's say the i-th segment covers l_i to r_i, and we are tasked with finding all the overlapping pairs quickly, where there are n segments at the beginning, and at most m pairs overlap.

  [fig]

  This problem can be solved in $O(n \log n + m)$ time with a sweep line algorithm. First, all the endpoints (l_i and r_i, a total of $2n$ endpoints) will be sorted from left to right. Then a sweep is performed from left to right, while keeping a list of which segments are active. When we get a left endpoint l_i, the segment is marked as active, and it will be paired with all other active segments. When we get a right endpoint r_i, the corresponding segment is removed from the active list. The list needs to support quick insertions and deletions, so a doubly linked list for example would suffice. The initial sorting costs $O(n \log n)$, the sweep costs $O(n)$, going through the pairs costs $O(m)$, adding to a cost of $O(n \log n + m)$, and is the most efficient algorithm for solving this problem.

  After solving this initial problem at the first time frame, the subsequent problems for later time frames can be slightly improved based on the assumption that they do not greatly deviate from the original problem. The bottleneck in the algorithm is the sorting of the endpoints. We could use the idea of insertion sort here - for each endpoint, check if it is to the right of the previous endpoint, and if not swap its position with the previous endpoint. Now the complexity becomes $O(n + s)$ where s is the number of swaps. Although in the worst case scenerio s could be as high as $O(n^2)$, we could expect s to be low on average. After re-sorting all the endpoints, the sweep step could continue as before, yielding $O(n + m + s)$ in total.

  But we could even do better, by not performing the sweep step at all! Let's assume the whole list of overlapping segments is recorded. Consider when a pair of segments, i and j, change from overlapping to not overlapping. Then, the relative order of the four endpoints l_i, r_i, l_j, r_j must have changed. So when doing the pesudo insertion sort, two of those four values must have swapping places with another when adjacent. Same goes when i and j change from not overlapping to overlapping. Therefore, whenever we perform a swap, we should re-check the overlapping status of the pair of underlying segments have changed. Now we successfully eleminated the need for sweeping phase, yielding a total complexity of $O(n + s)$.

- Three-dimensional case: Now going back to our 3D bounding boxes. There are efficient algorithm with advanced data structures that solve the one-time problem in $O(n \log n + m)$ as well. A general result is for d dimensions the overlapping pairs can be found in $O(n \log^{\max(1,d-2)} n + m)$.

  For subsequent steps, a similar optimization could be used. This time, the boxes have three sets of segments: $xl_i, xr_i, yl_i, yr_i, zl_i, zr_i$. Thankfully checking whether two bounding boxes overlap is still $O(1)$. So similarly, consider when a pair of boxes, i and j, change from overlapping to not overlapping or vice versa. The relative order of four endpoints on at least one axis must have changed. Therefore, if we simply maintain three sorted lists of each dimension, and perform a pesudo insertion sort for each of them with re-checking if pairs of bounding boxes still overlap, the whole list of overlapping bounding boxes is still maintained at a manageable $O(n + s)$ complexity.

Collision

Collisions are only further checked on overlapping bounding boxes. Each collision is recorded as a class:

```
class Collision {
    RigidBody *a, *b;
    Vector normal;
};
class FaceVertexCollision : public Collision {
    Face face;
    Vector vertex;
};
class EdgeEdgeCollision : public Collision {
    Edge ea, eb;
};
```

The two types of collisions are both accounted for. Degenerated cases like vertex-vertex collision will be described with the more general collision case. For face-vertex collison, I force `RigidBody *a` to be the body with the face and `RigidBody *b` to be the body with the vertex. The `Vector normal` describes the normal vector of the separating plane at the point of collision, which is either the face for face-vertex collision or the cross product of the two edges in the edge-edge collision case. All the coordinates here should use the world space coordinates. The process of collision checking is as simple as going through all possible pairs of faces-vertices and edges-edges, and checking if they have penetrated.

The test for penetration is done by simply testing the relative velocity defined previously, v_{AB}, against some constant threshold.

There are also optimization possibilties here. For example, it is possible to cache the history of collisions between a specific pair of rigid bodies. Therefore, when checking for a new collision, we could first check if previous collision setups work, say if the same pairs

of edges could separate them apart. If the check fails, we can first try to replace them with adjacent faces or edges, hoping that it only barely rotated. Currently these ideas are left as possible future improvements.

Finding the time of collision

Binary search could be used to find the exact time the collision takes place, assuming one already happens. If there are multiple pairs of collisions with the time frame t_0 to $t_0 + dt$, we are interested in finding the one that occurs first.

The idea of the binary search is to first fix the possible range, in this case the range t_0 to $t_0 + dt$. Previously we have discussed how to efficiently check if a collision happened at any time frame t. Therefore, the midpoint of the range can be queried to narrow down the possible range, up until a desired precision:

```
left = t0;
right = t0 + dt;
eps = 1e-5;
while (right - left > eps) {
    middle = (left + right) / 2.0;
    engine.advanceByTimeFrameUnderConstantForce(middle - left);
    if (engine.checkCollision())
        right = middle;
    else
        left = middle;
    engine.rollBack();
}
```

This gives a complexity of $O((n+s) \cdot \log \frac{dt}{eps})$.

3.2.6 Collision Resolution

Most of the computations are already given in the previous section [section XXX]. Here, we just plug in the computations for the relevant quantities. In the code, it can be encapsulated as a simple method `resolveCollision(Collision *c, double coefficientOfResititution)`, where the collision is resolved by supplying the collision and the coefficient of resititution. In the actual implementation I chose a default value for the coefficient as $\epsilon = 0.5$.

Now, after correcting the states of all rigid bodies, the states will be saved as a special time frame, and the simulation will resume in a physically correct manner.

3.3 Render

Strictly speaking the rendering step is out of the scope of the actual project here, because the users could just invoke the APIs of the core engine and get whatever data they would like to collect. However, it has ultimately become an important step of the project with

the need to perform experiments and visualize simple simulations. This is also where a lot of trouble shows up for my project which ultimately stems from my poor knowledge of rendering libraries prior to this project.

3.3.1 Motivation

I chose Blender because it is known as a very popular 3D rendering program. It turns out that it is popular for being friendly to people with little programming knowledge, with a good UI, but not particularly great for dumping external data in for it to render whatever I want. It seems that the only relevant functionality for this purpose is to use the Blender python scripting API, which, while has some documentations and some powers, aren't as well-studies as other functionalities of Blender, since it is still under active development and refinement. In fact, the documentation just updated to 4.1 [ref] within the duration of the project from 4.0.

3.3.2 Native python scripting

Blender scripting works with a custom module provided by Blender as the rendering module. However, it does not work like other common, distributed modules like matplotlib - the module is provided in a native python version inside Blender, hidden in the source directory.

In order to use it, most of the rendering related scripts have to be built on top of this native environment, which cannot be easily manipulated as common python distributions. For example, installing new libraries like scipy need to be done directly with the pip installer on the native python environment.

I ended up using a setup script to deal with all the nuances by directly specifying the `%BLENDERPATH%`, include the python in the `%PATH%` together with libraries installations, and then return to the current directory for running the script.

3.3.3 Rendering process

It took me some time to learn how to setup the rendering stage. First, the camera needs to be setup with an appropriate bounding box. Then, remove all existing objects in the stage, because for some reason some of the objects could persist to the next run. With each rigid body, create a separate mesh for them, add animation data for them, and interpolate the animation with the key data at each time frame from our simulation data. Finally, setup the rendering options and render the animation to a video.

There are also many details regarding each animation data. For example, the rotation used by Blender is the quanternion one. Quanternion functions as an alternative to the matrix form of rotation. Luckily `scipy.spatial.transform` provided translations between matrix form and quanternion form.

3.3.4 Pipeline

Finally, I created a way to pipe the data the engine outputs to be directly used by the blender python script. I used C++ code to setup the experiment, build it with cmake and makefile, which should automatically dump the data into the rendering directory. Then I head into the rendering directory and use my setup script to render the result as a video. It turns out the rendering could take a very long time, for reference a 10 second video ends up renderng for 10 minutes. To adapt to this I tried to work with small and simple experiment setups to perform some quick testings.

Chapter 4

Evaluation

Evaluation of physics engines in general can be quite challenging, especially for quantitative analysis. Little work has been done in this area as of now, likely due to the complexity, systematic bias, and the lack of needs.

The evaluation of this project is split into three parts: Benchmark selection, Quantitative evaluations, and Success criteria.

4.1 Benchmark selection

Quantitative evaluations will be largely comparison-based. I will be choosing two open-source physics engines that support similar features to compare against. The following engines have been found as possible candidates:

Physics engine	Website
Advanced Simulation Library	asl.org.il
Bullet	pybullet.org
Newton Game Dynamics	newtondynamics.com/forum/newton.php
Open Dynamics Engine	www.ode.org
PAL	www.adrianboeing.com/pal
PhysX	www.nvidia.com/en-gb/geforce/technologies/physx
Project Chrono	projectchrono.org
Siconos	nonsmooth.gricad-pages.univ-grenoble-alpes.fr/siconos
SOFA	www.sofa-framework.org
Tokamak physics engine	github.com/isegal/TokamakPhysics

They have then been further narrowed down in consideration of supportability, documentation and popularity. I end up choosing the following physics engines as benchmarks:

Physics engine	Reason???TBA, +screenshots, +brief description on setups
PhysX	www.nvidia.com/en-gb/geforce/technologies/physx
SOFA	www.sofa-framework.org

4.2 Quantitative evaluations

The functionalities of selected physics engines along with mine will be evaluated through a series of small runtime tests. The tests partially drew inspiration from other existing researches on physics engines[20].

4.2.1 Bounce test

Setup

To test whether the engines could handle object collisions, I measure if the momentum and energy are preserved.

Two identical cubes will be placed in a world with no gravity.

Cube A, the one on the left, is located at $(0, 0, 0)$, and is moving along the `+x` direction with a velocity of $1\,\mathrm{m\,s^{-1}}$.

Cube B, the one on the right, is located at $(5\,\mathrm{m}, 0, 0)$, and is moving along the `-x` direction with a velocity of $1\,\mathrm{m\,s^{-1}}$.

Both cubes are 1x1x1 and have a mass of $1\,\mathrm{kg}$. The coefficient restitution ϵ is a variable that ranges between 0 and 1.

Theoretically, the sum of momentum vectors should stay at $(0, 0, 0)$, and the total energy at $1\,\mathrm{J}$. The actual sum of momentum and total energy, as simulated by the engine, will be plotted against time. The less these values vary, the more accurate the simulation is.

Result

All perfectly correct, for all engines. [fig]

As seen in [fig], both the sum of momentum and the total energy stay true to the theoretical results with no noticeable inaccuracies. This is a very basic bouncing task, so as expected all physics engines could reliably accurately simulate it.

4.2.2 Support test

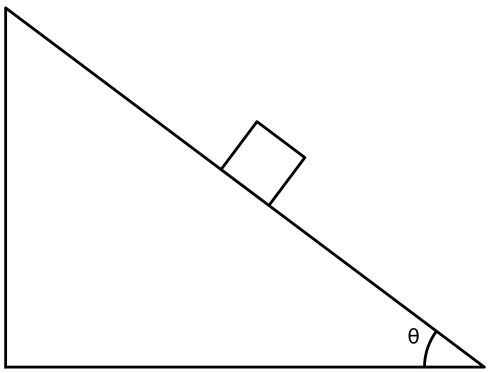
Setup

A box will be placed on a fixed inclined plane. The coefficient restitution ϵ is set to be 0 - this means a perfectly inelastic collision happens between the box and the plane, so the plane should be able to support the ball. Test if the ball could slide down the plane in an

accurate manner. In order to simulate the plane being "fixed" while still being a cuboid rigid body, its mass will be set to infinite, and will be carefully placed with some initial rotation.

The plane passes through the origin $(0, 0, 0)$ and is sufficiently long to hold the cube. The experiment is mostly within the $x - z$ plane. The cube is 1x1x1, and its center of mass is placed at $(-20\cos\theta + \frac{1}{2}\sin\theta, 0, 20\sin\theta + \frac{1}{2}\cos\theta)$. This means its middle contact point with the plane is always $d = 20$m from the origin. Constant gravity $G = 10\,\mathrm{N}$ will be applied to the cube. At any none 0 degree angle ($\theta > 0$), the cube should slide down the plane. The time it takes for the x coordinate of its centre to reach $\frac{1}{2}\sin\theta$ will be measured and checked with the theoretical result. Then the experiment will be repeated for different angles θ within the range $(0, \frac{\pi}{2})$.

In order to obtain a theoretically perfect result, consider a cube moving frictionless slope. Its acceleration along the slope should be $a = \frac{G}{m}\sin\theta$. If the time it takes when accelerating from a static position is t, the distance travelled should be $\frac{1}{2}at^2$. So the theoretical time it takes to reach the origin should be $t_0 = \sqrt{\frac{2d}{a}} = \sqrt{\frac{2dm}{G\sin\theta}} = \frac{2}{\sqrt{\sin\theta}}$.



Result

5, 7.2, 7.433 10, 5.04, 5.167 20, 3.5, 3.633 30, 2.88, 2.900 40, 2.54, 2.567 50, 2.32, 2.333 60, 2.18, 2.233 70, 2.08, 2.133 80, 2.04, 2.100 85, 2.02, 2.067

As seen in [fig], in general the time it takes for the cube to slide down matches with the theoretical result, but in general all the engines give a slightly higher result. This is likely due to accumulated inaccuracies of repeated collisions along the slope. From the view of a physics engine, the cube repeated collides with the slope, and the result is its downwards momentum transfers into sliding momentum. The transfers needed to be calculated every time, so it could be slightly delayed compared to the theoretical analysis.

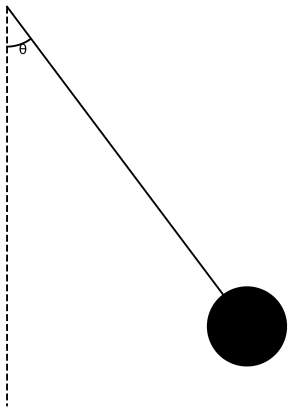
4.2.3 Pendulum test

Setup

Have a pendulum setup like in the following figure.

A small object is attached to a fixed point using a "thread" with a length of $L = 10\,\text{m}$. To simulate such a "thread" in my physics engine, the object is simply "pulled" back up after unconstrained simulation at each time step.

The initial angle between the bar that connects the sphere and an imaginary vertical line through the fixed point is θ.



θ is gradually increased from 5° to 85°. For each θ, the period of the pendulum is measured by recording the time it takes for the sphere to reach its lowest point 5 times.

Theoretically, the period of a pendulum is

$$T = 2\pi \cdot \sqrt{\frac{L}{g}} \tag{4.1}$$

The periods as simulated by the engines will be plotted against the angle θ.

Result

5, 18.24 (/ 2.5) 10, 18.30 20, 18.26 30, 18.26 40, 18.30 50, 18.36 60, 18.46 70, 18.58 80, 18.80 85, 18.93

As seen in [fig], this is an experiment where all the physics engine deviate quite a bit from the theoretical results. Engines generally take a longer time to swing the pendulum as the degree increase, possibly due to the accumulation of correction errors from the thread constraining the object, which should have resulted in an arc, but got simulated as small segments.

4.3 Success criteria

Success criteria from the project proposal:

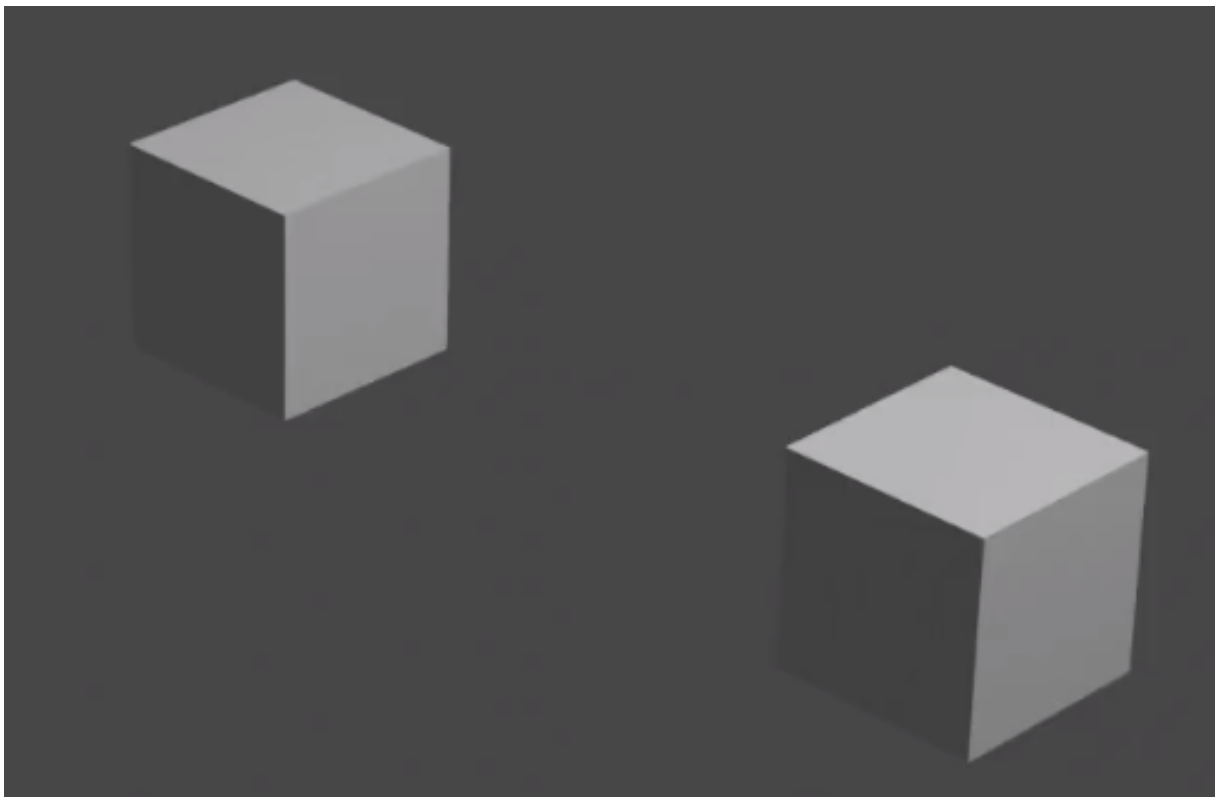
- Implement all basic modules: Object modelling, Collision detection, Bounce, Friction, Stability.
- Evaluate the engine by comparing it with popular existing engines with simple experiments.

- Demonstrate that the engine works with screenshots of simple examples.

For extensions (ordered by priority):

- Implement fluid dynamics.
- Implement different versions of the engine for whether GPU is used, and for other interesting parameters like the number of cores used. Then evaluate the performance.
- Implement real-time rendering, which should allow the project to meet all previous criteria without third-party rendering libraries.
- Implement soft-body dynamics.

Referring back to the success criteria, I consider all core criteria to be completed and successful. Extensions were attempted but not as successful. All basic modules related to rigid body simulation are implemented and evaluated with simple experiments. The followings are some example screenshots of the resulting rendered video.



[fig]

The results of the evaluations also show a high correlation with real-life physics results. Therefore, my physics engine is indeed capable of simulating simple rigid body interactions.

4.3.1 Limitations

As seen from some of the experiments [ref], the inaccuracies of simulating certain interactions are slightly greater than popular existing physics engine. Furthermore, external functionalities, configurable options, API documentations and bug considerations are definitely not as complete as other popular physics engines. Most modern physics engines provide several alternate simulation methods, for example for solving differential equations mentioned in [ref]. Some extensions like liquid simulation are also not in a working state in my engine but is supported by many other engines. On the other hand, they possess dedicated maintenance by large groups of developers, so it is not reasonable to aim at surpassing their massive project in every possible way.

Chapter 5

Conclusion

TBA

Bibliography

[1] 3d physics engine tutorial. https://www.youtube.com/playlist?list=PLEETnX-uPtBXm1KEr_2zQ6K_0hoGH6JJ0. Accessed: 2023-10-12.

[2] Blender. https://www.blender.org/. Accessed: 2023-10-12.

[3] Blender scripting documentation. https://docs.blender.org/api/current/index.html. Accessed: 2024-10-5.

[4] Box2d. https://box2d.org/. Accessed: 2023-10-12.

[5] Bullet. https://github.com/bulletphysics/bullet3. Accessed: 2023-10-12.

[6] Bullet on github. https://github.com/bulletphysics/bullet3. Accessed: 2024-10-5.

[7] Cuda. https://developer.nvidia.com/cuda-toolkit. Accessed: 2023-10-12.

[8] Flappy bird. https://en.wikipedia.org/wiki/Flappy_Bird. Accessed: 2024-10-5.

[9] C Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax–a differentiable physics engine for large scale rigid body simulation. *arXiv preprint arXiv:2106.13281*, 2021.

[10] Havok physics engine. https://www.havok.com/havok-physics/. Accessed: 2024-10-5.

[11] Hantao He. *Simulation of realistic soil behavior in geotechnical tests using physics engine technique*. PhD thesis, Iowa State University, 2021.

[12] Milan Klöwer, Sam Hatfield, Matteo Croci, Peter D Düben, and Tim N Palmer. Fluid simulations accelerated with 16 bits: Approaching 4x speedup on a64fx by squeezing shallowwaters. jl into float16. *Journal of Advances in Modeling Earth Systems*, 14(2):e2021MS002684, 2022.

[13] Sizhe Li, Zhiao Huang, Tao Du, Hao Su, Joshua B Tenenbaum, and Chuang Gan. Contact points discovery for soft-body manipulations with differentiable physics. *arXiv preprint arXiv:2205.02835*, 2022.

[14] Harinarayanan Nampoothiri MG, Chinn Mohanan, and Rahul Antony. Slip ratio prediction in autonomous wheeled robot using ros-physics engine based hybrid classification approaches. *Journal of Intelligent & Robotic Systems*, 109(1):9, 2023.

[15] Octavi Obiols-Sales, Abhinav Vishnu, Nicholas Malaya, and Aparna Chandramowliswharan. Cfdnet: A deep learning-based accelerator for fluid simulations. In *Proceedings of the 34th ACM international conference on supercomputing*, pages 1–12, 2020.

[16] Physx. `https://www.nvidia.com/en-gb/drivers/physx/physx-9-19-0218-driver/`. Accessed: 2023-10-12.

[17] Rigid body dynamics. `https://en.wikipedia.org/wiki/Rigid_body_dynamics`. Accessed: 2024-10-5.

[18] Raúl Rojas and Ulf Hashagen. *The first computers: History and architectures*. MIT press, 2002.

[19] Andrew Sanders. *An introduction to Unreal engine 4*. AK Peters/CRC Press, 2016.

[20] Axel Seugling and Martin Rölin. Evaluation of physics engines and implementation of a physics module in a 3d-authoring tool. *Umea University*, 2, 2006.

[21] Shubham Singh, Ryan P Russell, and Patrick M Wensing. Efficient analytical derivatives of rigid-body dynamics using spatial vector algebra. *IEEE Robotics and Automation Letters*, 7(2):1776–1783, 2022.

[22] Russell Smith et al. Open dynamics engine. 2005.

[23] sofa. `https://www.sofa-framework.org/`. Accessed: 2024-10-5.

[24] Unity. `https://unity.com/`. Accessed: 2024-10-5.

[25] Ricardo Vinuesa and Steven L Brunton. Enhancing computational fluid dynamics with machine learning. *Nature Computational Science*, 2(6):358–366, 2022.

[26] Kun Wang, Mridul Aanjaneya, and Kostas Bekris. Spring-rod system identification via differentiable physics engine. *arXiv preprint arXiv:2011.04910*, 2020.

[27] Web tutorial 1. `https://code.tutsplus.com/series/how-to-create-a-custom-physics-engine--gamedev-12715`. Accessed: 2024-10-5.

[28] Web tutorial 2. `https://www.youtube.com/watch?v=-_IspRG548E`. Accessed: 2024-10-5.

[29] Gang Yang, Siyuan Luo, and Lin Shao. Jade: A differentiable physics engine for articulated rigid bodies with intersection-free frictional contact. *arXiv preprint arXiv:2309.04710*, 2023.

Appendix A

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Building a Physics Engine From Scratch

17 Oct 2023

Introduction

Physics Engine, a term frequently used in video game industry and science research, is a tool that simulates physical phenomena using a computer. The first computer, ENIAC, at one point used a physics engine to help design military weapons[18]. At the core of physics engine, semi-realistic results are obtained through a combination of computation-efficient numerical approximations, careful modelling of the objects, and sometimes clever hacks that enable the engine to just make the cut.

There are some common terminologies in physics engines.

- **Rigid body** is an individual object that will be simulated in the engine. A rigid body is an object that does not distort or bend, as opposed to soft body or fluid, which gives rigid bodies the benefit of simplicity. A typical physics engine needs to keep track of its mass, position, orientation, linear velocity, angular velocity and impulse.
- **Collider** is a part of a rigid body. Complex rigid bodies tend to get separated into simple, convex colliders like spheres and boxes in the pipeline.
- **Collision detection** and **Collision resolution** are components of the engine which handle the interaction between colliders. Commonly physics engine continuously advances the time by a small fraction of a second, moving the objects according to their speed. After each position update, Collision detection will kick in and detect if a pair of colliders will intersect. If so, Collision resolution will decide whether and how they will get bounced and separated apart, updating their velocities accordingly.

Many open-source physics engines are available online, including PhysX[16], Box2D[4] and Bullet[5]. However, the most common way a physics engine is presented is as a big, mysterious library where all the computation is done under dozens of dependencies and documentation. As a result, adding a simple feature could take a lot of effort of plowing through documentation and files. I want to build my own physics engine, which gives me the flexibility to add whatever I want because I would know exactly how it works.

Starting point

Describe existing state of the art, previous work in this area, libraries and databases to be used. Describe the state of any existing codebase that is to be built on. I am already able to write prose using the English language. I have an online dictionary, etc.

Description

My project aims to build a 3D real-time physics engine from scratch that implements basic modules, without making use of any currently available physics engine. My focus will be to realise the visual effects rather than being extremely accurate or efficient, considering the limited time, my available hardware resources, and the ease of experiments and showcases. This is also why I chose to build a real-time physics engine over a high-precision one. In addition, this project will also provide a basic framework that is easy to extend upon in the future, if necessary. Overall, this project will be a great opportunity for me to learn more about physics and programming.

Here is a list of the basic core modules I plan to implement:

- Object modelling: Create data structures for recording attributes of rigid bodies and design interfaces for applying forces and adding colliders. Typical physical attributes of rigid bodies include mass, position, orientation, linear velocity, angular velocity, and impulse.
- Collision detection: Adopt many algorithms to decide if collisions happen.
- Bounce: Part of Collision resolution where some maths are used to find the approximated results of a bounce.
- Friction: Part of Collision resolution where vast assumptions are used to simulate real-life friction.
- Stability: Part of Collision resolution where some heuristics are used to prevent certain visual artefacts. For example, if two objects are stacked together, the upper one should not be bouncing up and down constantly.

The project could then be evaluated by comparisons using simple experiments against existing popular physics engines. More on this in the Evaluation section.

Extension modules I plan to look at include:

- Fluid simulation:

  Fluid simulation involves approximations to fluid equations, and can have different levels of complexity depending on the topic. For example, the simulation of buoyant force of hard objects submerged in water will be simpler than the simulation of the flow of the fluid. I will try my best to cover as much in fluid simulation as I can.

- Real-time rendering:

  For the sake of showcasing, I might be using existing rendering libraries like Blender[2], but it is certainly better to not rely on them.

- Performance evaluation

  I will give different implementations for CPU and GPU, which will then allow me to draw comparisons about their contributions to performance.

- Soft-body simulation

  Unlike rigid bodies, the shape of objects can change to a certain degree. Therefore, it is much harder to model them and deal with collisions. I will need to do research and choose what to implement.

Evaluation

The evaluation of this project is split into three parts: Benchmark selection, Core functionality, and Performance evaluation.

Benchmark selection

Since the functionality evaluations will be largely comparison-based, I will be choosing at least three open-source physics engines that support similar features to compare against. The following engines have been found as possible candidates:

Physics engine	Website
Advanced Simulation Library	asl.org.il
Bullet	pybullet.org
Newton Game Dynamics	newtondynamics.com/forum/newton.php
Open Dynamics Engine	www.ode.org
PAL	www.adrianboeing.com/pal
PhysX	www.nvidia.com/en-gb/geforce/technologies/physx
Project Chrono	projectchrono.org
Siconos	nonsmooth.gricad-pages.univ-grenoble-alpes.fr/siconos
SOFA	www.sofa-framework.org
Tokamak physics engine	github.com/isegal/TokamakPhysics

I plan to further narrow them down by testing if they work in my environment, if they have sufficient documentation, and if they support the features I am going to compare on.

Core functionality

I will evaluate the accuracy of the physics engine by setting up experiments like pendulums and boxes sliding down inclined planes in my physics engine. I will then compare how the rigid bodies move in the engine with the well known physical results of how these systems behave in real life.

Performance evaluation

This part is considered as an extension. The performance will similarly be measured with an experiment.

There will be comparisons between my engine and other benchmark engines, as well as between different implementations of my engine.

I will define, measure and then plot the average computational time of a time step, while I gradually make the scene more complicated by adding in more objects.

Starting Point

Personally, I have some basic knowledge of programming in C++, having completed a small project using it during my internship. My graphics knowledge only comes from the two Part IA and IB graphics courses.

Additionally, I have done a bit of research online, and am therefore decently confident about the abundance of tutorial resources, despite not actually having spent time reading through them. For example, a full video series about physics engine is available on the Internet[1]. I might also make use of 3D graphics libraries like CUDA[7], but as of now I have no prior working experience with them.

The aforementioned open-source physics engine libraries are also available for me to look into some possible solutions or draw comparisons, but I will not be using their simulation code in my project.

Success criteria

For the core:

- Implement all basic modules: Object modelling, Collision detection, Bounce, Friction, Stability.
- Evaluate the engine by comparing it with popular existing engines with simple experiments.
- Demonstrate that the engine works with screenshots of simple examples.

For extensions (ordered by priority):

- Implement fluid dynamics.

- Implement different versions of the engine for whether GPU is used, and for other interesting parameters like the number of cores used. Then evaluate the performance.
- Implement real-time rendering, which should allow the project to meet all previous criteria without third-party rendering libraries.
- Implement soft-body dynamics.

Work plan

Michaelmas term

Now - 15 Oct

Write project proposal
    Research for libraries to use
    Environment setup
    Milestone: Complete full project proposal

16 Oct - 29 Oct

Set up the project framework
    Implement basic interfaces into rendering libraries
    Milestone: Able to produce a blank video or an empty interactive demo

30 Oct - 19 Nov

Implement Object Modelling

20 Nov - 3 Dec

Implement Collision Detection

Christmas break

4 Dec - 17 Dec

Implement Bounce, Friction

18 Dec - 14 Jan

Implement Stability
    Milestone: Complete the core

Lent term

15 Jan - 28 Jan

Buffer phase for core implementation
Core evaluation if core is completed
Write progress report
Milestone: Completed a draft of the progress report

29 Jan - 15 Feb

Buffer phase for core evaluation
Start extension implementations
Finish progress report
Milestone: Submit the Progress Report (Deadline 2 Feb)

16 Feb - 1 Mar

Implement extensions
Start dissertation write up

1 Mar - 15 Mar

Wrap up implementations
Continue writing the dissertation
Milestone: Implementations completed

Easter break

16 Mar - 1 Apr

Complete a draft of the dissertation, available for view and feedback
Milestone: First draft of dissertation completed

2 Apr - 22 Apr

Improve the dissertation based on the feedback
Milestone: Second draft of dissertation completed

Easter term

23 Apr - 1 May

Improve the dissertation based on the feedback
Milestone: Third and final draft of dissertation completed

2 May - 10 May

Finalise the dissertation

Milestone: Submit the final dissertation (Deadline 10 May)

Resource declaration

- I will be using my personal laptop (specs) as my main working device.
- For backup and workflow tracking, I will make use of GitHub, Google Drive, and Overleaf
- For development, I will be using rendering libraries like Blender, as well as GPU interfaces such as CUDA.
- As a backup plan I have another laptop for working.