JOHANNES KEPLER
UNIVERSITY LINZ

Implementing a simple 2D physics engine

Author
Dominik Staudinger

Submission
Institute for System Software

Thesis Supervisor
DI Lukas Makor

March 2024

Bachelor Thesis
to obtain the academic degree of
Bachelor of Science
in the Bachelor's Program
Informatik

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Bachelor's Thesis

Implementing a simple 2D physics engine

Student: Dominik Staudinger

Advisor: Dipl.-Ing. Lukas Makor, BSc

Start date: October 2023

Dipl.-Ing. Lukas Makor, BSc
Institute for System Software
T +43-732-2468-3435
lukas.makor@jku.at

Physics engines are used in various areas, e.g., game development, simulation, and movies. Typically, physics engines are categorized into 2D and 3D physics engines.

As part of this thesis a simple 2D physics engine, supporting discrete collision detection and collision resolution for circles, axis-aligned bounding boxes and arbitrary polygons should be implemented. Additionally, forces such as gravity should be considered and simple rotations for axis-aligned bounding boxes should be implemented. Only rigid bodies need to be considered for this work.

Goals of this thesis:

- Research up-to-date and reasonable algorithms for collision detection and collision resolution
- Implement a simple physics engine as described above
- Implement a simple 2D visualization to visualize the state of the world simulated by the physics engine
- Setup test scenes to test and illustrate the correctness of the physics engine

<u>Modalities:</u>
The progress of the project should be discussed at least every three weeks with the advisor. A time schedule and a milestone plan must be set up within the first four weeks and discussed with the advisor. It should be continuously refined and monitored to make sure that the thesis will be completed in time. The final version of the thesis must be submitted not later than 31.03.2023.

Abstract

Physics engines are utilized for a variety of use cases, including, but not limited to, video games, simulations concerning flight or driving, and mechanical simulations. They rely on mathematics to define entities and their behaviour in an imitated physics world. Engines enable simulations to replicate the laws of physics with varying degrees of realism. Generally, physics engines are separated into two classes: real-time and high-precision engines. High-precision engines rely on highly accurate and compute-intensive algorithms to calculate precise and realistic simulations. In contrast, real-time engines utilize approximations to enable very fast simulations, which are often used for entertainment applications. The aim of this work was to develop a simple real-time 2D physics engine, namely *Rustycs*. It is easy to understand and written in a programming language called Rust.

Kurzfassung

Physik-Engines werden in diversen Gebieten verwendet, beispielsweise in Videospielen, Flug- und Fahrsimulationen oder auch mechanische Simulationen. Sie verwenden mathematische Konzepte, um Objekte und deren Verhalten in einer simulierten Welt zu definieren. Diese Definitionen ermöglichen es, dass Engines die Regeln der Physik in Applikationen replizieren können. Die Genauigkeit mit der die Physik repliziert wird variiert. Grundsätzlich gibt es zwei verschiedene Arten von Physik-Engines: Echtzeit-Engines und präzise Engines. Präzise Engines verwenden hochakkurate und rechenintensive Algorithmen, um realistische Simulationen zu berechnen. Echtzeit-Engines hingegen benutzen Annäherungen, um effizientere Simulationen zu ermöglichen. Daher werden Echtzeit-Engines oft in Entertainment-Applikationen verwendet. Das Ziel dieser Arbeit war, eine einfache Echtzeit-2D-Engine namens *Rustycs* zu implementieren. Sie ist einfach zu verstehen und wurde in der Programmiersprache Rust entwickelt.

Contents

1 Introduction

The task of simulating physics has changed considerably in the last few decades. In the past, it was necessary to program engines from scratch, if a solution in a given programming language did not exist. This was a lengthy and error-prone process, due to the lack of proper resources compared to today. Nowadays, there are all kinds of physics engines written in various programming languages, and some even have built-in two-dimensional and three-dimensional capabilities. More importantly, numerous physics engines are open source and free to use for developers. For instance, *Box2D* is a relatively simple 2D physics engine that takes care of the complex aspects of simulating physics [1].

The granularity and realism needed in separate domains differ greatly. A simple two-dimensional platformer will certainly not require advanced or realistic physics. On one hand, the realism of simulations could influence the game experience negatively, on the other hand, more realistic physics require expensive calculations, which could impact performance. In contrast, a realistic fluid simulation in 3D space requires sophisticated maths and intricate optimizations.

As there are multiple existing engines that cover a lot of use cases ranging from simplified 2D physics to very realistic 3D simulations, one might wonder if it is worthwhile to "reinvent the wheel" [1, 2]. Of course, this depends on what kind of physics a given application needs. Specifically, in 3D space, implementing an efficient and accurate physics engine is a monumental task. Despite this, understanding the fundamental concepts utilized in such engines to a certain extent is beneficial in many domains. Programming a physics engine from scratch enables the developer to define constraints and approximations suiting the explicit needs of the application. It teaches problem-solving skills, software design, critical thinking, and general concepts like optimizing and writing clean code.

When researching resources about physics engines, it quickly becomes clear that the domain and existing implementations are quite abstract and difficult to understand [3]. This highlights one of the main focus points of this paper: coherence of the concepts that are used. The goal of the engine implementation is to be easy to understand and to utilize concepts that are beginner friendly. This allows us to focus more on the core concepts of the engine, rather than optimizing every possible aspect of it.

However, we did not ignore aspects like time and space complexity, or memory safety. To address these issues we took a different approach than most implementations out there. We tackle several facets of these issue domains at the language level [4, 5]. Rust is quite unique in terms of memory management, because it employs the concept of Ownership & Borrowing [6]. It has no required manual memory management and no garbage collector. Of course, this concept was not invented by Rust, but it is the first general-purpose programming language that uses this concept past an experimental level. The language also offers concepts like zero-cost abstractions, which enable the utilization of higher-level code paradigms, such as iterators, without an additional run-time overhead.

Summarizing the goals of this work, it was in our interest to develop a coherent physics engine in a modern language like Rust. More importantly, we aimed to research techniques and concepts used in physics engines and to apply or compare them to our implementation.

The rest of the paper is structured as follows: Section 2 explains terminology and concepts relevant to this work. It also provides a rationale as to why we decided to use Rust to implement our engine. In Section 3, we elaborate on important implementation aspects regarding the engine itself, and its visualisation. Section 4 evaluates the integrity of the engine by discussing demo scenes and the visual debugger we have implemented. In the 5th Section, we focus on the lessons we learned during the implementation regarding physics engines themselves. Additionally, it evaluates some Rust-specific factors that impacted the development process in various ways. In Section 6, we consider the limitations of the engine and outline potential future work. Lastly, we conclude this thesis in Section 7.

2 Background

This section provides the necessary information to reason about the topics at hand. Section 2.1 roughly describes what physics engines are, lists different characteristics of them, and explains how basic concepts like bodies can be modelled within the simulation. In Section 2.2 we present Rust, the programming language the engine is implemented in, and some of its advantages. Section 2.3 elaborates crucial mathematical branches, that are needed to simulate physics.

2.1 Physics Engines

Physics engines encompass the simulation of physical systems such as rigid body dynamics [7], soft body dynamics [8], or fluid dynamics [9]. These engines are utilized in various domains to simulate realistic behaviour of physical entities. To define how such entities interact with each other engines make use of mathematical concepts and formulate rules or constraints with them. While simulations made with physics engines can adhere to the expectations of reality, they are not required to. In Figure 1, we depict a simple example: We can define gravity to model its observed behaviour, going straight down. However, nothing prohibits us to utilize a custom gravity direction that is inclined to the right.

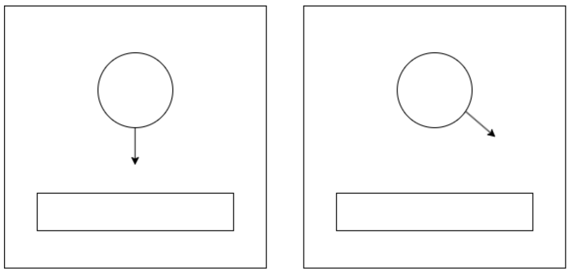


Figure 1: Standard gravity compared to a custom gravity.

The primary tasks of physics engines are to move dynamic bodies through a simulated space, as well as detecting and resolving their collisions with other entities. Those entities could be other moving bodies, immovable bodies or even the boundaries of the world. Unless the space is empty collisions are inevitable and the engine has to detect and handle them according to a set of defined rules. In two-dimensional engines, it is comparatively simple to analyze whether two bodies collide. Collisions are the intersections between bodies, or rather intersections between the shapes of the bodies. Figure 2 illustrates a simple collision between two circle-shaped bodies. In Section 3, we will explore how such a collision resolution can be implemented.

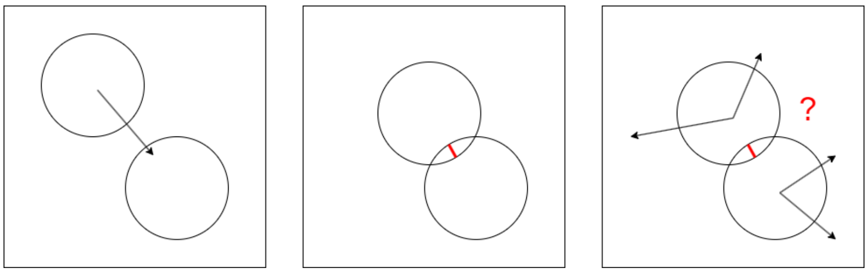


Figure 2: Example of a simple collision between two circles.

Physics engines can be classified by numerous characteristics. One rather obvious distinction is, whether the engine operates in two dimensions, like *Rustycs*, or in three dimensions. The absence of a whole dimension makes two-dimensional engines comparatively simple and they are the main focus of our work. On the other hand, three-dimensional engines are very compute-intensive and require their developers to have a detailed understanding of various physical and mathematical concepts.

The detection of collisions is an important topic for physics engines and multiple techniques to enable it have been developed. *Discrete Collision Detection (DCD)* describes the process of evaluating the world state at discrete points in time (ticks) and suffices in a lot of use cases. Its shortcomings become apparent when working with bodies that move particularly fast, also known as *bullets.* In *DCD* it is possible that *bullets* move so fast that they completely pass a wall or other entities in between ticks, and therefore no collision gets detected. *Continuous Collision Detection (CCD)* prevents such evaluation artifacts by predicting and calculating collisions between ticks, which adds additional computational overhead [10]. Furthermore, it requires even more resources to calculate additional properties of collisions, such as the actual time of impact.

Various techniques exist to resolve detected collisions. For instance, one way to resolve collisions is called the projection method. This technique simply displaces colliding bodies to separate them again, but by itself, it does not account for the transferal of energy of colliding entities. An alternative resolution is the impulse method, which does not change the body locations directly, but rather applies instantaneous impulses to their velocities to separate them.

Bodies within physics engines are typically at least defined via their shape and whether they can move or not. In the context of a physics engine immovable bodies are called static and have infinite mass. In contrast, dynamic bodies can move through the physics world and are affected by forces within it. In addition to world forces, collisions also affect how bodies move since they have to be resolved. Furthermore, in *Rustycs* body shapes are classified into four categories: circles, axis-aligned bounding boxes (AABB), oriented bounding boxes (OBB) and polygons. Note that the main difference between AABBs and OBBs is that OBBs can rotate, whereas AABBs are unable to. Two-dimensional vertices are used to define the concrete shapes of AABBs, OBBs and polygons. Circles are not defined via vertices and we will discuss how we model them in Section 3.

Combinations of an arbitrary number of vertices theoretically allow us to define infinite distinct polygon shapes. However, within our implementation, we restrict valid polygon shapes to be convex and to use at most 8 vertices [11]. The subset of convex polygons separate themselves from the set of all polygons via a special characteristic in regard to their shape. Every point on any line segment between two arbitrary points within a convex polygon remains within the boundaries of the polygon. Restricting the engine with these boundaries improves performance and simplifies collision handling. The *Separating Axis Theorem (SAT)* and algorithms utilizing it are commonly used to detect collisions, and it assumes involved shapes to be convex [12]. Figure 3 illustrates the difference between a convex polygon and a concave polygon.

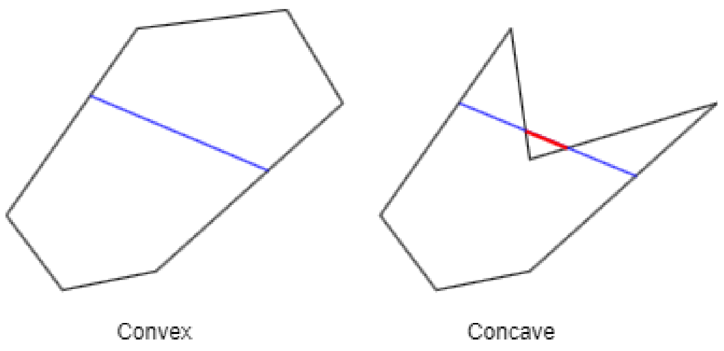


Figure 3: Difference between convex and concave polygons.

2.2 Rust

Rust is a general-purpose programming language initially designed by Graydon Hoare with a focus on performance, reliability, and productivity. It utilizes *LLVM* to compile its code directly to assembly [13]. However, being a compiled language is not its only performance advantage. Zero-cost abstractions enable developers to use high-level programming patterns, for instance, iterators, without worrying about performance impacts. Another performance factor is that the language manages memory uniquely. It leverages the concept of Ownership & Borrowing to manage its memory [6]. Ownership rules define the variable, which holds a value, to be the owner of that value. When the owner goes out of scope, the value will be dropped or rather deallocated. Developers can specify how to handle the owner in the context of their program. Values of owners can be borrowed to prevent them from being dropped when passing their value around, for example as function parameters. Consequently, Rust does not have a garbage collector and its associated run-time costs, nor does it require developers to manually handle memory like C or C++. Ownership also impacts the reliability regarding the memory safety of the language. Rust offers a rich type system and a sophisticated compiler, which enable it to find most errors at compile-time. The type system is very explicit and implicitly reveals how various concepts, for example, memory allocation, work. The language is well documented and even offers an integrated package manager, called cargo, to boost the productivity of developers. In addition, there are numerous other aspects, like its active and maintained community-driven ecosystem [14], which make Rust a great choice in various programming domains.

2.3 Mathematics

The primary tasks of physics engines involve the movement of bodies and the resolution of collisions within their simulation. Implementing these features requires a basic grasp of specific mathematical concepts. Two central domains of mathematics for this purpose are vectors and trigonometry. Vectors are used to reason about where objects are in the environment and how they are moved accordingly. Trigonometry allows us to extract collision data and to apply rotations to dynamic objects. Consequently, a fundamental understanding of these domains is of utmost importance in the development of physics engines.

In two-dimensional space, locations can be represented by two-dimensional vectors (x, y), where x represents the horizontal axis and y the vertical axis. Similarly, forces and velocities can also be described by vectors, representing the magnitude and direction of movement or acceleration. Trigonometry allows us to define collision resolutions and how to rotate bodies in our simulated space. For instance, to rotate any object around the world origin $(0, 0)$ by a given angle θ in two-dimensional space, one can observe the 2×2 rotation matrix R.

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{1}$$

Translating this into code is rather straightforward and by itself not a complicated operation. In Listing 1, we take the absolute world location of a body as (x, y) and an *angle* and perform the correct calculation according to the rotation matrix above:

Listing 1: Rotation around the world origin

```
fn rotate_body_location(
    x: f32, y: f32, angle: f32
) -> (f32, f32) {
    let x_new = x * angle.cos() - y * angle.sin();
    let y_new = x * angle.sin() + y * angle.cos();
    return (x_new, y_new);
}
```

However, if we want to rotate the body around its own center, this does not work, since we actually want to rotate the shape, or more specifically, the vertices that define the shape. Therefore, it is necessary to translate the vertices to the world origin before rotating the vertices themselves. Afterwards, we translate the vertices back to the actual location of the body. An exemplary function that does this can be viewed in Listing 2.

Listing 2: Rotation of a body

```
fn rotate_body_vertices(
    vertices: Vec<(f32, f32)>, x: f32, y: f32, angle: f32
) -> Vec<(f32, f32)> {
    for (v_x, v_y) in vertices {
        (v_x, v_y) = (v_x - x, v_y - y);
        (v_x, v_y) = rotate(v_x, v_y, angle);
        (v_x, v_y) = (v_x + x, v_y + y);
    }
    return vertices;
}
```

3 Implementation

This chapter describes the implementation process of our physics engine *Rustycs* and its corresponding demo application. The details regarding the physics engine are presented in Section 3.1. Section 3.2 elaborates optimization approaches. Two approaches are applicable generally and another one is specific to physics engines. Lastly, Section 3.3 presents implementation details regarding the demo application used to visualize the state of a *Rustycs* world.

3.1 Engine

We implemented a two-dimensional physics engine that checks for collisions in discrete time steps (DCD) and resolves them via instantaneous velocity impulses. The engine evaluates a subset of possible body shapes, which are illustrated in Figure 4. The locations of all bodies and the shapes of AABBs, OBBs and polygons are defined by two-dimensional vectors. Defining the shape of circles with vectors is not practical, since it is impossible to store enough vectors to represent a smooth circle shape. Instead, we define circle shapes via their radius, since this is sufficient and cheap in terms of detecting shape intersections. Their radii in combination with the concrete body locations allow us to analyse whether the circles collide with other entities. Furthermore, the bodies are considered to be rigid, which means they are unable to deform.

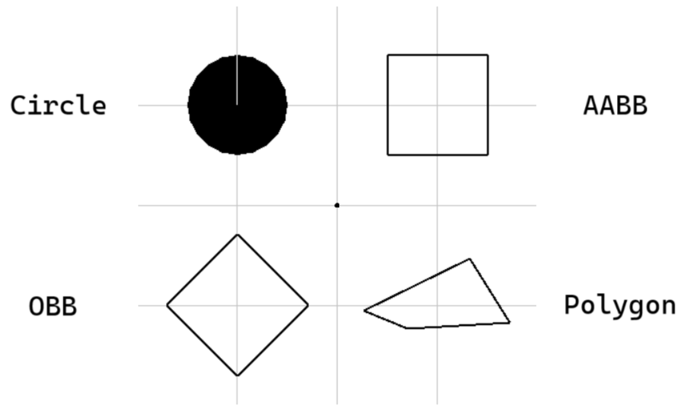


Figure 4: Illustration of all supported shapes.

In Figure 4, the center of the grid is the world origin $(0, 0)$ and every grid box has a width and height of 1. Consequently, the AABB has a world location of $(1, 1)$ and its top-left corner is located at $(0.5, 1.5)$. However, in our engine, shape vectors are stored inside of bodies, relative to the location of the body they belong to. For this reason, the top-left corner of the AABB is actually stored as $v'_{corner} = v_{corner} - v_{body}$. In Figure 4, this results to the vector $(0.5, 1.5) - (1, 1) = (-0.5, 0.5)$. Storing the shape vectors like this allows us to rotate the shape without having to translate the vertices to the world origin for each rotation. Additionally, we do not need to move individual shape vertices when moving bodies, just their location. When we need the absolute locations of the shape vertices, for instance during rendering or collision detection, we simply offset them with the body location.

Evidently, vectors are fundamental building blocks when reasoning in two-dimensional or three-dimensional space. Implementing custom representations of vectors to enable efficient implementations tailored to the necessary features is a common practice in engine development. Therefore, we decided to design our own vector data structure, namely *Vector2*. In many other languages, we would implement such a structure as a class, but classes do not exist in Rust. Instead, Rust offers simple structures, similar to other system programming languages. Listing 3 showcases the *Vector2* struct as well as how to extend it by utilizing an implementation block. These blocks are used to define methods or associated functions on structures, which enables us to extend them with the needed functionality. The simplicity of our homemade vector struct allowed us to implement vector functionalities concisely. Furthermore, the flexibility of the implementation blocks enabled us to focus only on crucial vector concepts needed by *Rustycs*.

Listing 3: The Vector2 struct

```rust
pub struct Vector2 {
    pub x: f32,
    pub y: f32,
}

impl Vector2 {
    // method
    pub fn rotate_self(&mut self, angle: f32) {
        let x = self.x;
        let y = self.y;

        self.x = x * angle.cos() - y * angle.sin();
        self.y = x * angle.sin() + y * angle.cos();
    }

    // associated function
    pub fn rotate(v: Vector2, angle: f32) -> Vector2 {
        return Vector2 {
            x: v.x * angle.cos() - v.y * angle.sin(),
            y: v.x * angle.sin() + v.y * angle.cos(),
        };
    }
}
```

In addition to defining locations within the world, vectors also describe location changes and other body characteristics, such as their velocity. When we want to move a body located at $L = (5, 7)$ to a target location of $T = (10, 3)$, we can calculate their difference M as $T - L = (5, -4)$. By adding M to L, the body will reach the target location T instantaneously, and consequently, the body will seemingly teleport. However, in our context, teleportation completely disregards the time needed to execute physical movement. *Rustycs* defines a concrete time interval, used to explicitly probe the world state and update it accordingly. It is common practice to declare the inverse time interval in the form of a fixed tick rate. For instance, if the evaluation has a tick rate of 64, we probe the world state 64 times each second, which results in a probing interval of $1/64 = 0.015625$ seconds. We call the probing interval the "delta time", and every location or velocity change within the simulation gets scaled with it. Without this consideration, movement is going to be inconsistent across different tick rates.

3.1.1 Forces & Attractors

Dynamic bodies can be moved in various ways in a simulated physics world. For instance, bodies can be moved by a fixed amount every tick, which is a concept known as linear velocity. Acceleration can be utilized to change the velocity of the body; this results in nonlinear movement that changes over time. Additionally, we can adjust the location of a body directly and simulate its teleportation. Regardless of the specific method chosen, movement in two-dimensional space is modelled by vectors in the form of (x, y).

To move bodies, *Rustycs* primarily relies on forces and attraction entities that we call "attractors". Forces affect all dynamic bodies within the physics world. This is comparable to gravity in reality, but we can simulate other forces, such as wind, with this concept as well. Attractors pull dynamic bodies towards them and are categorized into global and local attractors. Global attractors affect all bodies in the world similar to forces, whereas local attractors have a maximum range and do not affect bodies outside of it. Forces and attractors constantly affect body locations and allow us to simulate approximations of real world behaviour.

Our engine is not limited to a single force per simulation. However, separate forces get accumulated into a single world force, which accelerates and affects dynamic bodies in the simulation. This is functionally equivalent to the sequential application of individual forces but more efficient in terms of run-time overhead. For instance, we can create a force called "gravity" with an acceleration of $(0, -9.81)$. Next, we can simulate a strong wind in our world that flows to the right with an acceleration of $(2, 0)$. These two forces result in a total world force of $(2, -9.81)$ which accelerates bodies by $2.0m/s^2$ to the right and $9.81m/s^2$ downwards.

Like forces, global attractors influence all dynamic bodies within the simulation. However, instead of a predetermined direction and acceleration, they calculate these factors dynamically at run-time. Bodies are drawn towards the attractors, which means the direction and magnitude of their attraction is distinct for individual bodies. In addition, their magnitude depends on the mass of the attractor and the body, as well as the distance between them. In every physics step we have to calculate the attraction force for each body and consider all attractors. These attractions are accumulated and applied to the respective body. Local attractors have an additional maximum range. Bodies outside of the local attractor range do not get pulled towards them and are ignored by the attractor.

3.1.2 Bodies

The shape is not the only important characteristic of a body for a physically accurate simulation in *Rustycs*. The engine is aware of various characteristics, such as the body's material, or whether it is dynamic or static. The *Body* structure contains the following data:

- *Shape*: An enum used to analyse whether bodies intersect with others. In Rust, enums are special since they are not simple indicators of what they represent. Instead, Rust enums can associate an arbitrary data structure to their variants. For example, the circle shape variant holds a circle structure. This circle structure contains the radius of the circle and a single vector to represent the circle orientation. The area of the shape is utilized to dynamically calculate the body mass upon its creation. Furthermore, depending on the shape variant, different detection algorithms get applied in the collision pipeline.
- *Transform*: A structure that keeps track of the body location, velocity, and angular velocity. In *Rustycs*, the body transform state gets affected by forces, attractors, and collisions between bodies.
- *Material*: A structure containing values for the density, restitution, and friction of bodies. In *Rustycs*, the body material is used to define the body mass and inertia when bodies are created. Furthermore, it allows us to dynamically resolve collisions based on the material types of the colliding bodies. Specifically, material restitution and friction coefficients influence collisions.
- *BodyType*: An enum that determines whether a body is dynamic or static. It is crucial to distinguish between dynamic and static bodies, since they have to be handled differently in various aspects during the physics step.
- *Mass*: A numerical value that represents the mass of the body. It depends on the shape of the body and the density of the body material. In *Rustycs*, the body mass is used to define how a body is affected by attractors and collisions. In impulse based physics engines it is common practice to associate the absence of mass with immovable objects to simulate the effect of "infinite" mass. As a consequence, static bodies have a mass of 0. The reason for this lies in the way collisions are resolved with impulses, which we will discuss in Section 3.1.3.
- *Inertia*: A numerical value that represents the inertia of the body. It describes the tendency of an object that is already in motion to stay in motion. Similarly, it

also describes the tendency of objects at rest to stay at rest. Its value depends on the specific body shape and is proportional to the body mass. Inertia is commonly used to define the rotational behaviour of different objects in physics engines. In *Rustycs*, static bodies have an inertia of 0, for similar reasons that cause them to have 0 mass.

Finally, we also store the inverse mass and inverse inertia of the body. This enabled us to implement optimizations, which will be discussed in Section 3.2.

3.1.3 Collision Detection & Resolution

One of the main tasks of physics engines is to handle collisions, which happen when the shapes of bodies intersect. Whether intersections occur is easy to determine visually, but a physics engine is unable to do that. Consequently, engines have to approach this problem differently. They utilize algorithms that rely on mathematical characteristics of the body's geometry to detect intersections. The *Gilbert-Johnson-Keerthi (GJK)* algorithm is commonly utilized to detect collisions between bodies in real-time applications such as physics engines [15]. However, since it is designed for arbitrary shape types, it is quite abstract and therefore difficult to understand. *Rustycs* took a different approach and explicitly checks for collisions between concrete shape types. One the one hand, this improved the coherence of the detection algorithms. On the other hand, it was necessary to implement explicit algorithms for each possible shape combination.

The simplest shape combination in terms of detecting an intersection is a collision between two circles. Circle bodies are represented by two characteristics: their location in the world and their shape, which is defined by the circle radius. Listing 4 showcases an algorithm to detect intersections between two circle bodies via their characteristics. First, we compute the difference between the circle locations as a vector. Next, the sum of their radii is calculated. Finally, we can return whether the length of the difference vector is smaller than the sum of the radii.

Listing 4: CircleCircle collision detection

```rust
fn circle_circle_b(a: &Body, b: &Body) -> bool {
    let distance = b.transform.location - a.transform.location;
    let r = a.shape.as_circle().r + b.shape.as_circle().r;

    distance.len() < r
}
```

Additionally, Figure 5 illustrates circles A and B intersecting and compares the radii sum to the respective distances. Evidently, if the distance between them is smaller than their radii sum, the circles intersect.

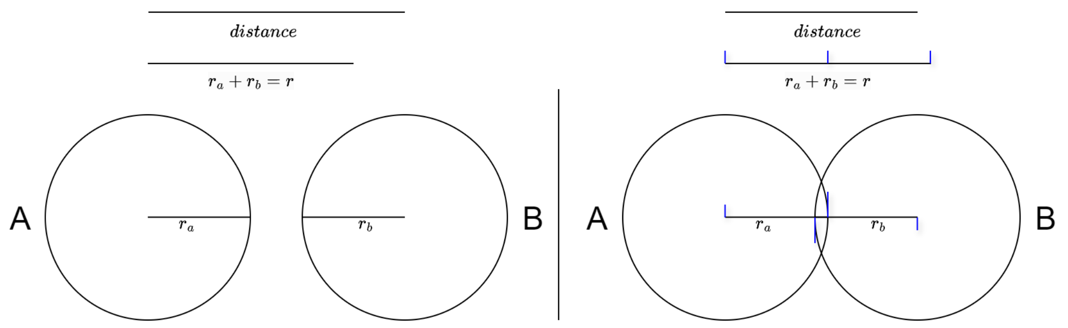


Figure 5: Comparison of circles that do not intersect vs. circles that do intersect.

The detection function in Listing 4 is only useful in terms of detecting whether a collision happened. However, it provides no actual information on how to resolve the collision and separate the bodies accordingly. The collision detection functions of *Rustycs* extract important data after the collision was detected, and stores it in manifolds. Collision manifolds enable us to streamline collecting collision data. They contain the collision information that is needed to resolve collisions in the resolution phase. Information of interest includes characteristics such as how deep the intersection is (collision depth), in which direction the bodies should be moved (collision normal) and where exactly the bodies collided (contact points).

Listing 5 showcases the collision detection implementation of two circles in *Rustycs*. Instead of returning if an intersection occurs, we use that knowledge to generate a manifold if necessary. When a collision gets detected, we generate an empty manifold as seen in line 9. During this initial construction of the manifold, the average friction and bounce coefficients of the bodies get inferred from the involved body materials. For "circle-circle" collisions we calculate the collision normal, depth and contact point and store them in the manifold before returning it. After detecting the collision and generating its manifold, the collision is almost ready to be resolved.

Listing 5: CircleCircle collision detection with manifold generation

```
1 fn circle_circle(
2     a: &Body, a_idx: usize, b: &Body, b_idx: usize
3 ) -> Option<Manifold> {
4     // ...
```

```
    if distance.len_squared() >= r * r {
        return None;
    }

    let ra = a.shape.as_circle().r;
    let mut m = Manifold::new(a, a_idx, b, b_idx);

    let normal =
        distance.normalized().unwrap_or(Vector2::rand_normal());

    m.normal = normal;
    m.depth = r - distance.len();
    m.contacts[0] = a.transform.location + (normal * ra);
    m.contact_count = 1;

    Some(m)
}
```

Collisions between other shapes other than circles are harder to detect. *Rustycs* relies on the *Separating Axis Theorem* and two-dimensional vector projection techniques to extract information from collisions that do not exclusively involve circles [12]. Furthermore, collisions that involve no circles at all are special. Figure 6 depicts how their collisions might involve multiple contact points, which are limited to at most two in two-dimensions. If non-circle bodies collide on their surfaces in parallel, the collision needs to involve two contact points to enable a proper resolution.

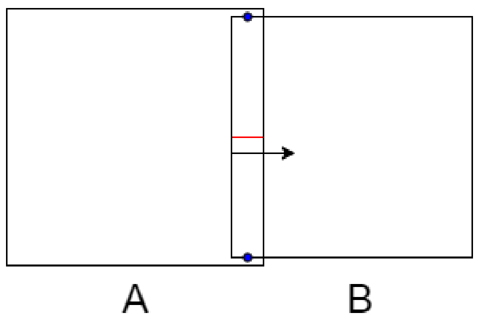


Figure 6: Example of an AABB collision with two contact points its collision normal and depth.

When all precomputable coefficients are calculated and stored in the manifold we can resolve the collision with velocity impulses. We calculate velocity impulses sequentially in separate resolution iterations and apply them to the bodies [16, 17]. By adjusting the number of resolution iterations we control how accurate the resolution is. The more velocity impulses get calculated, the more accurate the resolution. Furthermore, we apply multiple impulses to the body velocities so that we can accumulate all velocity changes of a physics step and apply them at the end. This prevents new collisions from happening during the collision resolution process, which would require us to scan for collisions before each resolution iteration.

Each contact point of a collision requires two impulses, to consider the separation as well as the friction during the resolution. Listing 6 shows how resolutions apply impulses to the body's velocity and angular velocity. Apart from the calculated impulse magnitude, the velocity impulse depends on the distance between the locations of the involved bodies and the concrete contact point locations.

Listing 6: Applying resolution impulses

```rust
fn apply_impulses(
    a: &mut Body, b: &mut Body,
    impulse: Vector2, ac: Vector2, bc: Vector2
) {
    a.transform.velocity -=
        impulse * a.inverse_mass;
    a.transform.angular_velocity -=
        ac.crossed(impulse) * a.inverse_inertia;

    b.transform.velocity +=
        impulse * b.inverse_mass;
    b.transform.angular_velocity +=
        bc.crossed(impulse) * b.inverse_inertia;
}
```

In general, resolution impulses are proportional to the relative velocity between the involved collision bodies. The rotation impulse correlates to the collision normal, whereas the friction impulse correlates to its tangent. Figure 7 illustrates how a rotational impulse gets applied to the angular velocity of an OBB. Figure 8 illustrates how a friction impulse gets applied to the angular velocity of a circle. However, simply applying these impulses is not enough to separate the bodies. *Rustycs* does not correct the body positions via the resolution impulses. Instead, it adjusts the body locations according to the collision depth after the velocity changes have been applied. In addition to the collision depth, the positional correction depends on the masses of the involved bodies. For instance, if the involved bodies have distinct masses, the body with less mass gets displaced more than the heavier body. If a static body is involved in any collision, it will not be displaced since its mass is infinite and modelled as zero and the dynamic body gets displaced by the total collision depth.

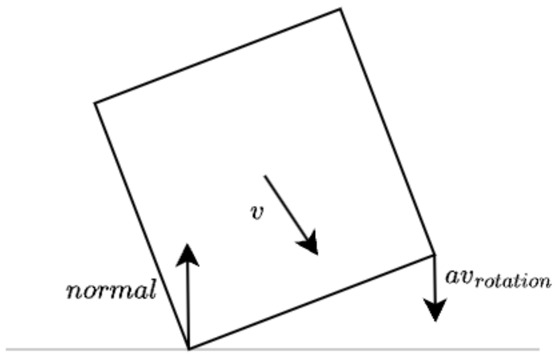


Figure 7: Visualisation of a rotational impulse.

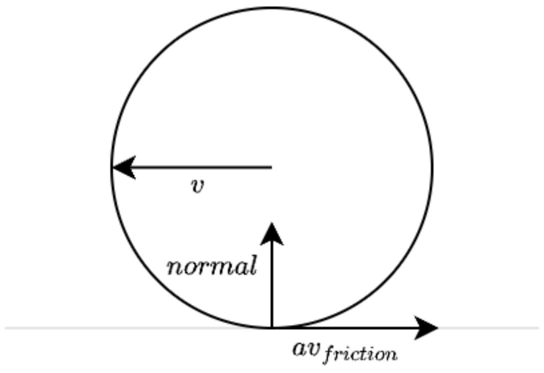


Figure 8: Visualisation of a friction impulse.

Listing 7 shows how the rotation impulse gets calculated. The calculation utilizes pre-computed coefficients, which are stored in the collision manifold m and its actual contacts c. Calculating the friction impulse is done similarly, but utilizes the tangent of the collision normal instead of the normal itself. Additionally, the friction impulse does not involve the bounce factor and it gets clamped proportionally to the rotational impulse.

Listing 7: Collision resolution - rotation/normal impulse

```rust
v_rel = b.transform.velocity
        + cross(b.transform.angular_velocity, c.diff_to_b);
v_rel -= a.transform.velocity
         + cross(a.transform.angular_velocity, c.diff_to_a);

let v_rel_n = v_rel.dotted(m.normal);

let mut jn = c.normal_magnitude
             * v_rel_n
             * m.bounce_factor
             * m.inv_contact_count;

jn = f32::max(jn, 0.);

apply_impulses(a, b, jn * m.normal, c.diff_to_a, c.diff_to_b);
```

3.1.4 The World

The world contains all the necessary data structures to simulate our physics world. It is the main entry point into *Rustycs* and describes how its physics evaluation works at the highest level. Its primary purpose is to store bodies, attractors and forces. Next, it keeps track of collision manifolds that get generated in the collision detection phase. Finally, it holds the simulation tick rate, delta time and a fixed number of resolution iterations, the collision precision, that should be performed in the collision resolution phase. A physics step in *Rustycs* involves 4 sub-steps and its implementation can be viewed in Listing 8:

- Collision detection: Every dynamic body gets compared to all other bodies. The comparison classifies the bodies according to their shape combination, e.g., circle vs. AABB. Afterward, *Rustycs* analyzes the shapes and generates a collision manifold if the body shapes intersect.

- Applying world and attractor forces: Once the collisions have been detected, we apply the accelerations of the world force and all attractors to the body velocities.
- Collision resolution: Here we resolve all collisions by their manifold. The resolution process sequentially calculates velocity and angular velocity impulses and applies them to the body's transform state. At the end, the body locations get adjusted directly to revert their intersection.
- Moving dynamic bodies: Finally, we apply the velocities and angular velocities that were accumulated in the physics steps to the body. We have now cleanly updated the body with respect to all present forces and collisions that occurred.

Listing 8: The physics step

```rust
pub fn update(&mut self) {
    // Collision detection
    self.detect_collisions();

    // Applying world and attractor forces
    for body in self.bodies
            .iter_mut().filter(|b| b.body_type == Dynamic)
    {
        let mut f = self.forces;
        self.attractors.iter().for_each(|a| {
            f += a.get_attraction(body);
        });
        body.transform.velocity += f * self.delta_time;
    }

    // Collision resolution
    if !self.manifolds.is_empty() {
        self.setup_resolutions();
        for _ in 0..self.collision_precision {
            self.resolve_collisions();
        }
        self.correct_positions();
        self.manifolds.clear();
    }

    // Moving dynamic bodies
    for body in self.bodies
            .iter_mut().filter(|b| b.body_type == Dynamic)
    {
        body.transform.location +=
            body.transform.velocity * self.delta_time;
        // rotate according to angular velocity
        body.rotate(self.delta_time);
    }
}
```

3.2 Optimizations

Numerous techniques to optimize the time and space complexity of algorithms or applications exist. Some techniques we will discuss in this section are applicable to any algorithm. However, they require basic knowledge about how instructions get executed on the CPU or how memory management works. Techniques specifically designed to optimize physics engines are generally less flexible but can be useful in other domains.

Allocating memory happens in two main memory regions of a program: the stack and the heap. Memory allocations on the stack are fast and possible when the program knows the size of the data at compile-time [18]. In contrast, allocation on the heap is much slower. However, it is more flexible and is e.g., used when the specific size of the data structure can only be determined at run-time, or if the structures are too big to be efficiently stored in the stack. Consequently, a collection of data structures can only be stored on the stack, if it has a predefined size. This corresponds to an array in most programming languages. Since allocation on the stack is much faster, it should be used whenever it is possible and reasonable to do.

In *Rustycs* we store the vertices of the body shapes in arrays. This is possible, because we defined a maximum of 8 possible vertices per shape. While this eliminates the possibility of more complex shapes, it is certainly not a downside. Detecting collisions is a process that gets exponentially more expensive for each additional vertex, especially when we want to detect collisions between two polygon bodies. Additionally, allocating the vertices on the stack reduces memory fetching time. Therefore, we increase the computation speed greatly since the engine potentially needs to execute this process hundreds, if not thousands, of times in a single physics step.

In programming, most of the time, there is more than one way to achieve a desired outcome. However, the cost to achieve it is not necessarily always the same. During the implementation process, we noticed that physics engines use inverse values in various parts of them. The reason is that the CPU requires more cycles to calculate divisions than multiplications. In fact, divisions require up to 10 times more cycles than their counterparts [19]. So, when we need to divide values by the same constant in a lot of places, it is beneficial to calculate the inverse of the constant once and multiply the values by the inverse instead. Other examples would be the power function and taking the square root of a number. Multiplying a number by itself is equivalent to taking its power of two, but in terms of computational complexity, the power operation performs worse than a simple multiplication. Taking the square root of a number is also a highly complex procedure for the CPU.

In line 10 of Listing 5 we can see a practical example of these optimizations. There is no reason to compute the actual distance between the two circles since we can simply compare the squared distance to the squared total radius. This allows us to delay the computation of the actual distance, which involves taking a square root. Furthermore, instead of computing the square radius via a power function, we can simply multiply the radius by itself. Now the check for the actual collision is optimized and we only compute expensive characteristics when we need them.

Although the improvements above certainly result in performance gains, in our case, they are not enough by themselves. Collision detection of n bodies has a run time complexity of $O(n^2)$, which bottlenecks our engine quickly if we implement the detection process naively. The run-time complexity is especially problematic if we accurately check for intersections between each body immediately. By utilizing a physics engine optimization technique that involves a broad and a narrow phase, we can reduce the compute intensity of the collision pipeline.

The broad phase initially decides if it is possible that two bodies intersect, and marks them as a possible collision. It calculates axis-aligned bounding boxes for each body with respect to their current rotation, which we call their "hitboxes", which represent the outer bounds of the bodies. OBBs and polygons have to continuously update their bounding boxes since a rotation changes their orientation, which could influence their outer bounds. Only if the outer bounds of the bodies collide, we consider the pair to possibly collide. With this technique we filter out the majority of bodies that do not intersect with cheap AABB collision checks. In Figure 9, we illustrate examples of hitboxes intersecting and getting flagged as possible collisions. On the left side, we see that the OBB hitbox is equal to its actual shape due to its rotation, and does not intersect with the circle hitbox. However, rotating the OBB by 90 degrees results in a hitbox that intersects with the hitbox of the circle even though the OBB location did not change. For that reason, the broad phase marks the example on the right side as a possible collision.

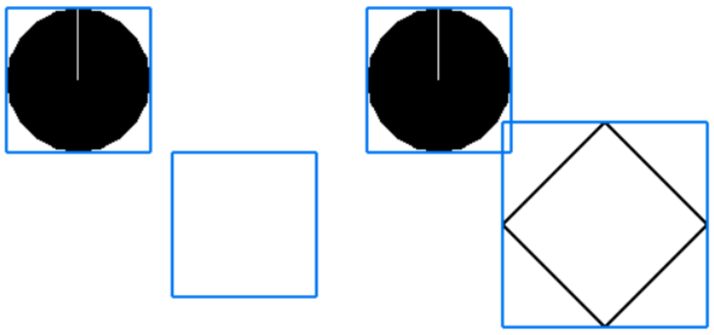


Figure 9: Comparison of hitboxes.

Generating the manifolds and checking whether the bodies actually intersect is deferred to the narrow phase. Since we filtered out the majority of cases where bodies could not possibly intersect, the number of accurate and expensive checks is reduced dramatically.

3.3 Demo Application

Developing a physics engine from scratch involves various challenges. Arguably, the most crucial aspect is verifying whether the engine works as intended. A simple way to check how the engine behaves is to visualize it. Other engines usually come with a demo showcase to tackle this problem. However, since we developed our very own engine from scratch, there are no existing solutions to visualize the inner workings of the simulation. One obvious reason for this is that other developers can not possibly predict what the world state looks like or how its contents are defined. Interestingly, this problem begins with the decision regarding the data types used. For example, there is no guarantee that existing visualisations utilize the same floating point accuracy as *Rustycs* and we potentially need to convert every single number of our world state to visualize it.

To circumvent these issues, we decided to implement a custom demo application to visualize the world's state. However, seemingly simple tasks like rendering pixels on the screen are intricate processes that utilize the graphics pipeline and require direct calls to the GPU and CPU. For this reason, we defer this task to an existing rust crate called "Macroquad" [20]. This library is very easy to use and we utilize it to render lines, circles and text to the screen.

One basic challenge of displaying world entities on the screen is the discrepancy in the internal coordinate systems. Rendering libraries commonly use screen coordinates, which diverge from a classic coordinate system. The main distinction is the location of their origins. Figure 10 illustrates the difference between classical screen coordinates and world coordinates. Screen coordinates usually have their origin in the top-left corner of the screen and are bounded by the physical screen, whereas world coordinates are centered around their origin and unbounded.

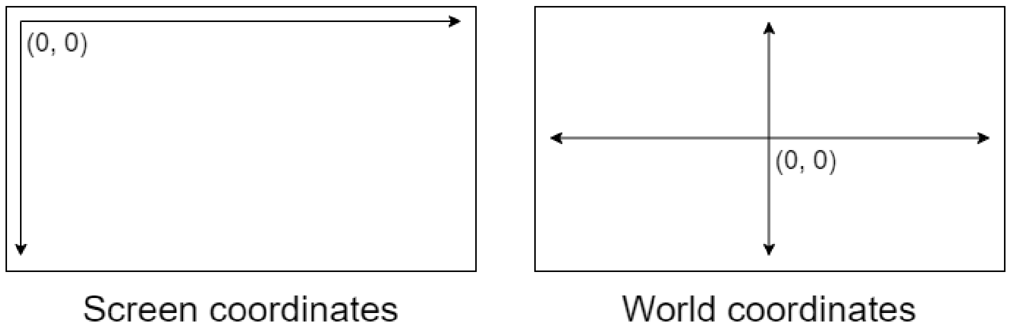


Figure 10: Comparison of screen and world coordinates.

Screen coordinates have no negative values and incrementing the vertical axis value correlates to downward movement. Furthermore, we needed to define a ratio of world distance to pixels to achieve a stable visualisation of distances. This was modelled as the "pixel to meter" ratio (PTMR) in the world struct and was also incorporated in the translation. Additionally, the ratio allows us to zoom the visualisation in or out, by adjusting it during the application run-time. In Listing 9, we show how the actual translation is achieved.

Listing 9: Translating between screen and world space

```rust
pub fn world_to_screen(
    &self, coordinate: Vector2, w: f32, h: f32
) -> (f32, f32) {
    let x = (coordinate.x * self.pixel_to_meter) + w * 0.5;
    let y = (-coordinate.y * self.pixel_to_meter) + h * 0.5;
    (x, y)
}

pub fn screen_to_world(
    &self, x: f32, y: f32, w: f32, h: f32
) -> Vector2 {
    let x = (x - w * 0.5) * self.inv_pixel_to_meter;
    let y = -(y - h * 0.5) * self.inv_pixel_to_meter;
    Vector2::from(x, y)
}
```

To translate world to screen coordinates we multiply them by the PTMR. Note that it is necessary to invert the vertical coordinate, since the screen handles them differently. Afterwards, we can simply add half of the screen height or width to offset them to the screen center. Translating the screen to world coordinates is achieved by reversing the order of the instructions. Once we are able to freely translate between the two coordinate systems, we can render the state of the world. We simply iterate over all entities we want to render, translate their world coordinates to screen coordinates, and apply any additional scaling that is necessary. In addition to visualisation, the demo application offers more advanced features that interact with the engine state. In the interest of conciseness, interacting with the simulation is done via keyboard strokes, which are detected with the macroquad crate. In Section 4, we expand on these additional features of the demo application.

4 Evaluation

This chapter evaluates *Rustycs* in regard to its functionality and coherence. In Section 4.1, we analyze *Rustycs* with the help of implemented demo scenes that test various aspects of the engine. Section 4.2 expands on the debug features of the demo application. In Section 4.3, we attempt to evaluate how coherent the implementation of *Rustycs* itself is.

4.1 Demo Scenes

We implemented various demo scenes that enable future users to quickly verify if the correctness and accuracy of *Rustycs* suffices their needs. Demo scenes efficiently verify *Rustycs'* integrity since there is no need to understand the source code of *Rustycs* to test its functionality. Due to page constraints, we will only discuss a subset of the implemented demo scenes in this thesis.

To showcase how bodies interact with each other during collisions, we designed a demo scene that spawns randomly generated bodies that can rotate. These body types are circles, OBBs, and polygons. Each body has a random material associated with it and has a random size. Figure 11 depicts the scene once the bodies have fallen into the pool and come to a standstill.

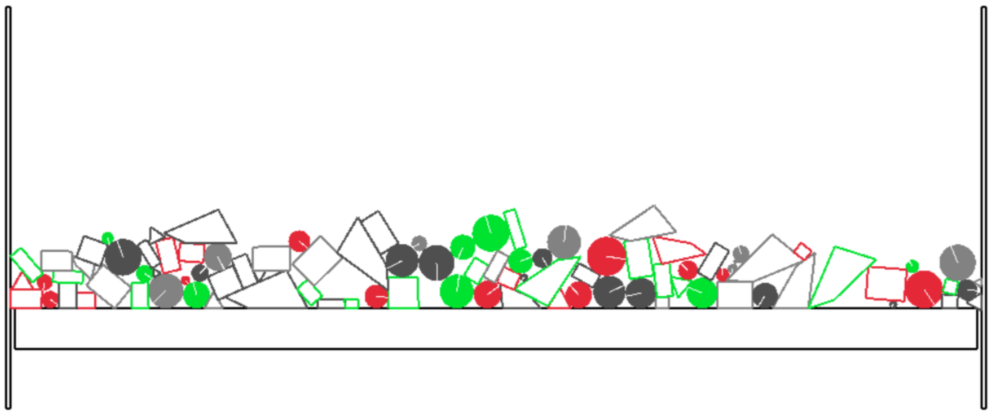


Figure 11: Demo scene: piling of rotating bodies.

Next, to showcase how body materials behave differently, we designed a demo scene that spawns five circles with different materials. They all fall from the same height and spawn next to each other, making the comparison very clear. Additionally, the user can spawn other objects to interact with the different bodies and analyze how the material friction factors affect the collisions. Figure 12 depicts the scene after the initial bounce of each circle. The difference in their bounce height is clearly visible and results from their distinct restitution averages. The circles have the following materials: rubber (red), plastic (green), stone (grey), metal (dark grey), and default (black). We can clearly see that the rubber ball bounces the highest, which makes sense since the rubber material has the highest restitution coefficient.

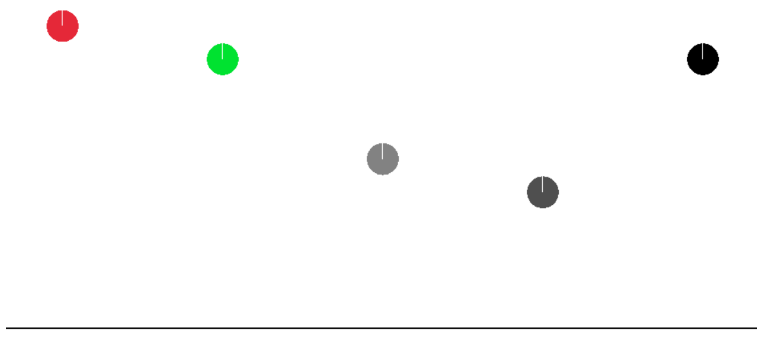


Figure 12: Demo scene: bounce difference of materials.

Lastly, to showcase the integrity of attractors, we implemented an approximated simulation of the solar system. The sun is an attractor in the middle of the demo scene, and all planets are bodies that that oscillate around the sun. Planets have manually assigned mass values to enable the approximation of their behaviour in regard to the sun's attraction force. Additionally, we associate names with the attractor and bodies to render them with custom colors instead of the material colors. Figure 13 illustrates an arbitrary state during the simulation of the solar system.

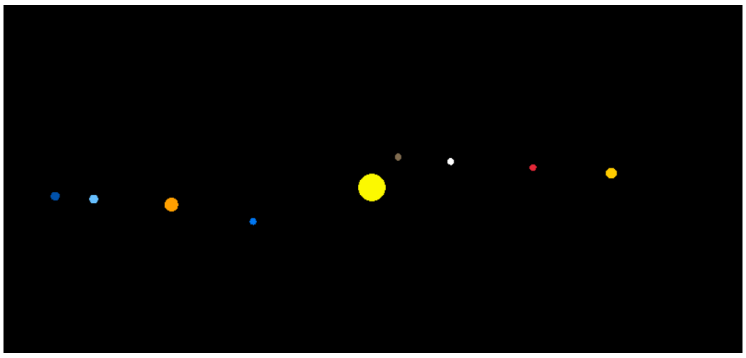
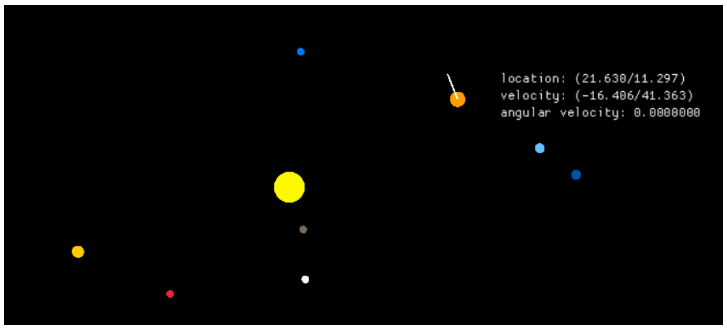


Figure 13: Demo scene: approximation of the solar system.

In Figure 14, we highlight the orbital velocity of Jupiter after some time has passed. Its velocity indicator showcases how the gravitational force of the sun prevents Jupiter from leaving its approximated orbit around the sun.



```
location: (21.630/11.297)
velocity: (-16.406/41.363)
angular velocity: 0.0000000
```

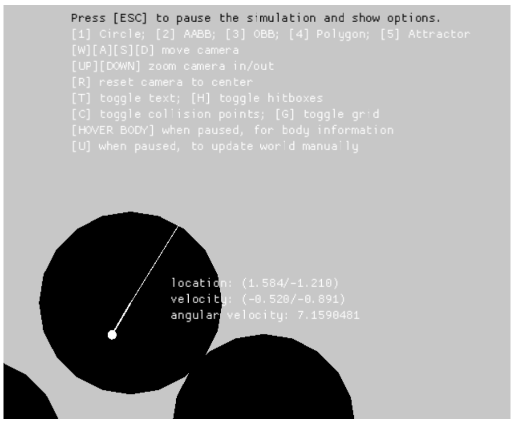
Figure 14: Demo scene: Jupiter's orbital velocity after numerous physics steps.

4.2 Manual Testing Sandbox

Verifying the correctness of *Rustycs* proved to be challenging when solely relying on a debugger or printing various states. To improve the process of debugging the engine, we implemented several features that helped us gain a detailed insight into the current state of the physics world.

Implementing these features was done by utilizing the existing functionalities of the macroquad crate. The crate enabled us to probe for specific keystrokes, so we defined a mapping for each debugging feature and implemented its functionality. For instance, it is possible to pause the engine by pressing the ESC key.

Figure 15 displays the simulation in its paused state. When paused the demo application displays all implemented debugging features and their associated keybindings.



```
Press [ESC] to pause the simulation and show options.
[1] Circle; [2] AABB; [3] OBB; [4] Polygon; [5] Attractor
[W][A][S][D] move camera
[UP][DOWN] zoom camera in/out
[R] reset camera to center
[T] toggle text; [H] toggle hitboxes
[C] toggle collision points; [G] toggle grid
[HOVER BODY] when paused, for body information
[U] when paused, to update world manually
```

```
location: (1.584/-1.210)
velocity: (-0.520/-0.891)
angular velocity: 7.1590481
```

Figure 15: Visual debugging information.

As depicted in Figure 15, the application offers various helpful features, such as stepping through the simulation one tick at a time. Being able to analyze individual physics steps and what happens between them allowed us to detect bug sources, especially those present in the collision pipeline. Displaying collision points and body hitboxes allowed us to further analyze the collision detection process, and these features are illustrated in Figures 16 and 9, respectively.

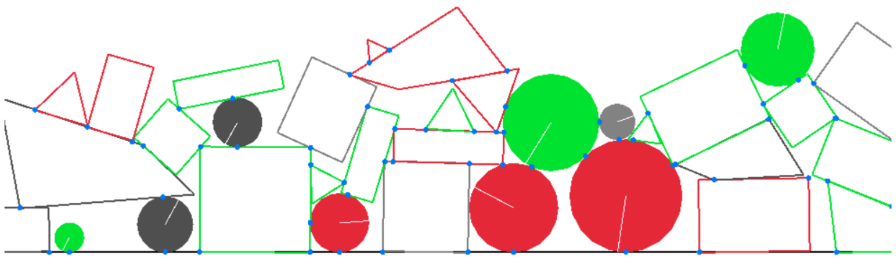


Figure 16: Visualisation of collision contact points.

Spawning bodies or attractors dynamically at the mouse location allowed us to test all kinds of interactions in a flexible manner. Finally, adjusting the location and zoom factor of the camera enabled us to get a more detailed depiction of interactions.

4.3 Coherence

One of the primary goals of this work was to create a coherent physics engine in terms of its structure and source code. We achieved coherent structure by separating the engine by responsibilities:

- The world struct: The main entry point into the engine, and when utilizing the engine in separate projects, we wanted other developers to simply work with the world in their application loop. Therefore, the world is used to initialize the simulation state in regard to its tick rate and the bodies, forces, and attractors it contains. Once a world is initialized, it is only necessary to update it sequentially in a continuous loop and visualize its state.
- Entities within the world and their properties: The engine updates the world state according to the present bodies, forces, and attractors. We structured the code base accordingly and created separate files for bodies and their properties, such as their material and transform state. Attractors and forces, on the other hand, are less complex and self contained files.
- The collision pipeline: Detecting and resolving collisions is a huge factor in terms of the complexity of the physics engine. Therefore, the whole collision pipeline and its associated data types, such as the manifolds or hitboxes, were separated into their own submodule. Since the entire pipeline is complex, we additionally utilize a concept called "do or delegate". It intentionally breaks up complex code into trivial problems and delegates them to sub-functions that handle them. For instance, classifying collisions according to the involved body shapes and delegating them to their respective detection functions uses this pattern.
- Visualisation: A key focus during implementation was to keep the demo application and the engine completely separated. We achieved this by implementing all the advanced features of the demo applications,such as predefined polygon shapes, in the demo application. Furthermore, when translating world coordinates to screen coordinates, we provide the concrete floating point values instead of Vector2 instances. This enables all future projects that want to use *Rustycs* to use custom structures and processes to visualize the world state.
- Demo application features: Additional advanced features of the demo application, like the movable camera or scene management, were also implemented outside of the engine. They are not necessary to the engine's functionality and have no place in its source code.

Another key aspect of coherence is the readability of the source code. We put effort into naming variables used within the engine and demo application to represent what they mean in their context. This was especially important when comparing our source code to the implementations of other engines, which often use abstract variable names and are comparatively less readable as a result. Rust as a language also enabled us to write readable and modern source code without sacrificing run-time performance. We utilized various zero-cost enum types to represent states in favor of simple boolean values. Combining speaking variable names and enums with other zero-cost abstractions of Rust, such as its iterators, resulted in coherent code, which enables future developers to understand the concepts used in *Rustycs* easily.

5 Lessons Learned

This chapter highlights interesting lessons we learned during the implementation of our physics engine. Section 5.1 focuses on general concepts and knowledge we gained by implementing the engine. In Section 5.2, we explain how Rust assists and sometimes almost forces developers to deepen their understanding of low-level concepts.

5.1 General

Separating large projects based on functionalities, e.g., physics bodies and their materials, is vital for their coherence and future maintainability. Section 4.3 already elaborated on how implementing *Rustycs* honed our ability to write coherent code in the context of physics engines. However, numerous concepts we utilized in this project are applicable to general programming.

By implementing a simplified two-dimensional physics engine, we gained insight into the way such engines function at a high level. Modelling locations and movement within a simulated space is done via vectors, which are easy to interpret and extend with functionality in actual code. There are various ways to implement entities that live in the simulated world, and they should be defined according to the use case of the engine. For instance, if there is no need for different types of materials, they should not be included in the body representation. Extensive planning and analysis of what the engine will be used for is crucial for preventing unnecessary development overhead. Performance quickly becomes a significant aspect when implementing physics engines. Therefore, it is of utmost importance to understand how individual parts of the engine interact with each other to optimize internal and external processes.

Utilizing the engine in our demo application surfaced issues regarding the separation of concerns. The engine should only be responsible for tasks that concern the engine to prevent external dependencies. Otherwise, there is a possibility that the engine depends on the visualisation that uses it, which leads to several issues for the usage of the engine in third party software. Furthermore, extending the demo application with additional debugging features deepened our understanding of utilizing a continuous loop to interact with and visualize the world state. Additionally, we were confronted with distinct representations of coordinate systems, how to translate between them, and why they are necessary.

5.2 Rust

On the one hand, choosing Rust initially slowed the implementation process down, in a way languages like Java or Python would not have. It was necessary to study its type system, its syntax, and how to utilize its memory management efficiently. On the other hand, Rust deepened our knowledge about how code actually works under the hood. As we got more familiar with the language, we realized its potential and noticed shortcuts or bad practices that are embedded in other languages. However, we acknowledge that Rust is not perfect and certainly not a language that should be used for every task. Its comparatively complex, explicit syntax and type system, in combination with its unique way to manage memory, provides advantages, that are not necessarily needed in many domains.

A common issue for new Rust developers is its complex string handling. However, their complex nature is not unwarranted and could be considered a feature of Rust. Strings of any kind, especially UTF-8 encoded strings, are complex data structures [21]. Typical programming languages, such as JavaScript, tend to hide the complexity of strings in their compilers and runtimes to simplify their usage. Rust deliberately exposes the real complexity of strings to developers. This trade-off prevents incorrect handling of UTF-8 characters (which require more memory than 8-bit ASCII characters) and thus prevents errors that could occur otherwise.

In the context of memory management, Rust is not only an efficient language, it highlights the intricacies and complexity of managing memory itself. As previously mentioned, Rust manages memory via the concept of Ownership & Borrowing [6]. It enables the compiler to detect most memory issues, i.e., a program will not even compile with memory management issues in the code base. Interestingly, during the implementation of *Rustycs* we encountered a total of zero memory issues once the engine successfully compiled. Since the Rust compiler rigorously checks numerous ownership rules and performs other tasks, such as inferring static types, the compile times can become quite long. However, the Rust project maintains their own a language server protocol (LSP) [22]. It is common practice to embed a LSP into the programming environment because it highlights faulty code immediately after it is written. Regarding Rust specifically, the LSP enables developers to detect errors without compiling the code base, which in turn eliminates the wait times during compilation.

Since Rust is statically typed, the compiler and therefore the LSP can leverage its type system to infer additional context information. Rust's rich and explicit type system improves the coherence of source code and even models memory management concepts. Smart pointers in Rust act like traditional pointers in languages like C or C++ but also provide additional capabilities and metadata [23]. For instance, we can wrap any variable of type T that would usually get stored on the stack with a Box<T> smart pointer, which explicitly tells Rust to store the variable on the heap instead. In addition, Rust supports type aliasing out of the box. This is important since developers often have to wrap data types with multiple smart pointers to achieve the intended functionality. Listing 10 models the traditional functionality of a binary tree node in Rust. We can see that the concrete type definition is quite verbose and complex. To ease its future usage, we can type alias it.

Listing 10: Example of type aliasing smart pointer type

```rust
struct Node {
    pub val: i32,
    pub left: Option<Box<Node>>,
    pub right: Option<Box<Node>>,
}

type TreeNode = Option<Box<Node>>;
```

6 Limitations and Future Work

In this section, we discuss the current limitations of our work and how we will address them in the future. Section 6.1 lists a few common physics engine features we neglected for this work. In Section 6.2, we elaborate various techniques that could be utilized to further improve the performance of *Rustycs*.

6.1 Missing Features

For this work, we focused on a subset of body shapes that are relatively simple to implement. Fully functional physics engines do not take such shortcuts. Concave polygons are the obvious next shape type to support within *Rustycs*. The concavity of these polygons makes special guarantees of convex polygons with regard to the separating axis theorem obsolete. Algorithmically splitting up arbitrary polygons into triangles is the common way to circumvent these issues. However, this makes the collision detection process more complex.

When engines detect collisions in a *DCD* fashion, they run into the issue of potentially missing collisions that happened in between distinct ticks. *CCD* would prevent these detection artifacts but it again adds additional complexity and demands more resources since it needs to predict whether collisions happen.

Particle systems are objects within a physics world that emit or contain numerous minute particles [24]. Common use cases are the simulation of fire or sprinkling water within the engine. Their primary feature is not to simply display the particles, but that their individual particles can interact with other entities present in a scene. Since such systems usually emit numerous particles, the main factor that prevented us from implementing them was performance. Additional optimizations would be necessary to ensure good performance when particle systems are present in *Rustycs*.

6.2 Potential Optimizations

The glaring issue of *Rustycs*' current state is that it is evaluated by only a single CPU core. Implementing multithreading to utilize multiple CPU cores would dramatically boost performance. However, correct and deterministic multithreading is a hard challenge in any programming language, and more so in Rust due to its explicit and precise syntax nature.

Caching collision manifolds across physics steps and warm starting them helps to stabilize the collision handling [25]. Instead of detecting all collisions and generating their manifolds in each physics step, caching them allows us to simply update the manifolds that were present in the previous step. If the bodies no longer collide, the manifold can be removed; otherwise, it needs to be updated accordingly. To stabilize the collision simulation properly, it is important to constraint updates and potentially only update a manifold if the change passes a certain threshold.

In the current iteration of *Rustycs* the world space is not bounded. Therefore, bodies could theoretically reach coordinates where they overshoot the maximum value of 32-bit floating point numbers. Once they reach these kinds of coordinates, their behaviour in an optimized Rust program is undefined. Optimized compilations do not check for overflows, since that would affect run-time efficiency. However, Rust debug compilations are intentionally less optimized to identify issues such as numerical overflows. So, by limiting the size of the world, we eliminate the issues that undefined behaviour introduces in such cases. More importantly, a size limit would allow us to define sectors within the world, which can be used to enhance the broad phase efficiency in the collision detection phase. Bodies that move past the world boundaries can be removed, and if we limit the allowed size of bodies, we would only need to check neighbouring sectors within the world.

7 Conclusion

Physics engines enable the simulation of realistic physical behaviour in various domains, including, but not limited to, video games and CGI. In recent years, established physics engines, such as *PhysX* by Nvidia, have accumulated large code bases and have become hard to analyze [2]. Engines tailored to educational purposes, such as *Box2D-lite* by Erin Catto, tend to be smaller, but often suffer from practices that impact the coherence of their structure or source code.

In the course of this work, a physics engine and a separate demo application to visualize and interact with the engine were implemented. The engine supports four distinct shape types, namely circles, AABBs, OBBs and polygons. Collisions between bodies are detected via DCD and the rate of detection (tick rate) is freely adjustable. The demo application makes use of the implemented engine but is separated from any engine logic.

This project was developed in the programming language Rust and utilizes external libraries to generate random numbers and render the world state to the screen. Careful consideration during the implementation of *Rustycs* and partially the usage of Rust enabled the source code and project structure to stay coherent over the development life cycle. The final iteration of this work consists of 4055 lines of code spread across 29 files. The implementations of the engine[1] and demo application[2] can be viewed in their respective GitHub repositories.

While the functionality of *Rustycs* is rather simple, the project still serves as a great starting point for developers interested in venturing into the world of physics engines or Rust alike.

[1]https://github.com/divtor/rustycs
[2]https://github.com/divtor/rustycs-macroquad-demo

8 Literature

List of Figures

Listings

References

[1] E. Catto, “Box2d.” https://github.com/erincatto/box2d, October 2020. (visited on 06/02/2024).

[2] Nvidia, “Physx.” https://github.com/NVIDIA-Omniverse/PhysX, January 2024. (visited on 06/02/2024).

[3] R. Gaul, “Collision detection in 2d or 3d – some steps for success.” https://randygaul.github.io/collision-detection/2019/06/19/Collision-Detection-in-2D-Some-Steps-for-Success.html, June 2019. (visited on 06/02/2024).

[4] T. R. Foundation, “The rust programming language.” https://www.rust-lang.org, August 2023. (visited on 06/02/2024).

[5] A. B. et al, “System programming in rust: Beyond safety,” *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pp. 156–161, 2017.

[6] T. R. Foundation, “Understanding ownership.” https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html, August 2023. (visited on 06/02/2024).

[7] Wikipedia, “Rigid body dynamics.” https://en.wikipedia.org/wiki/Rigid_body_dynamics, February 2024. (visited on 19/02/2024).

[8] Wikipedia, “Soft-body dynamics.” https://en.wikipedia.org/wiki/Soft-body_dynamics, February 2024. (visited on 19/02/2024).

[9] Wikipedia, “Fluid animation.” https://en.wikipedia.org/wiki/Fluid_animation, February 2024. (visited on 19/02/2024).

[10] Unity, “Continuous collision detection (ccd).” https://docs.unity3d.com/Manual/ContinuousCollisionDetection.html, February 2024. (visited on 20/02/2024).

[11] Wikipedia, “Convex polygon.” https://en.wikipedia.org/wiki/Convex_polygon, February 2024. (visited on 06/03/2024).

[12] D. Wiki, “Seperating axis theorem (sat).” https://wiki.delphigl.com/index.php/Tutorial_Separating_Axis_Theorem, September 2013. (visited on 05/03/2024).

[13] LLVM, “Llvm.” `https://llvm.org`, November 2023. (visited on 20/02/2024).

[14] T. R. Foundation, “The crates ecosystem.” `https://crates.io/crates?sort=downloads`, Februrary 2024. (visited on 20/02/2024).

[15] Wikipedia, “Gilbert-johnson-keerthi (gjk) algorithm.” `https://en.wikipedia.org/wiki/GilbertâĂŞJohnsonâĂŞKeerthi_distance_algorithm`, February 2024. (visited on 28/02/2024).

[16] E. Catto, “Box2d lite.” `https://github.com/erincatto/box2d-lite`, January 2019. (visited on 02/03/2024).

[17] E. Catto, “Sequential impulses.” `https://box2d.org/files/ErinCatto_SequentialImpulses_GDC2006.pdf`, January 2019. (visited on 02/03/2024).

[18] Wikipedia, “Stack-based memory allocation.” `https://en.wikipedia.org/wiki/Stack-based_memory_allocation`, January 2024. (visited on 06/03/2024).

[19] Wikipedia, “Computational complexity of mathematical operations.” `https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations`, February 2024. (visited on 27/02/2024).

[20] F. Logachev, “Macroquad.” `https://macroquad.rs`, Februrary 2024. (visited on 21/02/2024).

[21] T. R. Foundation, “Storing utf-8 encoded text with strings.” `https://doc.rust-lang.org/beta/book/ch08-02-strings.html`, February 2023. (visited on 04/03/2024).

[22] Wikipedia, “Language server protocol (lsp).” `https://en.wikipedia.org/wiki/Language_Server_Protocol`, September 2023. (visited on 04/03/2024).

[23] T. R. Foundation, “Smart pointers.” `https://doc.rust-lang.org/book/ch15-00-smart-pointers.html`, February 2023. (visited on 04/03/2024).

[24] D. Shiffman, “Particle systems.” `https://natureofcode.com/particles/`, February 2024. (visited on 03/03/2024).

[25] A. Chou, “Collision warm starting.” `https://allenchou.net/2014/01/game-physics-stability-warm-starting/`, January 2014. (visited on 03/03/2024).