HAMBURG UNIVERSITY OF APPLIED SCIENCES

BACHELOR THESIS

In order to attain the academic degree B. Sc.

Physics simulation in Mobile Applications:

Creation of a two – dimensional rigid body Physics Engine for cross platform App Development

Submitted by:

Simón Hoyos Cadavid – [redacted].

Hamburg – 26.02.2020

Thesis Advisor: Prof. Dr. Jan Andries Neuhöfer (HAW Hamburg)

Second examiner: Jörg Pechau (ICNH GmbH)

Faculty of Design, Media and Information

Department Medientechnik

Media Systems

HAMBURG UNIVERSITY OF APPLIED SCIENCES

Hochschule für Angewandte Wissenschaften Hamburg

HAW HAMBURG

BACHELOR THESIS

In order to attain the academic degree B. Sc.

Physics simulation in Mobile Applications:

Creation of a two – dimensional rigid body Physics Engine for cross platform App Development

Submitted by:

Simón Hoyos Cadavid - [redacted]

ABSTRACT

Physical simulation in mobile applications is, for the most part, non-existent; with the exception of games. This might be influenced by the fact that applications are normally not expected to react in natural ways, even when they try to create an intuitive user interaction. Because of this, this thesis tries to break the stigmas of physical simulation on mobile applications, by creating an API for the creation of physically accurate motion in cross-platform app development. This was followed by the creation of a subsequent example app, as both experimentation, and argument reinforcement.

The results of creating an application, that used the developed physics API showed that such a tool is indeed helpful for the development of apps in possible real use case scenarios, as it shortens the amount of code, time and knowledge required for its creation. Nevertheless, the engine built is still opened for expansion; but it does represent a proof of concept that a broader API is needed for this branch of development.

ABSTRACT

Physikalische Simulationen in mobilen Anwendungen sind größtenteils nicht vorhanden, mit Ausnahme von Spielen. Dies könnte durch die Tatsache beeinflusst werden, dass von Anwendungen normalerweise nicht erwartet wird, dass sie auf natürliche Weise reagieren, selbst wenn sie versuchen, eine intuitive Benutzerinteraktion zu erzeugen. Aus diesem Grund versucht diese Arbeit, die Stigmata der physischen Simulation bei mobilen Anwendungen durchzubrechen, indem sie eine API für die Erzeugung von physikalisch akkurater Bewegung bei der Entwicklung von plattformübergreifenden mobile Anwendungen schafft. Darauf folgte die Erstellung einer anschließenden Beispielanwendung, die sowohl als Experiment als auch als Argumentationshilfe dient.

Die Ergebnisse dieser Erstellung der Anwendung, bei der die entwickelte Physik-API verwendet wurde, zeigten, dass ein solches Tool tatsächlich hilfreich für die Entwicklung von Anwendungen in möglichen realen Anwendungsszenarien ist, da es die Menge an Code, Zeit und Wissen verkürzt. Dennoch ist die gebaute Engine noch immer offen für Erweiterungen, aber es stellt einen Beweis für das Konzept dar, dass eine breitere API für diesen Entwicklungszweig benötigt wird.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

Abbreviation	*Definition*
AABB	Axis Aligned Bounding Box
APA	American Psychological Association
API	Application Programming Interface
GJK	Gilbert, Johnson, Keerthi algorithm
OO	Object Oriented
SAT	Separating Axis Theorem
SDK	Software Development Kit
UI	User Interface
UML	Unified Modeling Language

LIST OF CODE FRAGMENTS

LIST OF EQUATIONS

LIST OF FIGURES

All figures with no clear references in this thesis were self-made utilizing either GeoGebra, for geometrical explanations; or StarUML for architecture clarifications and UML diagrams.

1. INTRODUCTION

The laws of the universe are already written; the human being only tries to uncover them. This is a fact that no matter what anyone says, cannot be discarded. It does not matter how hard one tries; it is impossible to change them… At least in the real world. One aspect that can be grasped and influenced are the rules of *digitally* created worlds, which in turn brings with it the fact that this digital worlds also interact with real people.

But then, the question of how these types of worlds are connected to each other arises. Well, one idea to unify them would be trying to make the digital world behave in a similar way to the real world. This is where the main statement for this thesis is seeded, being that digital worlds might not need to always follow the same laws as reality; but it might be useful to, at the very least, have the option to make a digital world behave as the real one.

Many fields integrate these laws of reality, which are of course called physics. Games, for example, are one of the biggest users of physics simulation, when accounting to their creation. However, other disciplines could also benefit by the utilization of physics. An example of one such subject might be mobile app development.

Following this logic, an valuable aspect that was recognized by the writer was the lack of a sturdy physics simulation API in the branch app development. Since the writer of this thesis was already acquainted with this section of development, by utilizing the *Flutter* SDK, it was decided to develop an API, that would help with the physical simulation.

Not only this, but even with existing physics engines in many other platforms, *Flutter* as an SDK does not really integrate two-dimensional physics simulation at all. So, it would be an interesting idea to create an engine in order to prove that such a physical simulation might be needed.

In order to achieve the creation of the engine, the first steps in this thesis will be to introduce all the core aspects, of both physics and mathematics, which were needed for the development. This will then be followed by the exact proposition, while short after explaining how the development process took place. At the end, an analysis will be made in order to conclude if the created engine was indeed useful, and needed for the previously mentioned SDK.

Before moving forward to explain the goals of the thesis, an important clarification is going be made: All citations and references follow the APA Sixth edition format.

1.1. Goals

It is of extreme importance to declare what the main focus the investigation will be. Therefore, this chapter will try and explain the core principles which were taken into account in order to start the investigation, as well as clarifying the goals and the direction that this written document will take.

1.1.1. Formulation of the Problem

The problem which will be addressed in this thesis is the absence of a complete physics simulation engine in the *Flutter* SDK. Because of this SDK being fairly new, only about four years old to the writing of this document, it is of no surprise that developing environment barely has any kind of physics integration. It must also be noted that this SDK was created with the intent of app development, which few people would associate with physics. This does not mean it might not be needed, nonetheless.

How would it be possible to implement the SDK in order to create, for example, video games if there is no physical simulation? The industry nowadays has an almost mandatory focus on physical simulation in many their products, which could be explained with the necessity of virtual games to feel more tangible, more natural and closer to our world. Not only games, but other application could be developed, that might need to implement physics; like learning apps for school students.

Of course, the answers are not as simple, but the idea behind this project is to break this barrier of the unknown, while at least trying to implement the very core principles of physics; experimenting with what is already being utilized in other industries.

1.1.2. General and specific Goals

General Goal

The general goal of the investigation is the demonstration of the value of a 2D, rigid body focused, physical simulation engine for the *Flutter* SDK, through the development of such an engine and subsequent implementation in a mobile application.

Specific Goals

The specific goals are as follows:

1. Investigation of the core principles of rigid body kinematics and dynamics.
2. Exploration of the motion of 2D rigid bodies in a digital space and the physical interactions between each other.
3. Study of the collision detection and response between 2D rigid bodies in a digital space.
4. Implementation of 2D physics simulation into the *Flutter* SDK through the development of a physics simulation API.
5. Application of the studied concepts in the implemented engine.
6. Justification and exploration of the developed engine's architecture, methods and functionality.
7. Integration of the developed API in a cross-platform application.
8. Analysis of the impact of the physics simulation API during the development of the application (Developer view).

1.2. Hypothesis

Noticing that during this thesis the development will be implemented in the *Flutter* SDK, the hypothesis for the project is as follows:

A 2D physics engine would be of significant value for the *Flutter* SDK because of the constraints of creating natural and intuitive interactions without the usage of physics. Meaning that, as of now, every time one would want to integrate any kind of physical interaction to resemble the real world, it will have to be conceived manually. The problem arising with this is the lack of experience with physics that many developers have, since not every programmer is a physicist. The extra work, which has to be invested in order to create a simple and intuitive interaction is gigantic and not worth it. Because of this, for at least the use cases that handle physics, the need of an engine is quintessential.

2. STATE OF RESEARCH AND THEORETICAL FRAMEWORK

Before one can even start the development of any project, investigation, etc. one needs to have all of the bases clear, meaning one needs to gather the knowledge in order to tackle the uncanny path of an investigation. When the understanding of the direction of the project is set, the next step to take is of course to learn what path is going to be taken to help grow the seed of one's work. It should come to no surprise to the reader that the second chapter of this thesis is therefore the theoretical bases for the understanding of the study; since it must not only be clear for the researcher how everything works, but it is his job to explain to whomever is reading how he got to the conclusions at hand.

This chapter will be divided in several lesser, more manageable pieces of information, in order to avoid overloading the reader with data. Because of this, it will be important to take note that this chapter will include two general parts being *Physics and Simulation* and *The Flutter Framework*. These two will try and explain the general concepts behind the two-dimensional rigid body mechanics needed for the project and the functionality of *Flutter* respectively.

2.1. Part I: Physics and Simulation

"[...] we do not think that we know a thing until we are acquainted with its primary causes or first principles, and have carried our analysis as far as its elements."

– Aristotle, trans. 1991.

Before starting, it must be made clear what the two main topics of this chapter are. It is easy to state how things work, but it might not be as easy to back it up. While a madman can state that the world functions in a way that defies all expectations, sometimes this madman will be proven to be right, redefining the how we understand the inner workings of our universe. Physics is just that at its finest, a science that defines and redefines the laws of the world in which we all live. Better explained by the Merriam-Webster Dictionary as "a science that deals with matter and energy and their interactions" (Merriam-Webster Dictionary, n.d.), physics declare the interactions between physical objects, explaining how they react to each other in the real world.

It has, though, been made clear that the focus of this thesis is to emulate the interactions of objects in the physical world; to copy them as closely and precisely as possible in order to create a digital world which includes our own set of predefined laws. Making it of utter importance to explain how this laws work, which will of course be the focus of the following chapters.

Having described the first facet of this part, the second general aspect, which is the term simulation, should be defined. Simulation is referred by the Merriam-Webster dictionary (n.d) as the imitation of one process in order to make another one function. This is exactly what the project aims to do with the created API: imitate the real world; making it so physics simulation is the most important issue that the thesis will have to address.

2.1.1. How the universe moves: The Newtonian Laws of Motion

With the establishing of the direction of the thesis, it is possible to now get into one of the topics which will be the core of the investigation: The Three Laws of Motion. These laws helps one explain how many things in the universe move, from tiny pebbles to cosmic objects. These *laws* are proven concepts which have existed for centuries and are also the three basic concepts which will be used in order to build the physics simulation API.

These laws of Motion were first declared by Sir Isaac Newton in his publication *Mathematical principles of Natural Philosophy* and are as follows[1]:

> "1. A body continues in its state of rest or of uniform speed in a straight line unless it's compelled to change that state by forces acting on it (Law of Inertia)
>
> 2. The acceleration of an object is directly proportional to the net force acting on it and is inversely proportional to the mass. The direction of acceleration is in the direction of the applied net force ($f = ma$)
>
> 3. Whenever one object exerts a force on a second object, the second object exerts an equal and opposite force on the first object" (Kenwright, 2012, p. 72).

In easier to understand terms, these laws will be as follows:

First law (Inertia): Inertia is the ability for objects to stay in the predestined state, unless an external force is applied on them. This means that objects that are not moving, will not move on their own and objects that are already in motion will never stop without something stopping them. This law is uncommon to see on Earth, unless it is in a vacuum, since there are external forces being applied on everything all the time. If a ball is just placed somewhere it will try to move downwards because of gravity, and if it does not it is because another object is creating an equal but opposite force, equalizing it.

Second Law: This law describes how the force, acceleration and mass depend on one another. The typical example of this law is letting two fruits fall to the ground, an apple and a watermelon. Both will hit the floor at the same time, because the acceleration is the same (gravitational acceleration). Nonetheless there has to be something different going on between both objects. Since the mass of the watermelon is greater, the force with which the watermelon hits the floor is higher than that of the apple.

[1] The laws were first declared by Newton but will cited from other sources.

Third law: The most commonly heard phrase that describes this law is usually that every action has an equal but opposite reaction. A great example of this can be as basic as standing up. When one stands up, one is exerting a force on the floor, which in turn has to be greater than the gravitational pull, at least for the moment that one desires to stand up. When one is done and encounters itself not moving upwards, flying into the sky, is because the body has achieved a neutralization of the forces: the body is exerting the same force as gravity, but in an opposite direction, coming to a resting state.

2.1.2. Rigid Body Dynamics

"A rigid body is characterized by the region that its mass lives in."

– David H. Eberly, 2010.

Basically, everything that moves is bound to follow the three aforementioned laws in some way or another, but the focus of this project will be, for the sake of simplicity, rigid bodies. It might be of a certain importance to clarify, what exactly rigid bodies are, before explaining how they work and what are there going to be used for.

Explained lightly, rigid bodies are bodies of mass that do not deform, or at least that behave as such (Kenwright, 2012. p. 72). A single point, or particle, defines how the whole object acts. This makes it so all the infinite amount of points that surround the particle, while at the same time belonging to the addressed object, react to the changes made during this point's motion; this is called a *continuum of mass* (Eberly. 2010. p. 14.), and will be used as synonym for rigid body in this written text. As a clarification, other types of rigid bodies exist, like single particles or particle systems, but the term rigid body will be referred as afore defined during the length of this thesis.

But, no actual real object is completely oblivious to changes when being subjected to forces. Nonetheless, even if perfect examples do not exist in real life, objects with neglectable changes in their structure when applied forces can be also treated as rigid bodies; hence making it so many, if not all, solid objects can fit in this category.

Because of the already mentioned qualities of rigid bodies, these can be observed in physics simulation as a single point that contains information about mass, position and rotation (in the form of scalar values for the first and vectors for the rest) while the object itself, represented either by a mesh or a sprite[2], reacts to this point of information, mimicking its properties.

As seen bellow in figure 1, the point *D* changes both location and rotation, while the vector *u* demonstrates that the object surrounding the point also changes these same parameters accordingly. Because of this it can be inferred that the distance to the center from anywhere in the object always stays the same.

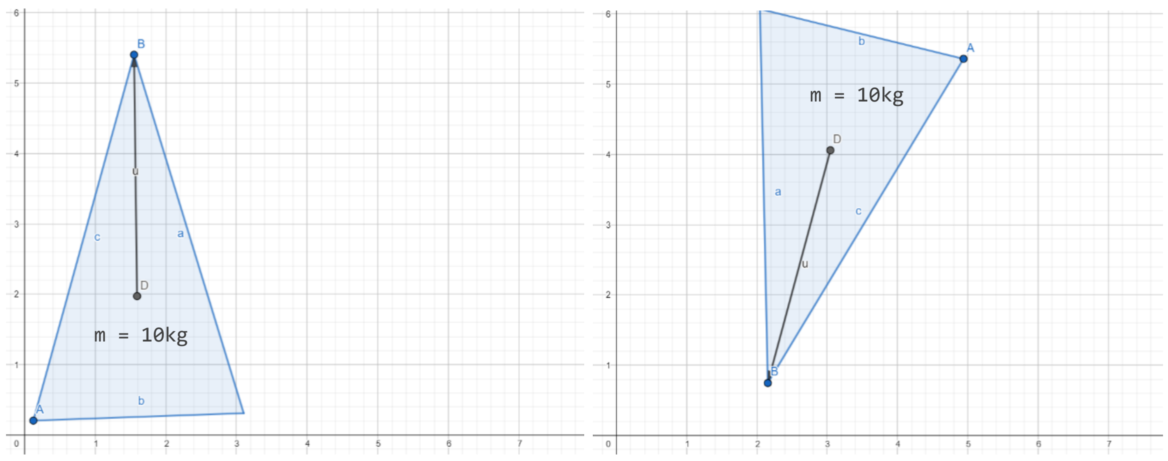


Figure 1: Demonstration of two-dimensional rigid bodies.

(Left) Initial state of the rigid body – (Right) Same rigid body after a translation and rotation

Rigid bodies are considered by the author to be the simplest, most basic forms to simulate in a two – dimensional space. Therefore, they will be used for the simulation API created during the development of this thesis. This is of course because this thesis will try to demonstrate that a physics engine is needed in the *Flutter* API, constraining itself to rigid bodies; avoiding concepts like soft bodies, fluids, etc.

2.1.2.1. Kinematics: Position, Velocity and Acceleration

Coming into topic with exactly how rigid Bodies behave in a given system, the first thematic to be discussed is that of Kinematics. “The study of motion of objects without considering the

[2] Mesh: Three-dimensional digital object.
Sprite: Two-dimensional digital object.

influence of external forces […]" (Eberly. 2010, p. 15) answers the question for the definition of the previous term. Now, how this is helpful, can only be explained with referring back to the definition of a rigid body being represented by a single point in a system, with kinematics, the first steps towards calculating how the particle should move are examined.

Position

First of all, there should be some kind of information of where the object, that is being tracked is at every point in time, or when talking about a computer program, every frame. This position should be checked, then modified for every single object on the *playing field*, or coordinate system to be exact, and since all objects that are going to be analyze during this thesis are mass continuums, one can infer that the only needed thing that needs to be kept track of is the center of each object… at least for what the position of the whole is concerned.

For the purposes of this thesis, no mathematical principles will be deeply explained. So, it is indeed recommended to have the basis of linear algebra and vector math under control. Another helpful aspect is basic understanding of OO programming, which will be touched upon in future chapters.

Now well, it is important to first define lineal displacement. As explained by David H. Eberly (2010) in his book *Game Physics,* the position is represented in cartesian coordinates as follows:

$$\boldsymbol{r}(t) = x(t)\boldsymbol{I} + y(t)\boldsymbol{J} \qquad (1)$$

Where $\boldsymbol{I} = (1, 0)$ and $\boldsymbol{J} = (0, 1)$. While $\boldsymbol{r}(t)$ represents the position vector (p. 15.). Both the x and y coordinates are equations which depend on the time. This equations will be further in the following chapter.

Velocity

Without getting into further detail, the velocity is the sum of the derivatives for $x(t)$ and $y(t)$, multiplied by the same vectors $\boldsymbol{I}$ and $\boldsymbol{J}$ (Eberly, 2010, p. 16):

$$\boldsymbol{v}(t) = \dot{\boldsymbol{r}} = \dot{x}(t)\boldsymbol{I} + \dot{y}(t)\boldsymbol{J} \qquad (2)$$

Acceleration

Following the previously stated logic acceleration is the second differentiation of the position vector (Eberly, 2010, p. 16):

$$\boldsymbol{a}(\boldsymbol{t}) = \dot{\boldsymbol{v}} = \ddot{\boldsymbol{r}} = \ddot{x}(t)\boldsymbol{I} + \ddot{y}(t)\boldsymbol{J} \qquad (3)$$

2.1.2.2. Force

One is now left with a conjunction of really similar equations that are just differentiations of one another and one does not even have the basic *x(t)* and *y(t)* to start any calculations. It is here, nonetheless, where newtons laws come to fruition, making sense of it all. It is now needed to know how to calculate the acceleration, so one can start pushing upwards in the tree of integration.

It will be now indicated to the reader to remember the second law of motion[3], acceleration is directly proportional to the force imposed over the object, and inversely proportional to its mass. The used force is a vector and it is calculated by adding all the force vectors being applied on the object. From this, it is clear here that one has a way to calculate the acceleration:

$$\boldsymbol{a} = \boldsymbol{F}\,\frac{1}{m} \qquad (4)$$

This equation will of course return a vector, but before one can start integrating to get to the position vector, it is needed to know the famous $\ddot{x}$ and $\ddot{y}$, but knowing a little mathematics it is

[3] Refer back to Chapter 2.1.1.

easy to infer that this are just the magnitude of the vector for both the *X* and *Y* axis of the acceleration vector. Because of this, one can already start integrating in order to get the other equations. As a small note, only the process for the *X* axis will be covered, but the process is exactly the same for *Y*.

The acceleration in a direction in a moment in time (t) can be taken as a constant for the purposes of this thesis, because the forces have to be all added in a frame when calculating on a computer program; therefore, returning a constant force for that frame. Consequently, the following:

$$\ddot{x}(t) = a_x(t) \tag{5}$$

$$\dot{x}(t) = v_x(t) = \int a_x \, dt \tag{6}$$

$$v_x(t) = a_x t + c \tag{7}$$

$$v_x(t) = a_x t + v_x(t-1) \tag{8}$$

As stated by Kenwright (2012) in its chapter about the Euler Method[4]: The constant generated while integrating is of course the velocity of the previous moment of calculation (also sometimes described as the initial velocity), while *t* is for the purposes of the integration the difference in time (Named normally Δt).

With the same logic one can integrate v_x to calculate the magnitude of the position vector in the *X* axis:

$$x(t) = \int v_x \, dt = \int (a_x t + v_x(t-1)) \, dt \tag{9}$$

$$x(t) = \frac{a_x t^2}{2} + v_x(t-1)t + c \tag{10}$$

$$x(t) = \frac{a_x t^2}{2} + v_x(t-1)t + r_x(t-1) \tag{11}$$

This process is then repeated for the *Y* axis.

[4] This method will be further clarified in future chapters.

Having done the needed integrations for both the X and Y components of the position vector, and incorporating them in equation (I), one is left with its cartesian representation as follows:

$$\boldsymbol{r}(t) = \left(\frac{a_x t^2}{2} + v_x(t-1)t + r_x(t-1)\right)\boldsymbol{I} + \left(\frac{a_y t^2}{2} + v_y(t-1)t + r_y(t-1)\right)\boldsymbol{J} \quad (12)$$

This is what will allow anyone to calculate where a particle is at a certain point in time, and with it, the representation of where the whole rigid body is.

2.1.2.3. Integration for computers: Approximation

With the principles of lineal position being covered, there is a thematic that has to be made emphatic before continuing the exploration of the physical principles acting on a rigid body on any given time. This being that computers are *dumb*. These machines are known for being able to do an extreme amount of calculations per second, but have to be in turn really simple. Computers are known to not be able to do calculus, because of its infinitesimal nature.

So, the question of how a program can compute the aforementioned differentiations still stands. This is where approximation methods come in. There are many numerical ways to approximate these integrations, some more precise, some faster to calculate. This chapter will try and explore some of the possible methods that will help in the creation of a sturdy physics engine. However, many methods exist and not every single one of them will be explained.

As a clarification, all of the integration methods that will be touched upon follow the Taylor Theorem, or at the very least derive from it. It is described as follows:

"If $x(t)$ and its derivatives $x^{(k)}(t)$ for $1 \leq k \leq n$ are continuous on the closed interval $[t_0,\ t_1]$ and $x^{(n)}(t)$ is differentiable on the open interval $(t_0,\ t_1)$, there exists $\bar{t} \epsilon [t_0,\ t_1]$ such that […]" (Eberly, 2010. pp. 765 - 766)

$$x(t_1) = \sum_{k=0}^{n} \frac{x^{(k)}(t_0)}{k!} \left(t_1 - t_{0)}\right)^k + \frac{x^{(n+1)}(\bar{t})}{(n+1)!} (t_1 - t_{0)})^{n+1} \tag{13}$$

Euler method

It is now encouraged to remember what was mentioned in the chapter of *Force*. The Euler method is exactly the approximation method that was used beforehand in order to calculate the position vector[5]. Utilizing the approximation with the second grade Taylor polynomial one can calculate the position, from the acceleration. Moreover, it was stated before that the acceleration will be treated as a constant for each frame of the calculation, making it easy to calculate with this.

However, the single grade Taylor approximation is also sometimes used in order to calculate the position. Following the theorem, and according to Kenwright (2012) in the book *Game Physics*, it will look like this (p. 76):

$$r_{n+1} = r_n + v_n \Delta t \tag{14}$$

$$v_{n+1} = v_n + a\Delta t \tag{15}$$

This is what is defined as the *Explicit Euler Method*, which approximates the velocity and position of the following frame, using the actual frame data.

Semi – Implicit Euler

Moreover, another method exists that, uses half the calculation in an explicit method. With this being, the calculation of the velocity, while calculating the position in the next moment in time. However, this procedure is described to not be theoretically correct, but produces an error loop that corrects itself (Kenwright, 2012, p. 77). It must be clarified that even if it is not correct, it

[5] Chapter 2.1.3.2.

is widely used in many game physics engines to calculate the position because very small time steps can be taken (60 fps for example) and the margin of error with this method is smaller, the smaller Δt is. Nonetheless, the error order is still in the *O(t)*, making it less accurate than other methods (Pronost, n.d).

As stated again by Kenwright (2012), the velocity is calculated the same way as in the explicit Euler, but it is, however, done first. The position can then be achieved with the calculated velocity, which now represents of course the velocity of the next frame, hence the name Semi – Implicit (p. 77):

$$v_{n+1} = v_n + a\Delta t \qquad (16)$$

$$r_{n+1} = r_n + v_{n+1}\Delta t \qquad (17)$$

In contrast the Explicit Euler method utilizes the velocity of the actual frame.

Other integration methods

The existence of many other useful approximation methods must be made clear, but are, for the purposes of this thesis, unimportant. The method that will be used for the engine is the Semi – Implicit Euler because of its simplicity.

Other famous methods also include Verlet Integration, which adds the previous Taylor expansion with the newly generated one; or Runge – Kutta, which uses a four order Taylor approximation (Pronost, n.d.).

2.1.2.4. Angular Kinematics and Torque

Apart to everything mentioned before, it might be obvious to state that an object does not just change positions in real life, it also rotates. This rotation is what is studied in angular kinematics and dynamics. Luckily, angular kinematics work in a very similar way to lineal kinematics, they almost mirror one another. Consequently, the three main aspects of circular movement, *angular displacement*, *velocity and acceleration* are also differentiations of one another; while the *torque*, works similarly to force. This will be further explained in this chapter.

Angular displacement, velocity and acceleration

Angular displacement is defined as "[…] the angle through which an object moved about a specified center and in a specified sense" (Ramtal; Dobre, 2014, p. 210), with the center being the reference point of the rigid body.

Using the same logic as before, the differentiation of the displacement depending on time produces the angular velocity, while the derivative of the latter allows the calculation of the angular velocity (Ramtal; Dobre. 2014. pp. 210 - 211):

$$\omega = \frac{d\theta}{dt} \qquad (18)$$

$$\alpha = \frac{d\omega}{dt} \qquad (19)$$

Where $\theta(t)$ is the angle in Radians in which the object rotated in the last interval, $\omega(t)$ the angular velocity and $\alpha(t)$ the angular acceleration. Moreover, knowing the differentiations, and taking into account that time cannot be calculated infinitesimally with computers, the two following formulas can be inferred:

$$\Delta\theta = \omega\Delta t \qquad (20)$$

$$\Delta\omega = \alpha\Delta t \qquad (21)$$

Torque and moment of inertia

At this moment is when the rotation gets a little more complicated, since the way an object rotates depends on the shape of the object; in direct contrast to lineal displacement where an object can move only by having a force applied to it. However, something yet to be explained, named the moment of inertia, can help with these calculations. It is defined as the sum of all the masses of the particles in a body multiplied by the square of the distance of each particle to the center of a rigid body (Serway; Jewett. 2014. p. 303):

$$I = \sum m_i \, r_i^2 \qquad (22)$$

Or in its infinitesimal representation, for objects like rigid bodies:

$$I = \int m_i r_i^2 \; dm \qquad (23)$$

Nevertheless, this leaves behind a new question: how this moment of inertia influences the angular acceleration. But, thinking back at Newton's second law, some parallels can be brought to light, being that the moment of inertia affects the angular acceleration the same way mass affects lineal acceleration, and that generally the three laws apply not only for lineal but for angular motion as well (Ramtal; Dobre. 2014 p. 339). Furthermore, apart from the relationship, Ramtal and Dobre (2014) also explain that the rotational force applied to an object, is named *torque*. This leaves one with the equality between angular acceleration, torque and moment of inertia as:

$$\boldsymbol{\alpha} = \boldsymbol{T} \, \frac{1}{I} \qquad (24)$$

With ***I*** being the moment of inertia and ***T*** the torque vector.

But it is still needed to be clarified what exactly the torque is and, continuing with the justification, and as explained above, torque is just the force that is applied to the body in order for it to rotate. However, the axis of this force does not pass through the center of mass, because a force passing through it will only create a translation, not a rotation. Therefore, torque is mathematically defined as the cross product between the vector from the center of the rotation (**r**) to the point of application and the force (**F**) being applied (Ramtal; Dobre. 2014, p. 338.):

$$\boldsymbol{T} = \boldsymbol{r} \times \boldsymbol{F} \qquad (25)$$

And with this a moment has come where everything fits into place. It is now just needed to know the forces being applied to each object for the calculation of position and rotation. Yes, it is true that now what is missing is to add all forces up and calculate everything else, but a moment should be taken to appreciate how amazing it is that everything now is reduced to a same level: *Force*. If there is a force, there is either a lineal displacement or a rotation, hence there has to exist a force in order for the objects in the created system to move. These forces will be given to the objects in the engine either on their creation or added to the system after and will allow them to interact with every other object in the canvas to create an actual moving world. Therefore, once the cogs in the system start moving, because of both the first and third law of motion, they shall never stop, unless new forces are created to stop it.

2.1.3. Collisions

Something is, though, still absent to achieve the purpose of this thesis, being that even if the rigid bodies in the digital canvas can move and rotate, they still cannot interact with each other, at least with the knowledge just described. This last statement opens an enormous query that shall not long stay in the air of how objects represented by vertexes, with characteristics like position and rotation, know that there is another object to react to.

Collisions in digital spaces are calculated mostly in three equally important steps: Detection, manifold generation and resolution (in that order). Each and every single one of this steps is crucial for an accurate collision response of an object and function in a matter that passes information from one of the steps into the other, like a chain. Summarizing the three steps can be explained as follows:

- ***Detection***: Recognizes if the object is intersecting with another object, triggering a collision response.
- ***Manifold generation***: Generates the contact points of the collision between two objects, indicating where the collision took place.
- ***Resolution***: Changes the objects properties like velocity or acceleration, in order to describe the newly generated movement, according to the information created by the manifold.

This three milestones of a collision response will be explored with higher detail on the upcoming three chapters, also it will be clarified how the algorithms work in order to obtain the needed information for each step.

2.1.3.1. Detection

In order to understand the stated problem, the topic of collision detection must be touched upon, but what is it exactly. Collision detection, as its name suggests, is how a rigid body *knows* that another one is nearby or occupying the same space. This is done in a matter a little different to the real world, since real objects cannot phase through each other, but instead push each other apart. As explained by Ramtal and Dobre (2014), Kenwright (2012) and Eberly (2010); digital objects must first check if they are intersecting another one, meaning that collision detection takes place not when to objects meet, but instead a frame later, when they are already on top of each other. It is important to note that as the first step towards collision response, collision detection must generate new information in order to continue, this information is if a collision has taken place or not, as well as some extra data like the collision normal vector.

Though many algorithms exist to calculate which objects are intersecting, this chapter will focus on just a couple of them, while explaining how they will be applied in this bachelor thesis.

Broad and narrow phase

Since the collision detection might take into account thousands, if not millions, of different polygons or polyhedra, the collision detection has to be split into two different stages. This is of course done because of performance issues. Having to compute and render sixty times per second every single object already strains a computer in an immense way, meaning that if extra computations can be avoided, this should be taken into account. At this moment is where the differentiation between the broad and narrow phase of the collision detection is born.

As explained by Kenwright (2012), the broad phase is in charge of identify if two objects *might* be colliding, indicating that it does not know for sure if a collision response should be triggered, but a closer check must be done. This aforementioned stage of collision detection usually splits the physical area into different sectors in order to check them separately, allowing for sectors with smaller concentrations of objects to be computed more rapidly, while at the same time completely negating possible collisions of bodies that do not share the same sector. In other words, it tells the program which objects are definitely *not* colliding, while allowing the narrow phase algorithms the detailed checks. Stating the obvious, the broad phase algorithms are extremely fast and effective checks, stopping the necessity for more memory consuming processes when possible.

It must be clarified that for the purposes of this thesis, it is unneeded to create an engine with broad phase capabilities, since the amount of rigid bodies will not be immeasurably great. It was, though, crucial to inform that the broad phase exists and its usefulness in scenarios like video game engines, where incredible amounts of objects are present in each scene.

Continuing with the narrow phase, it is the stage of collision detection where it is assured that two objects *are* colliding or intersecting (Kenwright, 2012), hence the importance for any collision detection algorithm. This stage will be more in depth described with some example of how it works, but mostly the idea behind all them is to know if two objects are on top of each other.

Test intersection with bounding circles

The first algorithm which will be explained for the narrow phase is of course the simplest, but also one of the most useful ones. This algorithm is applied to both particles and circle shaped rigid bodies and its basis is planted on the idea that calculating if two circles intersect is extremely easy and memory efficient. If two particles or circles are occupying any space, they will be intersecting if the distance between their centers of mass is smaller than the sum of their radii (Ramtal; Dobre. 2014, p. 284).

This can be seen in the following formula, and better displayed in figure 2.

$$d < r_1 + r_2 \qquad (26)$$

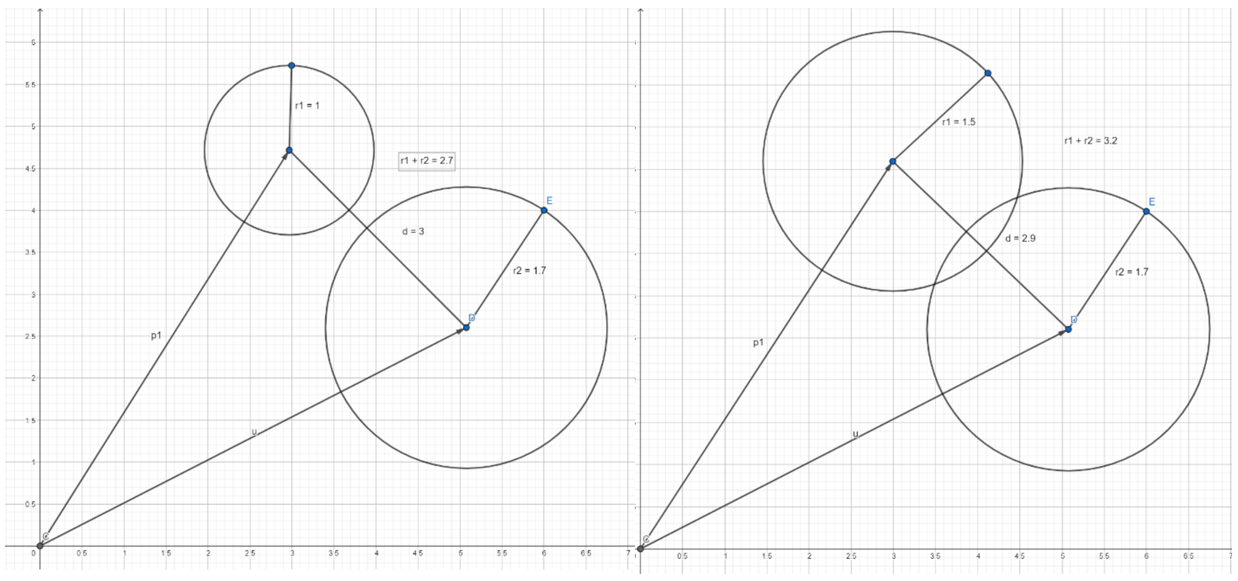


Figure 2: Non – Intersecting (left) vs Intersecting (right) circular rigid bodies.

Moreover, because all calculations will be measured using vectors in the desired engine, it is extremely easy to calculate the distance by subtracting both positions, while both radii should be saved in each particle, so it is just a matter of adding them up.

The beauty of this type of intersection test is that it cannot only trigger a collision response by detecting the collision, it can also provide really meaningful information for the next steps, which other algorithms do not allow. Both the *collision normal*[6] and the *contact point* for all the particles involved can be calculated. The former is easily deduced by some vector math; having the same difference in positions that gave us the distance between both position vectors. The formula describing it is as follows:

$$\hat{\boldsymbol{d}} = \frac{\boldsymbol{p_2} - \boldsymbol{p_1}}{|\boldsymbol{p_2} - \boldsymbol{p_1}|} \ , \qquad (27)$$

where the vector $\hat{\boldsymbol{d}}$ is the collision unit normal and both $\boldsymbol{p_1}$and $\boldsymbol{p_2}$ are the position vectors of each particle.

[6] Collision normal will refer, in this thesis, to the vector describing the collision direction.

Furthermore, Ramtal and Dobre (2014) also explain how with this information the contact point of the collision can also be calculated, which is explained by the fact that a collision works similar to a tangent passing through one point. If two perfect circles are colliding, only one point exist on which the contact is taking place, and at which the force is being exerted. Because of the information at hand, this point is then calculated utilizing the collision normal and the radii of each of the circles:

$$\boldsymbol{c_1} = r_1\hat{\boldsymbol{d}} \tag{28}$$

And using the inverse collision vector, it can be calculated:

$$\boldsymbol{c_2} = -r_2\hat{\boldsymbol{d}} \tag{29}$$

Where $\boldsymbol{c_1}$ and $\boldsymbol{c_2}$ are the collision points, represented by its position vector. They both should represent the same point after position correction, which will be explained in the next chapter. Each of this points represents a position vector with the center of mass of its respective rigid body as its origin (As seen in figure number 3).

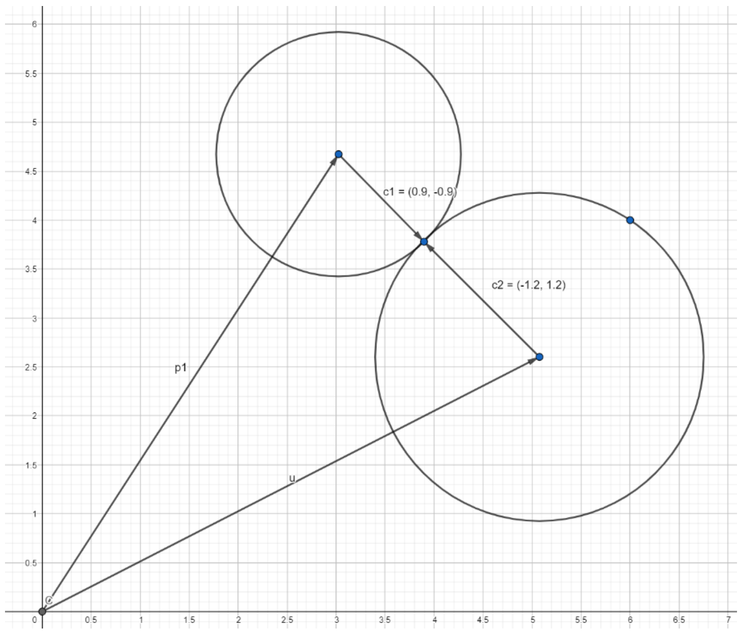


Figure 3: Representation of vectors $\boldsymbol{c_1}$ and $\boldsymbol{c_2}$.

SAT Method

Another common method for collision detection is called the SAT method. The SAT, as described by its acronym for *Separating Axis Theorem* searches for any separating axis between two polygons or polyhedra, in order to deny an intersection. It states: “If two convex objects are not penetrating, there exists an axis for which the projection of the objects will not overlap” (Bittle, 2010).

When two objects are occupying a space, if there exist even one random axis that separates them, the objects are *certainly not* intersecting. Nonetheless, as explained by Eberly (2010), endless amounts of axes exist in a predefined space, making it impossible to measure them all for a correct separation axis. But, and this is where one of the most important components of the SAT comes in, not all axes have to be tested, only the axes parallel to the outward pointing normal to each one of both polygon’s edges. This leaves one with a finite number of axes to test equal to the sum of the number of edges in each polygon.

With this in mind, the calculation of the normal vectors can just be done using some fast vector math for each edge. Each edge will be represented as a vector from one vertex to the next one, in a counterclockwise rotation, leaving one with:

$$\boldsymbol{edge_i} = \boldsymbol{v_{i+1}} - \boldsymbol{v_i} \,, \qquad (30)$$

while allowing for the normal to be calculated by negating the y component and swapping it with the x component, then normalizing the vector. This is not the most intelligent way to calculate a normal, however; for example, a ninety-degree rotation matrix could also achieve the same effect, but for general purposes this way of calculating the normal is extremely easy to replicate in a programmatic environment, while maintaining a real memory efficient process.

After calculating each normal, one is left with a set of normalized vectors that will be used as separation axes for the test.

The next stage of the SAT is checking on every single axis for intersection, which is done by projecting every vertex of both polygons onto the separation axes provided by the set of normals afore calculated. Although this procedure sounds complicated, it is exceptionally easy for both humans and computers to calculate. Knowing the projection formula, as described by Eberly (2010, p. 596),

$$proj(\boldsymbol{v}\,,\ \boldsymbol{u}) = (\boldsymbol{v}\ \cdot\ \boldsymbol{u})\,\boldsymbol{u} \,, \qquad (31)$$

it is easy to create projection vector of each vertex, especially knowing that $\boldsymbol{u}$ has to be a unit vector of the desired axis of which the vector wants to be projected onto.

Now two factors are known, all of the needed projection axes and the projections of each vertex onto these axes. With this information a fast calculation can be done in order to get the minimum and maximum vertex projection of each polygon onto the desired axes. This is done in order to get the interval of the projections of each polygons onto the tested axis to check for separation. If one of the intervals intersects with the other, then no separation was found and the algorithm can continue to the next separation axis. Nevertheless, if at any point the algorithm does not find an intersection, it can break the loop and prove that indeed both polygons are not intersecting at all (Eberly, 2010, p. 398). This creates an *early out* function that allows the algorithm to not test the rest of the axes when a separation was found; just one case loops through all of them, the least probable, when a collision was detected!

In upcoming figure number 4 an illustrated example can be appreciated in order to better understand the algorithm. In this case the algorithm would detect no intersection after checking the axis created with the normal ***v*,** because no intersection of the projected vertexes was found, whereas the one made with the normal ***w*** showed no separation. As stated the test has to be made until one separating axis is found, no intersection; or all axes are checked, collision.

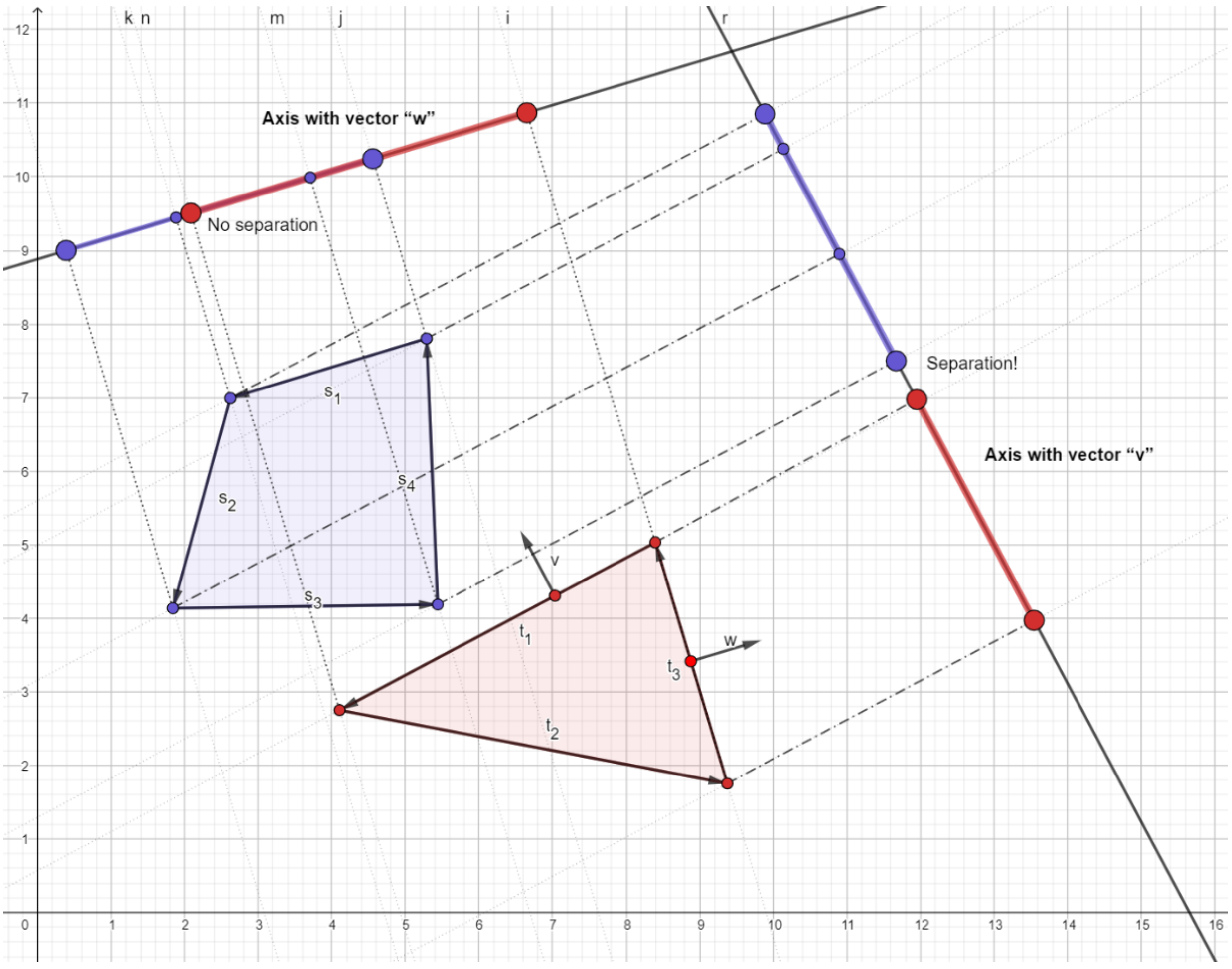


Figure 4: Example of SAT using two axes.

Other collision detection algorithms

Both previously discussed algorithms were explained in detail because of the importance for this thesis. Meaning, that they are both going to be used in the engine developed during this project. The afore stated point does not mean, however, that these are the only usable or best algorithms that exist, only that these were the ones chosen. It would not then be irrelevant to mention two other usable collision detection algorithms like *diagonal intersection* and the *GJK* algorithm, consequently they are going to be swiftly clarified.

First of all the *diagonal intersection* algorithm focuses on checking if the diagonal vectors of a polygon, position vectors from the center of the polygon to the vertexes, intersect with any of the edges of another polygon; and if they do, something know as a *vertex – edge* collision will be detected (Ramtal; Dobre, 2014). The edges in this specific case are clearly defined just as they were on the *SAT* algorithm.

On account of the intersection between two vectors needed to detect the collision during a *diagonal intersection*, different mathematical principles can be applied in order to resolve if an object is colliding or not, these will, nonetheless, not be explained in any further detail.

On the other hand, one of the algorithms for collision detection between two rigid bodies, which is most useful and widespread together with *SAT*, is the so-called *GJK* algorithm (Gilbert – Johnson – Keerthi). This algorithm is used for example in the *dyn4j* engine, a very common Java based physics simulation engine (Bittle, 2010).

The *GJK* algorithm focuses on a mathematical principle of the Minkowski difference (a difference between all points inside two bodies) that allows one to know if the two objects are intersecting. The calculation of this difference creates in turn a new infinitesimal set of points, or which can refer back to the previously mentioned continuum of mass. Consequently, the words of the *dyn4j* engine developers that explain how the algorithm works state: “If two shapes are overlapping/intersecting the Minkowski Difference will contain the origin.” (Bittle, 2010.). This citation allows for an easy to detect intersection check after making the Minkowski difference, with it being that if both of the centers of mass of the objects is inside the Minkowski difference, an intersection has been detected, creating the need for a collision resolution.

As a last point of emphasis, it has to be declared that both the *SAT* and the *GJK* methods are extremely useful and commonly used, but the *SAT* was picked over its contestant because of the low number of objects that will be present at once in the engine. The advantages of the latter of these algorithms relies in that a higher quantity of edges does not necessarily increase the number of calculations, because the algorithm does not take edges into account. This makes it so might be the preferred one to be used in three – dimensional scenarios, where the number of possible separation axes increases exponentially; while the former is, for the most part, more effective in environments where the number of polygons is relatively low.

2.1.3.2. Position correction

With the problem of the detection of each individual collision out of the way, a hitherto discussed problem has arisen, position correction. Briefly touched upon during the *test intersection with bounding circles*, rigid bodies detection takes form one frame after the collision has taken place, when objects are already on top of each other. Due to this, the participants of the collision have to have their position corrected (returned to a "just barely touching state").

The problem at hand might sound complicated, but it is not actually that hard to compute. There is a general idea of what is supposed to happen in order to create this correction, or better, to calculate the distance the objects have to be moved apart. It is, as with every mathematical analysis, needed to know what we are looking for, which in this case is represented by the distance between two objects in an intersection, or better described as the *overlap distance* between of the two objects.

Particle – particle collisions

The easiest instance to aboard is the collision of two particles, or bounding circles, which the correction can be done by just calculating the overlap distance and moving the particles by this distance along the collision axis. This distance is computed by building the addition between the magnitudes of their radii minus the distance between their centers of mass, taking the value as the scalar overlap distance between the two. Furthermore, the correction can be done by scaling the unit collision normal (See *2.1.4.1 Detection – Test intersection with bounding circles* for explanation of how this vector is calculated). The aforementioned steps would be mathematically described as:

$$d_c = \|\boldsymbol{r_1}\| + \|\boldsymbol{r_2}\| - d \qquad (32)$$

$$\boldsymbol{d}_c = d_c\, \widehat{\boldsymbol{d}} \qquad (33)$$

With the vector of total correction calculated, one can then decide how much correction should be applied for each particle. As a result, there is a freedom of choice when applying the collision correction in a physics engine, leaving it to the developer to decide how it should be done.

Examples of this could be depending of which particle is more massive, which particle was moving faster or simply of how particles are organized in the scene.

Particle – wall collisions

A special case is the collision correction of particles against walls, which will be defined for the purposes of this thesis as line segments. With a little bit of trigonometry, the calculations of what the desired correction distance is easy to achieve by following the defined steps by Ramtal und Dobre (2014) in their book *Physics for JavaScript Games, Animation, and Simulations* and defined by the following formula (p. 275):

$$\Delta s = \frac{r + \boldsymbol{d} \cdot \boldsymbol{n}}{\sin(\theta)} \qquad (35)$$

With Δs being the desired distance, r the radius of the particle, $\boldsymbol{d}$ as the position vector from the center of mas of the object to, and perpendicular to, the wall; $\boldsymbol{n}$ the normalized collision normal, and θ the angle generated between the wall and the velocity vector of the particle.

General collisions

Having touched upon the most specific cases of collisions with particles, some kind of correction has to be explained for cases like polygon – polygon or polygon – wall collisions. Unfortunately, these cases are not as easy to calculate as particle collisions, but are in turn way more general, applying to most cases.

Having used collision detection algorithms like *SAT*; one already has the collision unit normal vector, which describes in which direction the collision took place. In the case of *SAT* this collision normal is taken as the axis of least separation found while iterating through all of the possible axes explained in chapter *2.1.4.1 Detection – SAT Method* (Bittle, 2010). This might not be a hundred percent accurate, for example for really fast objects the axis of least penetration can be at the other side of the object, but for cases like this, special measures should be taken into account, like for example decreasing the time step at faster velocities.

Moreover, the mentioned axis of least separation is just the axis that had the least amount of separation between the extremes of the projections of both objects, or rudely explained as the axis where the maximum of the projection interval of the first object was closest to the minimum of the one belonging to the other object (Bittle, 2010). The overlap is on the other hand just the value of the separation of both objects on this selected axis.

So, with this collision unit normal and the overlap the rearrangement of the objects taking part in the collision can be done as before, by just moving each one by a factor of the overlap through the axis of least separation, making it so both objects are just barely touching[7].

2.1.3.3. Collision manifold

After knowing the previous steps of a collision response, before getting to the core of what exactly is the collision manifold, it must be indicated to the reader that this was one of the most difficult parts to piece together of the collision resolution. This was the case because of the huge impact a badly calculated collision manifold has, and the fact that a general collision manifold solution was being searched that could be applied to polygons, not only one which would calculate for example squares or circles. For this reason, a method known as the Sutherland – Hodgman polygon clipping was utilized. But a brief understanding of other methods is in order for learning purposes.

So, with this in mind, the collision manifold is just the stage of collision calculation that is in charge of finding the approximate contact points of two objects, that is, the points at which two objects collide. The reasoning of why it is called manifold may come from the contact points it generates, because it is not limited, normally, to just one, but instead to a set or collection of multiple contact points (Bittle, 2010).

[7] See formula number 24.

Specific cases

Some specific cases mentioned above that might be important for the understanding of how many physics engine work, and that will be described in this thesis, are Axis Aligned Bounding Boxes (AABBs) and circle intersection contact points. The latter was already briefly explained during *2.1.4.1 Detection – Test intersection with bounding circles* and it is only needed to clarify that during this type of collision the collision manifold is unneeded, but the specific contact point is generated by the collision detection algorithm.

On the other side, as explained by Randy (2013) in a practical way, the former of the two aforementioned cases tries to generate the set of points by checking the horizontal and vertical positions of the AABBs which engulf the rigid body in the space and verify where they intersect. This method requires the object to be surrounded in one of this bounding boxes, making it so it is not as precise for polygons while completely disregarding any kind of rotation. Despite of this it is really common in engines because it allows a really easy and efficient way to calculate collisions for objects that do not need the precision of polygons, for example, when the sprite of a player character tries to jump over a barrier.

Sutherland – Hodgman Polygon clipping Algorithm

For more general purposes, a nice algorithm that can help with the calculation of the collision Manifold is the Sutherland – Hodgman Polygon clipping algorithm. As clarified by Sutherland and Hodgman (1974) in their paper explaining the algorithm, the central idea is to check for the intersection points of two polygons, taking into account both the position and rotation of each one of them, making it perfect for the desired engine. The algorithm is divided in two steps. The first step is the distinguishing of which polygon is going to be clipped, with the second one being the clipping itself. A good analogy to understand what is being done is to imagine a triangle barely inside a picture frame; someone with scissors will then cut all the parts of the triangle out, which are not inside the frame (See figure 5).

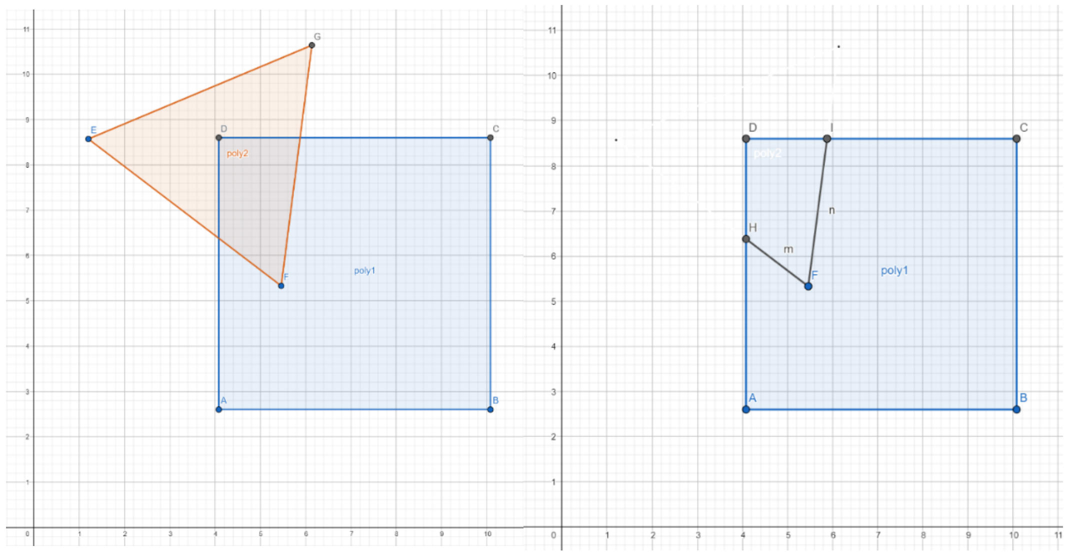


Figure 5: Polygon clipping example.

(Left) Polygon before being clipped – (Right) Polygon after being clipped.

The first step of the algorithm, the *identifying* of which polygon is going to be clipped, will be explained in further detail when the justification of how the algorithm was implemented comes. For the time being, the most important part is to understand how the clipping itself works.

In order to explain how this clipping method works, one must first understand the four rules that compose the algorithm. For the understanding of the representation of these rules, one must imagine two polygons in a space, where all of the edges of each polygons are represented as vectors. Taking two vectors at a time, one can be used to clip the other, taking into account that each vector should come from a different polygon. For the purposes of the explanation the vector performing the clipping will be called a *reference edge* from now on, while the other type of vector, the one being clipped, is considered an *incidence edge* (Bittle, 2010). As a clarification, the reference polygon is the one comprised of all the reference edges, while the incidence polygon of the incidence edges. With this line of thought, and as explained originally by Sutherland and Hodgeman (1974), the following rules must then be applied:

1. If both points of an incidence edge are visible, that is inside the reference polygon, save the terminal vertex of the incidence edge (p. 34).
2. If the incidence edge is completely out of the reference polygon, do not save any new points (p. 34).

3. If the incidence edge starts at the visible side, but ends outside of the reference polygon, save only the intersection between the incidence and reference edge (p. 34).
4. If the incidence edge starts outside the reference polygon, but ends in the visible side, then save both the terminal vertex of the incidence edge in question, as well as the intersection between the incident and reference edges (p. 34).

This process is then applied recursively for all reference and incidence edges, leaving one with a set of points that can be then used to describe the searched for collision manifold (Bittle, 2010). These points are of course the intersection points of the polygons when they are on top of each other.

For further clarification, it is strongly encouraged for the reader to go back to Figure number 5 and perform the four rules above stated having the blue square as the reference polygon.

2.1.3.4. Resolution: Impulses

After the whole process of identifying a collision and calculating the contact points of a collision, the “real physics” aspect of the physical simulation process can be applied. Different types of collision resolution of course exist, but the one that will be important for this thesis is the *impulse method*, meaning that the impulse of the collision has to be calculated in order to resolve it.

Impulse is defined as a force applied in a certain time interval (Ramtal; Dobre. 2014, p. 358):

$$\boldsymbol{J} = \boldsymbol{F}\Delta t = \mathbf{a}\, m\, \Delta t \qquad (35)$$

with its integral form representing the addition of all forces applied during a set interval (Serway; Jewett, p. 253):

$$\boldsymbol{J} = \int_{t_i}^{t_f} \sum \boldsymbol{F}\ dt \qquad (36)$$

The importance of impulses in collision resolution is actually because of the impulse – momentum theorem, which states “The change in the momentum of a particle is equal to the impulse of the net force acting on the particle” (Serway; Jewett, p. 254). Making it so the previous information is useful for the case of collision resolution because of the definition of momentum is stated to be: “[linear momentum] is defined to be the product of the mass and velocity of the particle […].” (Serway; Jewett, p. 248), so the impulse can be expressed as:

$$\mathbf{J} = m\boldsymbol{v}_f - m\boldsymbol{v}_i = m\,(\boldsymbol{v}_f - \boldsymbol{v}_i) \qquad (37)$$

This is incredibly helpful because it gives one the knowledge that the calculation of the final velocity $\boldsymbol{v}_f$ at any given moment (even after a collision) is none other than the initial velocity $\boldsymbol{v}_i$, just before the collision, plus the impulse $\mathbf{J}$ generated by the collision, divided by the mass m of the object in question![8]

$$\boldsymbol{v}_f = \boldsymbol{v}_i + \frac{\mathbf{J}}{m} \qquad (38)$$

A quintessential clarification that is still to be understood, if one knows the lineal velocity after a collision – giving in turn birth to the lineal displacement of the object – is of how the rotational factor of the body in question is changed after the collision. Luckily, as explained before in *2.1.3.4. Angular Kinematics and Torque* the laws of motion apply equally for angular and linear movement, making it so:

$$\omega_f = \omega_i + \mathbf{r} \times \frac{\mathbf{J}}{I} \qquad (39)$$

But, the reason why there is a cross product between the impulse and the vector from the center of mass to the point where the force is being exerted, ($\mathbf{r}$) is because referring back yet again to

[8] The following equation is just a simplification in order to get $\boldsymbol{v}_f$.

the chapter of angular kinematics, the torque acts like a force, but is represented as the cross product between this vector **r** and the force that is being applied (Ramtal; Dobre, 2014). This makes it so the generated impulse has to also be applied through the same point as the forces in order to create a rotation.

The given information is applicable to both objects involved in a collision, though the generated impulse is negative for the second object, because of the third Newtonian law of motion (for both linear and angular movement).

Even though this previous understanding of how impulse influences a collision answers the most important question of what the velocities of an object after a collision are, it still leaves the inquiry of how *impulse* itself is calculated. This is made easier by the explanation made by Ramtal and Dobre (2014), in their already mentioned book, that (p. 359):

$$J = -\frac{(1+C_R)\,\boldsymbol{v}_r^i \cdot \boldsymbol{n}}{{}^1\!/_{m_1} + {}^1\!/_{m_2} + \frac{(rp_1 \times \boldsymbol{n})^2}{I_1} + \frac{(rp_2 \times \boldsymbol{n})^2}{I_2}} \qquad (40)$$

Where J is the impulse magnitude, $\boldsymbol{n}$ the unit normal vector of the collision, m_1 and m_2 the masses of each object; $\boldsymbol{r}_{p1}$ and $\boldsymbol{r}_{p2}$ the position vectors from each object's centers of mass to the collision point; I_1 and I_2 the respective moments of inertia; C_R the coefficient of restitution, which is just the *bounciness* factor of a material, and depends on what the object is made of; and $\boldsymbol{v}_r^i$ the initial relative velocity between both objects, calculated by subtracting the velocities at the point of impact $\boldsymbol{v}_{p1}^i$ and $\boldsymbol{v}_{p2}^i$ (pp. 358 - 359):

$$\boldsymbol{v}_{p1}^i = \boldsymbol{v}_1 + \omega_1 \times \boldsymbol{r}_{p1} \qquad (41)$$

$$\therefore\ \boldsymbol{v}_{p2}^i = \boldsymbol{v}_2 + \omega_2 \times \boldsymbol{r}_{p2}$$

$$\boldsymbol{v}_r^i = \boldsymbol{v}_{p1}^i - \boldsymbol{v}_{p2}^i \qquad (42)$$

With the impulse generated by the collision, having in mind both the angular and lineal components of the movement, one can then proceed to resolve the collision. By adding the velocities generated to the objects in question, in order to define the new generated motion, one can move the bodies after colliding with each other.

2.2. Part II: The *Flutter* Framework

"Flutter is Google's UI toolkit for building beautiful, natively compiled applications for mobile, web, and desktop from a single codebase."

– Google LLC, n.d.

Having finally understood the whole physical and mathematical aspects needed for the desired physical simulation, one can proceed to the explanation of what the *Flutter* SDK is, while also describing why it was chosen as the developing kit for the thesis. This chapter will briefly clarify these aforementioned aspects, while at the same time giving a general understanding of which parts of physical simulation are already included in the *Flutter* SDK.

Some elements, that will not be present in this chapter, are the exact explanations of the inner workings of the *Flutter* framework, or a deep analysis of fundamentals such as its architecture. This chapter should only be taken as a small introduction to *Flutter* in order for the reader to comprehend the reasons for the utilization of the software. For more information one can refer oneself to the *Flutter* documentation[9].

Furthermore, most of the explanations done in this chapter will be done with the writer's previous knowledge of the SDK, as well as following the aforementioned documentation.

2.2.1. What is *Flutter*?

As explained shortly by the Google team in the quote at the beginning of the chapter, *Flutter* is a developing kit that enables cross platform app development. This encapsulates mobile apps (IOS and Android), Web and also desktop applications, making use of the same code basis, and avoids the need of different programming languages when coding for different platforms.

All the programming done in *Flutter* is performed with Google's programming language *Dart*. But, it should be obvious to an avid reader that this language is not as well-known as other programming languages such as Java or C. Nevertheless, it stands its footing for being pretty

[9] Flutter Documentation: https://flutter.dev/docs

useful in combination with the development kit. *Dart* as a programming language is an object-oriented programming language that was first born as a successor to JavaScript, before it was used as the foundation for *Flutter*. Furthermore, this language's syntax can be described to be similar to other OO languages, which was probably made this way in order for programmers to make an easy transition from JavaScript to *Dart.*

However, *Flutter* is accredited by its creators to create native applications, which just means code that is written in a specific programming language for a specific platform. But this aspect might contradict with the fact that the programs are written in *Dart*, instead of for example on *Kotlin* or *Swift*[10]. On further investigation on this issue, one can come to the inquiry of how something can compile with the same programming language on compilers that are completely different. This is where *Flutter* shines: the code can be written in *Dart*, but at building time, the code will compile to its native counterpart and be packaged in a way that matches with the system it is being deployed into – For example as an APK file for Android (Flutter documentation. n.d.). The whole process packages both the *Flutter* SDK and the written code by the user into this end file, making it so it can be bundled for different platforms.

2.2.2. Widgets

A quintessential factor that makes *Flutter* subjectively programmer friendly is the usage of what they call *Widgets*. These are the building blocks of all the applications in *Flutter*, while also being objects that can be built into each other, creating a parent – child relationship. In turn, this relationship makes the child widget adopt the properties of the parent (like position and constraints). Because of this, two advantages arise: Firstly, creating new layouts becomes extremely easy, because widgets always depend of their parents. Secondly, widgets in *Flutter* form a tree like structure that will continuously be built in order, making it so the child widgets will always be reconstructed when the parent rebuilds itself.

Furthermore, widgets can also be customized and be instantiated anew. By extending already stablished abstract widgets, one can basically integrate whatever one desires in the *Flutter* application as a widget. It will be highly encouraged to the reader to keep this fact in mind for future chapters.

[10] Kotlin and Swift are the programming languages for Android and IOS applications, respectively.

2.2.3. Plugins and Packages

Another really important capability that is supported by the *Flutter* SDK is the option of expanding the software when it does not meet one's needs. Through the so-named plugins and packages not only this enlargement of the framework can be achieved, but also allows for the creation for many open source projects, that expand the SDK. This system provides a way for other programmers with more knowledge of a certain area to create kind of *extensions* to the SDK.

It will now be explained what packages and plugins are, and their differences:

- ***Package:*** A package works like a library written in completely *Dart*. This code can then be reused in order to help with the creation of new programs without having to calculate everything anew.
- ***Plugin:*** A plugin works similarly to the before mentioned packages, but it is written natively in whatever programming language it is constraint to (this depends on the desired platform). The methods or functions established in the plugin can then be called from code written in *Dart* and are wrapped in a *package* in order to then be used in a *Flutter* program. This just means that methods and functions programmed natively, can be called from a *Dart* program, increasing the amounts of options when utilizing *Flutter*.

Having this in mind, a plugin is then just a package that is expanded to also include native code, but, as for the perspective of the user, they both work exactly the same.

Knowing this, one further explanation should be given: The physics engine planned for this thesis will take the form of a package, in order for it to be usable by any programmer without any regards to platform. Therefore, the development of the API can be done without being constraint by having to also write native code for specific platforms.

2.2.4. Physics Simulation in *Flutter*

Nonetheless, after now knowing what exactly the *Flutter* SDK is, an enormous question might arise: does such a complete developing kit have any kind of physics simulation? It should be possible for a tool that allows one to create applications for this number of platforms to have a physical simulation engine. however, the answer to the previous question is not as straightforward as one would like it to be, because the answer is: it *kind of* supports simulation.

The indefinite nature of the previously stated answer is born from the fact that *Flutter* seems to have most of the bases covered for a sturdy physics engine, but it just does not have a complete and functioning physics API. An example of this is how AABB's are already implemented in the engine; but are not used for either two- or three-dimensional physics simulation. This can, however, quickly advert to a keen-eyed reader that one dimensional was not stated, because the SDK *does* have libraries for this kind of simulation.

One dimensional physics simulation is used in flutter to create animations for applications. For example, a spring animation can be used in scrollables when one swipes farther away from the last element in a list. This animation will then be triggered to show the effect of going back to the last element.

Another implementation of physics simulation, apart from the official one-dimensional simulation libraries, is the *Dart* port of *Box2D*. *Box2D* is a two-dimensional physics engine for Java, and was adapted to *Dart*. However, as seen in the pub.dev web page[11], the biggest problems with this port is that it is both not an official Google project and that it has not really been tested, having a low coverage (around 28% at the time of writing). Additionally, the package is also not even compatible with *Dart 2*! (pub.dev. 2017).

Because of the low amount of physical simulations tools covered by the *Flutter* SDK, it is easy to see why a sturdier and functional physics simulation engine might be needed.

[11] The pub.dev page is where the open source *Flutter* packages and plugins are published.

3. PROPOSITION AND CONCEPT

Having finally finished with all the theoretical research done in order to advance the thesis forward, it would be wise to explain what exactly will be done with this information. In this chapter the proposition for the development of the engine will be announced and explained.

First of all, it should be indicated that a two-dimensional physics engine will be developed in the programming language *Dart* in conjunction with the *Flutter* SDK. This engine will focus on rigid bodies and will ignore other aspects of physics, such as lighting or Soft bodies.

The engine itself will, as referenced already, focus on rigid bodies, but most specifically, it will try to implement particles and polygons. These objects will react to each other in a physically accurate way.

Consequently, for its development, the algorithms and methods mentioned during the theoretical framework will be used. Not only this, but the engine will be developed as a package for *Flutter* meaning that it *could* be published in the future if needed, allowing other programmers to freely use the engine as they see fit.

Furthermore, in order to show the capabilities of the engine, an example app will also be developed. The application will try to use the engine at its fullest, while helping as a contrast point of what the engine actually brings to the table. Apart from this, the development of this app will help illustrate the amount of code that is avoided when one has a tool at hand, such as the developed engine. As a last remark to this point, the application will represent a real-life use case scenario in which such an engine can be useful.

For demonstration purposes, the scenario of an educational app will be employed for the development. This application will be made with students in mind, who are learning the Newtonian Laws of Motion for the first time. The reasoning of this decision is to demonstrate how physics simulation can make a difference when being applied to an end product.

With reference to what the example application is going to be, the application will didactically try to demonstrate the three laws of motion, as described by Sir Isaac Newton. It will have at least three different screens, each demonstrating one law in a different way.

1. ***Inertia***: The user will have an empty canvas with one object always being spawned in the middle of the screen. This element can be a random polygon. Moreover, objects spawned

will start in resting state, however, in order to demonstrate the first law of motion, the user change the force being applied to the object, making it move.

As mentioned before, the idea is to demonstrate the law of inertia, meaning that the world will not have any kind of collisions or friction, apart from probably the edges of the screen.

2. ***Force***: For the demonstration of the second law of motion, the user will be able to see a defined number of objects and accelerate them. The previously mentioned objects will have an acceleration and mass dictated by user input. After being added to the screen, the elements in question will move, with their motion being in accordance with the force being applied to them. The values for mass, acceleration and force might be displayed on screen.
3. ***Reaction:*** In what could be better described as a free for all, the user will be able to add objects to the screen, being at the beginning in resting state. Objects will start moving across the screen, while colliding with both the rest of the elements, as well as with the walls. For the purposes of the demonstration of the third law of motion, no gravity will be applied, but one might be able to influence the forces in the system with the usage of the device's gyroscope. Additionally, objects that collide with each other will also shortly show the direction of the force after the collision, for a clearer illustration of the third Law of Motion.

4. DEVELOPMENT, EXPERIMENTATION AND SOLUTION

After the first steps into answering the research question of the thesis; with them being the physics basics, the algorithms needed for the development of a physics engine and the core concept of what exactly is going to be developed. One can now turn to the next secondary goal: the implementation of 2D physics simulation into the Flutter SDK through the development of a physics simulation API.

During the extent of this chapter, it will be explored how one such an engine can be developed for the *Flutter* SDK, while at the same time describing the different steps that were taken during the experimentation of the engine. With *experimentation* of course referring to the creation of the example app. This chapter will be crucial for the goals of this thesis, because it will show the challenges of creating such an engine, while demonstrating that it might be incredibly valuable when being used for app development.

This chapter will be divided into two parts. The first component will focus on the development of the physics engine, whereas the second will be the creation of the app itself. Nevertheless, not every single method and function will be explained during this chapter, only the quintessential operations for the understanding of the project will be directly cited. For any further inquiries, the reader will have to be referred to the source code annexed with this thesis.

4.1. DEVELOPMENT OF THE PHYSICS SIMULATION ENGINE

4.1.1. Engine Principles

Before even trying to explain how the development of the engine started or which steps were taken in order to create a functioning physics API, it will be important to refer oneself to the architecture of the software. As a minor note, the architecture that will be shown in this chapter will be the one of the project after being finished, since it deviates from what was originally thought. Nonetheless, it is extremely useful for the understanding of the engine.

The reader will be referred to the appendix A1 to see the complete UML class diagram of the physics engine, but some critical pieces of information will be still be described and exhibited during this chapter.

The general idea of how the architecture of the software was divided was born with the main concept in mind: the physics component should work completely separated from the visual. This is not only because following clean code the UI should always be separated from the functionality of the written code, but also because these two aspects were thought to be *used* independently. Furthermore, the physical simulation component was imagined to be a standalone milestone to be achieved by the engine, allowing not only its usage for cases like game development, but also the creation of new physically animated UI pieces.

Because of the stated before, the graphical segment of the engine was called the `PhysicalCanvas` Widget, which allowed one to add physical objects to a scene, letting them be moved and refreshed completely by this element.

On the other side, the standalone physical classes were created. These classes include, for example, the collisions resolvers or the physical rigid bodies themselves. All of these classes will be further explored in the coming subchapters.

As for the rigid bodies themselves, just like in real life, they were thought to have a set of intrinsic properties that act on them. Examples of these would be mass, moment of inertia and position of the rigid body in the system. With this in mind, objects would then be able to act independently from each other.

In order to keep it as simple as possible, the following objects were used, since it has to be thought as a proof of concept that one such an engine is needed for *Flutter*:

- ***Convex Polygons***: The most general of the shapes used in the engine. These polygons allow for changes both in position and rotation.
- ***Particles***: Basically, circles as two-dimensional rigid bodies. Position is of course a variable issue, but because of the fact that particles are equal on all sides when being rotated, no rotation is be applied to them.
- ***Walls***: The static objects of choice for the engine. Meaning that they even if having a position in the system, there is no motion when forces are applied to them[12]. Static objects are considered to have an infinite mass for the most part, like unmovable objects. These walls have a starting point, an endpoint; as well as a direction to the *positive* side of the wall.

A last aspect that would be interesting to clarify, before taking a deeper look at each component, is how forces are displayed in the engine. Both Force and Torque are their own respective classes. However, force is actually just a vector. This difference in logic is answered by the fact that force also needs a differential in time to be applied in. Forces can be constant, like gravity, or generate a change velocity in an instant, like an impulse. For this reason, the class forces also needed a time of application[13], so it was easier to save together with the force vector in an instance of a class. On one side, for constant forces, the time of application is set to be null. On the other side, forces that are applied in an instant will be exerted in one sixtieth of a second. This value was manually set because *Flutter* refreshes its UI sixty times per second, coming to the conclusion that the value of what an *instant* should represent had to be just one frame. The user of the engine can then also manually decide the time in seconds that a force should be applied on the object; the engine will then exert this force for the desired number of seconds.

[12] Walls are represented by a single line, but one could use several walls to create more complex systems.
[13] See A1.

4.1.2. The Physical Canvas Widget and its Architecture

Following with the explanation of the physics API, the core of the graphical component, which was already expressed to be independent of the physical calculations, was called the `PhysicalCanvas` Widget. As the name implies, and remembering was implied in previous chapters, it is a widget for the *Flutter* SDK, meaning that it can be integrated in *Flutter*'s widget tree in order to be built.

Furthermore, `PhysicalCanvas` extends `StatefulWidget`. This is an abstract class provided by the SDK, that allows the creation of a widget that can be changed and rebuilt, given that there is a change in its state, which is a class of its own. For more information about this class, the reader will have to be referred to the *Flutter* documentation.

But `StatefulWidget` is not the only aspect that was taken from the SDK in order to create the desired widget. Other two classes had to be used. The first of them was employed by the state of `PhysicalCanvas`. This mixin class, called `TickerProviderMixin`, allowed the usage of a *ticker* to be called every frame[14]. This ticker was, as mentioned, used in the `PhysicalCanvas`' state to refresh the physical calculations with a frequency of sixty hertz.

Parallel to the aforementioned ticker, the state of the class `PhysicalCanvas` also had a child while building, which extended a class called `CustomPainter`. For any reader familiar with JavaScript, this class works similar to the `Canvas` element. For the rest, this class is in charge of providing a two-dimensional drawing area. Consequently, the canvas state can then render the objects in the scene every time the screen is refreshed, while at the same time utilizing the intrinsic characteristics possessed by each rigid body for the rendering process.

The diagram on the next page accurately depicts the relationship between the classes mentioned before.

[14] A ticker is kind of like a clock, that calls a function every time it ticks. This function is stablished when the widget is added to the tree. Afterwards, the ticker starts, recursively calling the function in a set time interval.

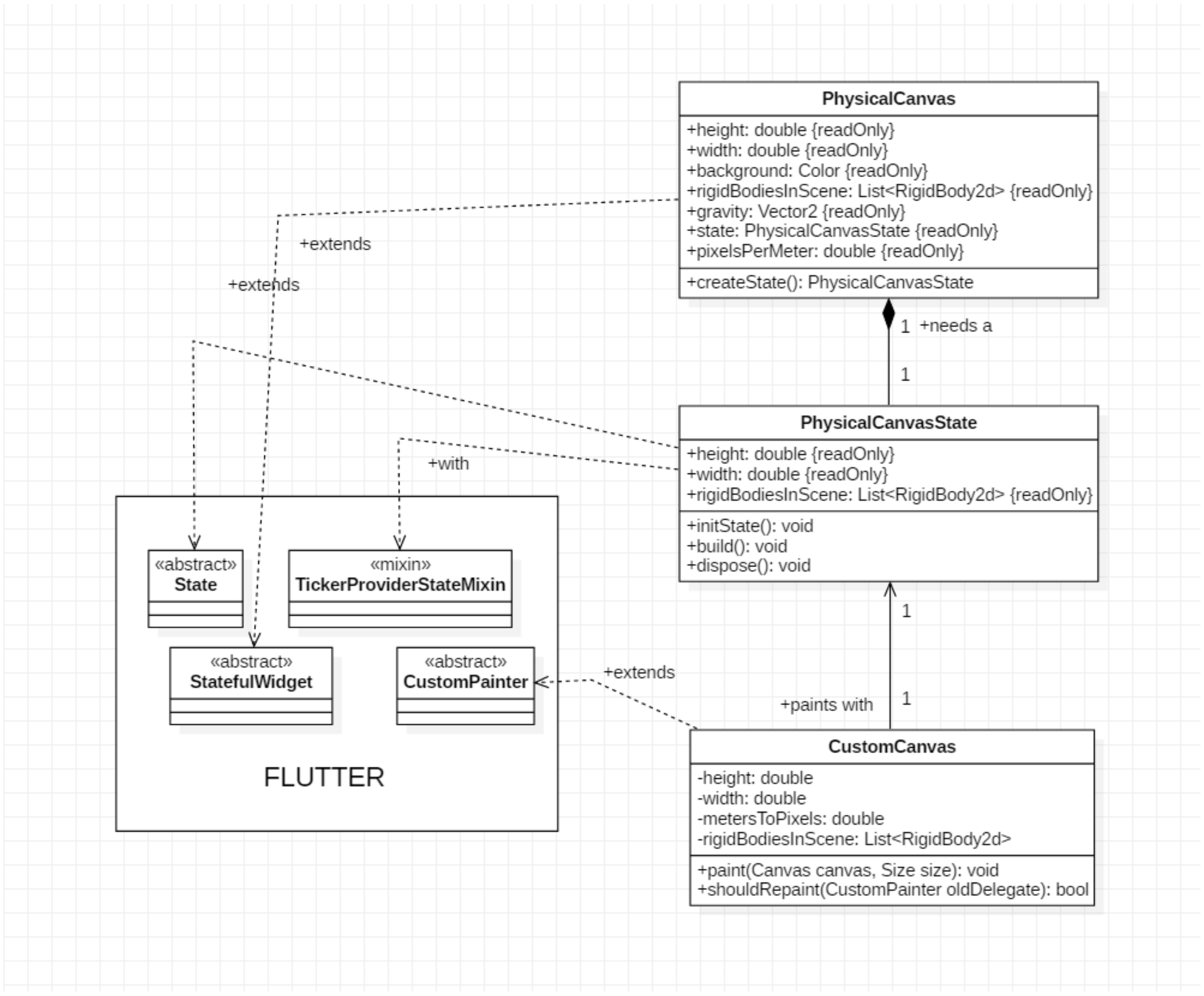

PhysicalCanvas
+height: double {readOnly}
+width: double {readOnly}
+background: Color {readOnly}
+rigidBodiesInScene: List<RigidBody2d> {readOnly}
+gravity: Vector2 {readOnly}
+state: PhysicalCanvasState {readOnly}
+pixelsPerMeter: double {readOnly}
+createState(): PhysicalCanvasState
+extends
+extends
1 +needs a
1
PhysicalCanvasState
+height: double {readOnly}
+width: double {readOnly}
+rigidBodiesInScene: List<RigidBody2d> {readOnly}
+initState(): void
+build(): void
+dispose(): void
+with
«abstract» State
«mixin» TickerProviderStateMixin
«abstract» StatefulWidget
«abstract» CustomPainter
+extends
FLUTTER
1
+paints with
1
CustomCanvas
-height: double
-width: double
-metersToPixels: double
-rigidBodiesInScene: List<RigidBody2d>
+paint(Canvas canvas, Size size): void
+shouldRepaint(CustomPainter oldDelegate): bool

Figure 6: Class Diagram – PhysicalCanvas.

Since the engine was designed to be used by programmers as a package, the class `PhysicalCanvas` asks for the user to provide it with the list of the rigid bodies that should be displayed. These rigid bodies can, of course, be changed or removed at any time, and will be presented accordingly on the screen. The aforementioned relationship, as well as the intrinsic properties of rigid bodies, can be appreciated in Figure 7.

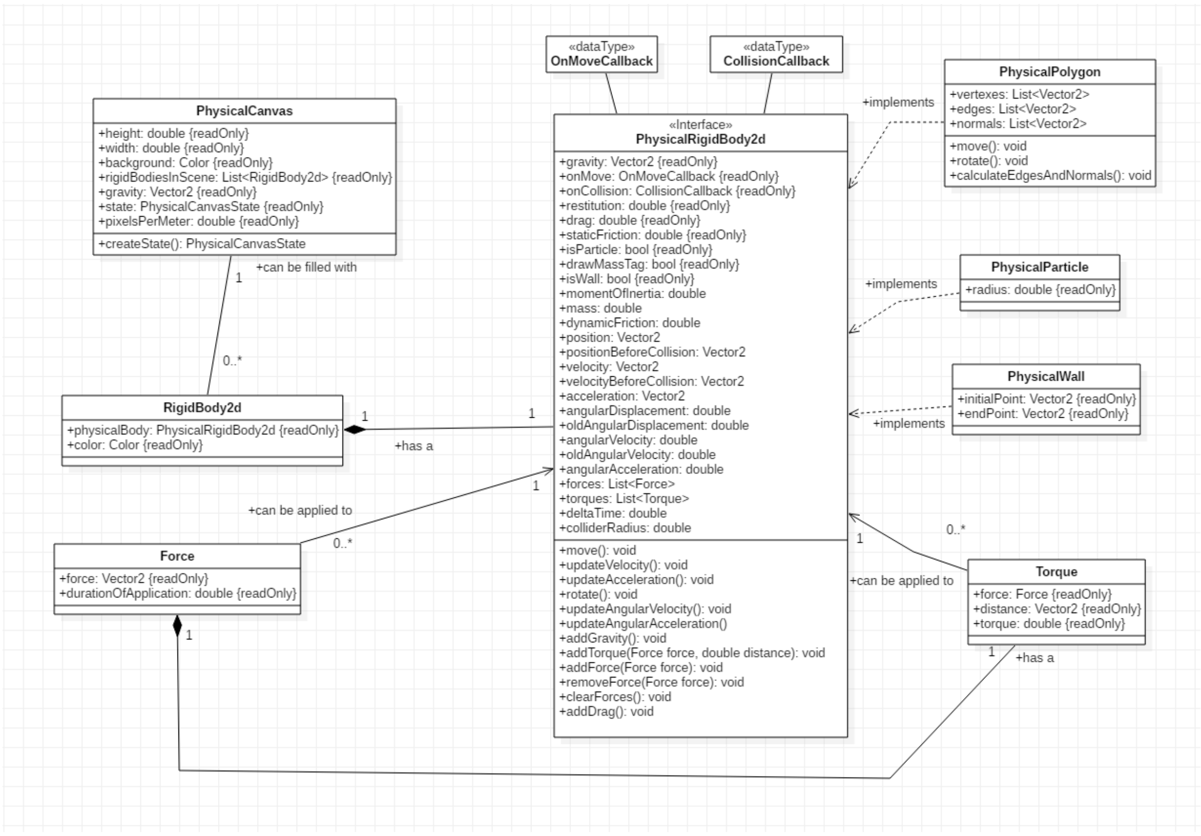

«dataType»
OnMoveCallback
«dataType»
CollisionCallback
PhysicalPolygon
+vertexes: List<Vector2>
+edges: List<Vector2>
+normals: List<Vector2>
+move(): void
+rotate(): void
+calculateEdgesAndNormals(): void
+implements
PhysicalCanvas
+height: double {readOnly}
+width: double {readOnly}
+background: Color {readOnly}
+rigidBodiesInScene: List<RigidBody2d> {readOnly}
+gravity: Vector2 {readOnly}
+state: PhysicalCanvasState {readOnly}
+pixelsPerMeter: double {readOnly}
+createState(): PhysicalCanvasState
«Interface»
PhysicalRigidBody2d
+gravity: Vector2 {readOnly}
+onMove: OnMoveCallback {readOnly}
+onCollision: CollisionCallback {readOnly}
+restitution: double {readOnly}
+drag: double {readOnly}
+staticFriction: double {readOnly}
+isParticle: bool {readOnly}
+drawMassTag: bool {readOnly}
+isWall: bool {readOnly}
+momentOfInertia: double
+mass: double
+dynamicFriction: double
+position: Vector2
+positionBeforeCollision: Vector2
+velocity: Vector2
+velocityBeforeCollision: Vector2
+acceleration: Vector2
+angularDisplacement: double
+oldAngularDisplacement: double
+angularVelocity: double
+oldAngularVelocity: double
+angularAcceleration: double
+forces: List<Force>
+torques: List<Torque>
+deltaTime: double
+colliderRadius: double
+move(): void
+updateVelocity(): void
+updateAcceleration(): void
+rotate(): void
+updateAngularVelocity(): void
+updateAngularAcceleration()
+addGravity(): void
+addTorque(Force force, double distance): void
+addForce(Force force): void
+removeForce(Force force): void
+clearForces(): void
+addDrag(): void
+can be filled with
1
0..*
PhysicalParticle
+radius: double {readOnly}
+implements
PhysicalWall
+initialPoint: Vector2 {readOnly}
+endPoint: Vector2 {readOnly}
+implements
RigidBody2d
+physicalBody: PhysicalRigidBody2d {readOnly}
+color: Color {readOnly}
1
1
+has a
+can be applied to
1
0..*
Force
+force: Vector2 {readOnly}
+durationOfApplication: double {readOnly}
1
1
0..*
+can be applied to
Torque
+force: Force {readOnly}
+distance: Vector2 {readOnly}
+torque: double {readOnly}
1
+has a

Figure 7: Class Diagram – Relationship PhysicalCanvas and RigidBody2d.

An interesting note is that many of the attributes of the classes are public. This is because of two reasons: First, many of these are final, meaning that even if they can be accessed from outside, they cannot be changed. Secondly, the attributes that should be changed by the user could be also wrapped in getters / setters, but it was decided otherwise to save some code (due to the sheer number of attributes in some classes). This is of course a programming decision made by the writer of this thesis, and might not represent the optimal way to do it.

The last important piece of information is that the `PhysicalCanvas` does not have a set size, which can bring two new things to the table. Firstly, the height and width of the canvas can be changed to fill the necessities of every app. And finally, that the pixel per meter ratio can be adjusted manually. With the latter of the arguments just explaining that the representation of how many pixels are one meter in the digital space can be set to meet the developer's desires.

4.1.3. Motion

Continuing with the exploration of the engine, after finishing the explanation of the graphical component, it will be crucial to know how exactly *motion* was handled. Motion, for reasons explained before, should be performed by following the three Newtonian Laws; as a consequence, these were the basis of how objects moved inside the engine.

So, recapitulating, the three laws are based on force, and how changes in forces influence objects in motion. For the case of the engine, the objects in question are the rigid bodies already described, while the forces are represented by both the classes *Force* and *Torque*. As seen in the theoretical framework, a change in the net sum of the forces also creates a change in the acceleration of the object in question. Because of this, the intrinsic properties of each object had to include the individual forces exerted on it, the acceleration, the velocity and the position of the object in the system[15].

Because of the previously mentioned premise, each Rigid body had to be able to calculate its own new properties, depending on the applied force. Most of the calculations take part in the `move()`-method, which looks as follows:

```
/// Moves the Rigid Body
void move() {
  oldPosition = position.clone();
  addGravity();
  rotate();
  updateAcceleration();
  updateVelocity();
  position.add(velocity.scaled(deltaTime));
  positionBeforeCollision = position.clone();
  if (onMove != null) onMove();
}
```

Code Fragment 1: The move()- Method.

As seen in the Code Fragment 1, the method contained inside each rigid body object, updates all motion properties of the rigid body, including acceleration and velocity.

The method `updateVelocity()`, as well as `move()`, utilizes the semi – implicit Euler formulas in order to calculate the velocity and position of the object at any given time[16].

[15] Refer back to Figure 7 – see class PhysicalRigidBody2d.
[16] 17 and 18 – Semi implicit Euler formulas for Position and Velocity.

Whereas `updateAcceleration()` applies the second Law of Motion in order to get the new acceleration, just after adding up all forces being applied on the object (See Code Fragment 2).

```
void updateVelocity() {
  /// Semi Implicit Euler:
  /// v_(n+1) = v_n + aΔt
  velocity.add(acceleration.scaled(deltaTime));
  velocityBeforeCollision = velocity.clone();
}

void updateAcceleration() {
  final tempForceVector = Vector2.zero();
  final tempForces = List.from(forces);

  /// Add saved forces and reduce the
  /// time of application of each force by 1/60 sec.
  for (final force in tempForces) {
    tempForceVector.add(force.force);
    if (force.durationOfApplication != null) {
      if (force.durationOfApplication <= 0) {
        /// If time of application is completed,
        /// remove the force
        forces.remove(force);
      } else {
        force.durationOfApplication -= 1 / 60;
      }
    }
  }

  /// a = F/m
  acceleration = tempForceVector.scaled(1 / mass);
  if (velocity.length2 != 0.0) addDrag();
}
```

Code Fragment 2: The updateVelocity() and updateAcceleration() Methods

Furthermore, the *move* method includes a call to the `rotate()`-method. This method, as implied by its name, is in charge of rotating the rigid body; with its calculations being similar to the ones made for lineal movement. One can imagine these processes mirroring each other, but with the rotational movement employing the formulas for angular movement[17].

Nonetheless, one of the biggest differences to `move()`, is that the `rotate()`-method has a need for a moment of inertia as well as torque (instead of mass and force respectively), making it so both of these have to be provided and changed by the user of the engine. An exception to this rule, regarding moments of inertia, were regular polygons of three, four, five and six sides;

[17] 21 and 22 – Formulas for Difference in Angular Displacement and Velocity.

for which the engine calculates them by itself. This was done because these have easy to use formulas, which were inserted in the engine for quality of life.

Another aspect of rotation that was also treated differently to displacement was the fact that rotational displacement, velocity and acceleration are interpreted as scalars in the engine; whereas the lineal counterparts are two-dimensional vectors. This made it so the rotational aspects were easier and more efficient to calculate.

As a final note for the `move()`-method, it also checks for an on`Move()`-callback function. This function is given to the rigid body on the time of instantiation, and if existent, it will be called every time the object moves. The usefulness of such a function cannot be underestimated, since it gives the user higher control of how objects move in the engine, without having to change the engine itself.

4.1.4. Collisions

Already knowing how motion was applied in the engine, the next step to achieve a system that makes sense for calculations of rigid bodies, is the process of collision resolution. This aspect took the most time to research and implement, since as seen in the theoretical framework, the algorithms needed are fairly complex and time consuming to understand.

Collisions were then achieved by separating them in the four steps explained during the theoretical framework: detection, rearrangement, manifold generation and, finally, resolution (as explained in chapter *2.1.3 – Collisions*). However, in order to accomplish the needed calculations, and as seen in the following diagram, the collisions were separated depending on the objects that were being analyzed.

Two collision resolver classes were created, one for the convex polygons and another one for the particles. These two classes were utility classes with just one public method each: a method to check collisions of this type (See Figure 8).

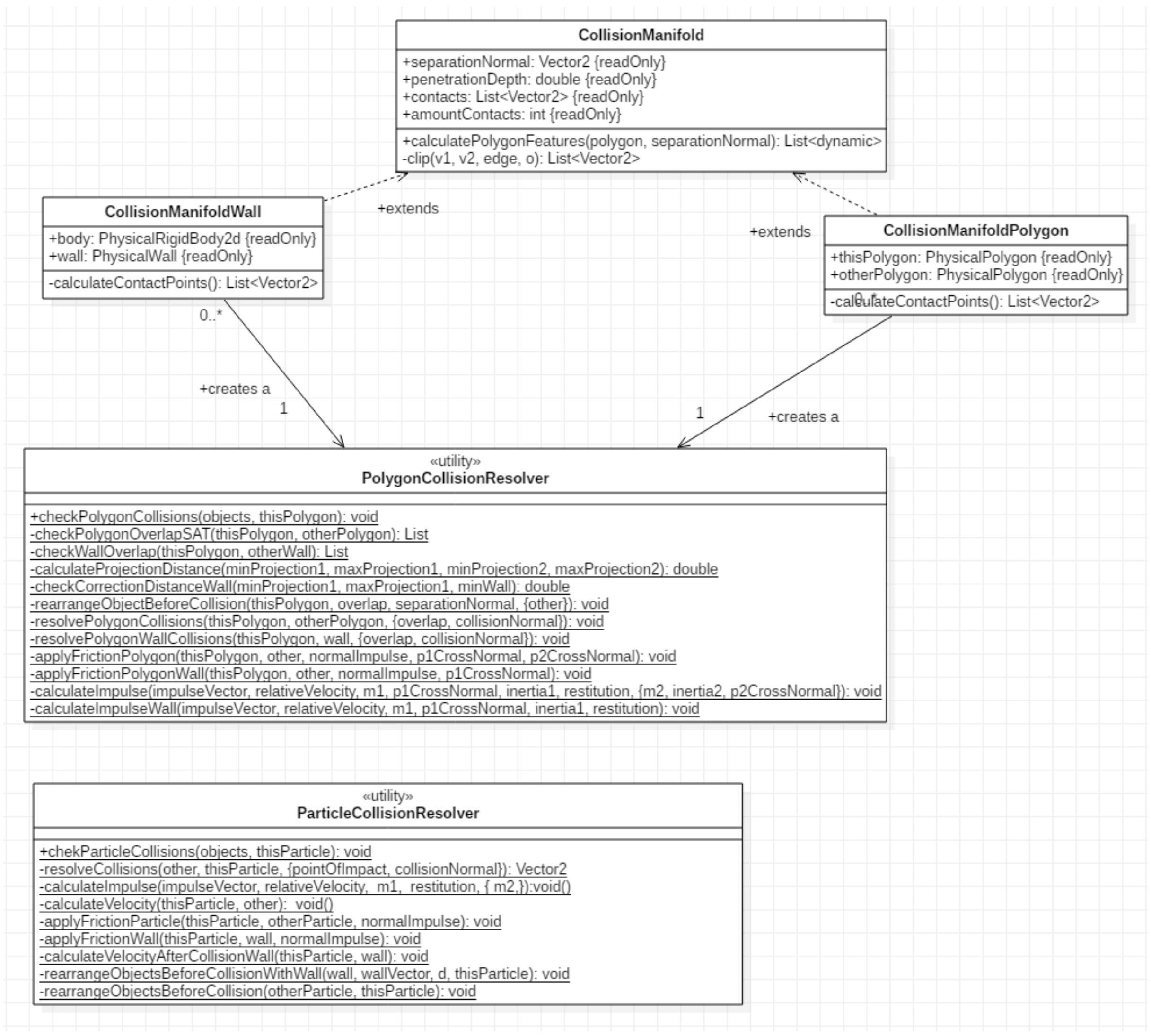

CollisionManifold
+separationNormal: Vector2 {readOnly}
+penetrationDepth: double {readOnly}
+contacts: List<Vector2> {readOnly}
+amountContacts: int {readOnly}
+calculatePolygonFeatures(polygon, separationNormal): List<dynamic>
-clip(v1, v2, edge, o): List<Vector2>
+extends
+extends
CollisionManifoldWall
+body: PhysicalRigidBody2d {readOnly}
+wall: PhysicalWall {readOnly}
-calculateContactPoints(): List<Vector2>
CollisionManifoldPolygon
+thisPolygon: PhysicalPolygon {readOnly}
+otherPolygon: PhysicalPolygon {readOnly}
-calculateContactPoints(): List<Vector2>
0..*
0..*
+creates a
1
1
+creates a
«utility»
PolygonCollisionResolver
+checkPolygonCollisions(objects, thisPolygon): void
-checkPolygonOverlapSAT(thisPolygon, otherPolygon): List
-checkWallOverlap(thisPolygon, otherWall): List
-calculateProjectionDistance(minProjection1, maxProjection1, minProjection2, maxProjection2): double
-checkCorrectionDistanceWall(minProjection1, maxProjection1, minWall): double
-rearrangeObjectBeforeCollision(thisPolygon, overlap, separationNormal, {other}): void
-resolvePolygonCollisions(thisPolygon, otherPolygon, {overlap, collisionNormal}): void
-resolvePolygonWallCollisions(thisPolygon, wall, {overlap, collisionNormal}): void
-applyFrictionPolygon(thisPolygon, other, normalImpulse, p1CrossNormal, p2CrossNormal): void
-applyFrictionPolygonWall(thisPolygon, other, normalImpulse, p1CrossNormal): void
-calculateImpulse(impulseVector, relativeVelocity, m1, p1CrossNormal, inertia1, restitution, {m2, inertia2, p2CrossNormal}): void
-calculateImpulseWall(impulseVector, relativeVelocity, m1, p1CrossNormal, inertia1, restitution): void
«utility»
ParticleCollisionResolver
+chekParticleCollisions(objects, thisParticle): void
-resolveCollisions(other, thisParticle, {pointOfImpact, collisionNormal}): Vector2
-calculateImpulse(impulseVector, relativeVelocity, m1, restitution, { m2,}):void()
-calculateVelocity(thisParticle, other): void()
-applyFrictionParticle(thisParticle, otherParticle, normalImpulse): void
-applyFrictionWall(thisParticle, wall, normalImpulse): void
-calculateVelocityAfterCollisionWall(thisParticle, wall): void
-rearrangeObjectsBeforeCollisionWithWall(wall, wallVector, d, thisParticle): void
-rearrangeObjectsBeforeCollision(otherParticle, thisParticle): void

Figure 8: Class Diagram – The Collision Resolvers.

As a last important note, each rigid body also expects a `onCollisionCallback`() function as a parameter, which is in charge of calling a user defined function, having access to all the collision information at the time of collision. This includes both objects taking part in the collision, as well as the generated impulse[18]. A quintessential usage for such a function could be, for example, the usage of triggers for game development.

[18] See Figure 7. PhysicalRIgidBody2d – onCollision method.

The Particle Collision Resolver

The first collision resolver, which is going to be explained, is the `ParticleCollisionResolver`. As described by its name, it was the class in charge of checking and resolving the collisions of particles. Referring back to chapter *2.1.3.1 Detection*, one can appreciate how the collision detection for particles was performed exactly as explained by the *test intersection with bounding circles*. Utilizing the difference of radii and distances between particles, collision detection was achieved. It is advisable to turn back to the aforementioned chapter if any doubts with the algorithm might arise.

As for how particle detection against walls was done, it must be stated that it was more complicated. The following steps were taken in order to identify Particle – Wall collision[19]:

1. First, two position vectors were calculated from the center of the particle. One to the initial and one to the end point of the wall.
2. The projections of both calculated vectors onto the wall was calculated. These will be called ***Initial Projection*** and ***End Projection*** in the next steps.
3. The normal vector from the center of the particle to the wall was calculated by subtracting the ***Initial Projection*** to the vector from the center of the particle to the initial point of the wall . This vector was named ***d***.
4. Finally, a check was made: if the magnitude of ***d*** resulted smaller than the radius of the particle in question and the magnitude of the addition of both projections onto the wall smaller than the length of the wall, then a collision was detected (Ramtal; Dobre, 2014).

The described algorithm can be appreciated in the following code fragment:

```
final other = objects[i] as PhysicalWall;
final wallVector = other.endPoint - other.initialPoint;
final unitWallVector = wallVector.normalized();

final vectorToWall1 = other.initialPoint -
thisParticle.positionBeforeCollision;
final vectorToWall2 = other.endPoint -
thisParticle.positionBeforeCollision;

final tangentialVectorToWall =
unitWallVector.scaled(vectorToWall1.dot(unitWallVector));
```

[19] The vector math will not be explained, only the concepts necessary for the understanding of the algorithm.

```
/// Normal vector to wall
final d = vectorToWall1 - tangentialVectorToWall;

/// Addition of vector projections to wall
final projectionAddition =
unitWallVector.scaled(vectorToWall1.dot(unitWallVector)) +
    unitWallVector.scaled(vectorToWall2.dot(unitWallVector));

if (d.length < thisParticle.colliderRadius && projectionAddition.length <
wallVector.length) {
/// Rearrange and Resolve ... Removed for clarity
}
```

Code Fragment 3: Particle – Wall Collision Detection.

As a reiteration to previous knowledge explained in the theoretical framework, collision detection with bounding circles returns the collision normal and the contact point, so it does not need extra collision manifold generation. Furthermore, the just detailed algorithm (*Particle – Wall* collision) also generates these same factors:

- The collision normal is of course the $\boldsymbol{d}$ vector.
- The contact point is just the scaling of the normalized $\boldsymbol{d}$ vector ($\widehat{\boldsymbol{d}}$) by the magnitude of the particle's radius. This will yield the position vector of the contact point from the particle's center.

To conclude particle collision, it is in order to know that both *Particle – Particle* and *Particle – Wall* collision's next steps, in order to achieve a complete collision resolution, were applied as explained in the theoretical framework and will not be reiterated. Because of this one can refer oneself to chapters covering the needed methods and formulas if needed. These aforementioned chapters are *2.1.3.2. Position Correction* and *2.1.3.4. Resolution: Impulses*. The code fragments for these can be found in the accompanying source code.

The Polygon Collision Resolver

The next step in the explanation would be the `PolygonCollisionResolver`, which was in charge of checking and resolving possible collisions between polygons. Even when compared to the `PolygonCollisionResolver`, the `PolygonCollisionResolver` took way more effort to develop. However, since most of the needed algorithms were already deeply explained during the theoretical framework, a quick overview of what this resolver does will be done to illuminate the reader. For a more detailed explanation of the algorithms it is recommended to consult previous chapters.

First of all, the collision detection was performed using the *SAT* algorithm, just as described in chapter *2.1.3.1. Detection – SAT Method*, and can be appreciated in the following code fragment:

```
static List _checkPolygonOverlapSAT(
  PhysicalPolygon thisPolygon,
  PhysicalPolygon otherPolygon,
) {
  /// Add all edge - normals of both polygons to a single list
  final projectionNormals = thisPolygon.normals + otherPolygon.normals;
  double projectionOverlap = 0;
  double minProjectionOverlap = double.infinity;
  Vector2 correctionNormal = Vector2.zero();

  /// Check all possible axes, having in mind
  /// all normals
  for (final projectionNormal in projectionNormals) {
    final projections1 = List<double>();
    final projections2 = List<double>();
    for (int i = 0; i < otherPolygon.vertexes.length; i++) {
      projections2.add(otherPolygon.vertexes[i].dot(projectionNormal));
    }

    for (int i = 0; i < thisPolygon.vertexes.length; i++) {
      projections1.add(thisPolygon.vertexes[i].dot(projectionNormal));
    }
    projections2.sort();
    projections1.sort();
    projectionOverlap = _calculateProjectionDistance(
      projections1[0],
      projections1[projections1.length - 1],
      projections2[0],
      projections2[projections2.length - 1],
    ).abs();

    /// If there is an axis of separation, then
    /// return false: Objects are NOT colliding
    if (!(projections1[0] <= projections2[projections2.length - 1] &&
        projections2[0] <= projections1[projections1.length - 1])) return
[false];

    /// If the checked axis shows no signs of
    /// separation, then continue checking the next axes.
    ///
```

```
      /// Also: Save and compare the axes of minimum
      /// separation in order to perform rearrangement
      if (projectionOverlap < minProjectionOverlap) {
        minProjectionOverlap = projectionOverlap;
        correctionNormal = projectionNormal;
      }
    }

    /// If all the axes were checked and no separation
    /// was found, return true: The polygons ARE colliding
    return [
      true,
      minProjectionOverlap,
      correctionNormal,
    ];
  }
```

Code Fragment 4: SAT Algorithm.

As further clarification, the return value when a collision is detected also returns the minimum projection overlap and the correction normal, which are needed by the rearrangement step.

Nonetheless, the previous algorithm represented only collisions between polygons. As for the case of *Polygon – Wall* collisions, a modified version of the *SAT* algorithm was used using the normal vector to the wall. Because this vector is the only possible separation vector provided by the wall, this algorithm then took place by utilizing all normals from the polygon, plus this one normal generated by the wall.

Shortly after, the next phase of the collision resolution had to be completed. This was the position correction of each polygon, which took place by moving each object back by a factor of the overlap amount through the collision normal. A more detailed explanation can be appreciated in chapter *2.1.3.2. Position Correction – General Collisions.*

As for the collision manifold generation, the next step to take, the Sutherland – Hodgman algorithm was used. This helped with both *Polygon – Polygon* and *Polygon – Wall* collisions. Nonetheless, a keen eye and a good memory will reveal that in the chapter explaining this type of manifold generation one aspect was missing: Reference and incidence object recognition. This was made on purpose in order for it to be specified how it was made in the code, since there are several possible ways to perform this final result. Consequently, the former aspect was calculated by following the subsequent set of rules (Bittle, 2010):

1. Firstly, the farthest vertex of each polygon through the separation normal was searched for.

2. Then, with this vertex, both left and right edges connected to this vertex were checked, in order to discover which of them was most perpendicular to the separation normal.

3. With these two edges identified, one for each polygon, one could then compare which of them was most perpendicular to the separation normal. This edge was then decided to be the reference edge, as a result making the polygon containing this edge the reference polygon.

The programming of the first two steps of this ruleset can be appreciated in the following code fragment:

```
/// Code fragment inspired by:
/// Dyn4j Organization. “Contact Points Using Clipping”, 2011.
/// http://www.dyn4j.org/2011/11/contact-points-using-clipping (30. Dec.
2019).
List<dynamic> calculatePolygonFeatures(
  PhysicalPolygon polygon,
  Vector2 separationNormal,
) {
  /// Search for the farthest vertex through the separation
  /// normal attained during collision detection
  int vertexIndex = 0;
  double max = separationNormal.dot(polygon.vertexes[0]);
  for (int i = 0; i < polygon.vertexes.length; i++) {
    double projection = separationNormal.dot(polygon.vertexes[i]);
    if (projection > max) {
      vertexIndex = i;
      max = projection;
    }
  }

  final Vector2 farthestVertex = polygon.vertexes[vertexIndex];
  Vector2 vNext;
  Vector2 vPrevious;

  if (vertexIndex == 0) {
    vNext = polygon.vertexes[vertexIndex + 1];
    vPrevious = polygon.vertexes[polygon.vertexes.length - 1];
  } else if (vertexIndex == polygon.vertexes.length - 1) {
    vNext = polygon.vertexes[0];
    vPrevious = polygon.vertexes[vertexIndex - 1];
  } else {
    vNext = polygon.vertexes[vertexIndex + 1];
    vPrevious = polygon.vertexes[vertexIndex - 1];
  }

  /// Save left and right edges
  final Vector2 leftEdge = farthestVertex - vNext;
  final Vector2 rightEdge = farthestVertex - vPrevious;

  leftEdge.normalize();
  rightEdge.normalize();

  /// Check if left or right edge should be returned
  if (rightEdge.dot(separationNormal) <= leftEdge.dot(separationNormal)) {
    return [
      farthestVertex,
      farthestVertex,
      vPrevious,
    ];
  }
else {
    return [
      farthestVertex,
      vNext,
      farthestVertex,
    ];
  }
}
```

Code Fragment 5: Reference polygon recognition.

Having the information of the reference and incidence polygon, the clipping could then be completed following the four Sutherland – Hodgman polygon clipping rules as described chapter *2.1.3.3. Collision Manifold – Sutherland – Hodgman Polygon clipping Algorithm*.

In addition to the last steps, one needed to also generate and apply the impulse in order for the objects to change direction and position. By taking a look at the formulas for impulse [20] and the new velocities[21], one can infer that all the information needed is now provided and can be used in order to resolve the collision. As a consequence, the engine applied this formulas as they were, yielding really positive results.

[20] 41 – The impulse formula.
[21] 39 and 40 – Formulas for angular and lineal velocity depending on the generated impulse.

4.2. Development of the example app

After the whole process of creating the engine, the programming of the example app was needed, in order to proof that such an engine is indeed helpful for cross platform app development. All in all, the application took less time than expected, having in mind that the engine overtook a huge portion of the work.

The application was created being divided in three main screens, each one representing one of the three Newtonian Laws of Motion. Furthermore, they were given short names describing each law: Inertia, Forces and Reactions. These are not the official names of the laws, but it was better than naming each screen first, second and third. Because of the importance and independence of each screen, this following chapter will explain the development of each screen separately.

Every calculation made in this screens, that does not involve UI management, is made through the engine. This is because the engine itself, when used in conjunction with the described PhysicalCanvas, only needs a description of how it should act. For example, gravity or the intrinsic properties of each object are important factors that the engine needs from the user, but how they physically act, is calculated by itself.

On the other side, all the explanations for the laws were taken from quotes directly cited in this paper, for any further allusion, it will be requested for the reader to look up the reference page of this thesis.

Nevertheless, this application is not only composed of the physical aspects and representation, the design must also be referenced. The design of the application was done by following the guidelines of Google's Material Design, which are for the most part integrated in the *Flutter* SDK. Consequently, a darker color pallet was chosen for the representation, because of personal preference of the developer.

As a final note before proceeding to the explanation of every single screen, that was developed for the app: the app was tested using a Huawei Mate 10 Pro with Android version 9 (Pi), for Android; and an iPhone XS Max with IOS 13. This is important to have in mind, because it might be possible that the application is not optimized for older versions of these operating systems.

4.2.1. The Inertia Screen

Even though, the three laws were given individual names for didactic purposes, *Inertia* is actually the only one of these laws which is commonly referred this way. As described in past chapters, inertia describes how objects will only change their state of motion when subjected to a change in the forces exerted on it. Because of this the Inertia screen was made to contain a `PhysicalCanvas` widget, together with a short explanation of the law.

The `PhysicalCanvas` takes three quarters of the screen, being initialized with four `PhysicalWalls`, as constraints for the canvas, and a `PhysicalPolygon` in the center. The latter of these is a regular polygon initially generated with a number of sides of either five or six, while being spawned at resting state.

Gravity and drag are deactivated in this scenario, whereas restitution and friction are set to an arbitrary value between zero and one. Because of this, if a force is applied, and shortly after deactivated, the object will slow down after every collision with the walls.

The user experience that was planned for the user to have, was for him to understand how objects will not move unless an interaction is made. Because of the law of inertia, the center polygon should not have any motion at all, since it is at rest: the system is stopped (Figure 9, left). The only possible way for the object to change its velocity, is for a force to be applied. This aforementioned force can then only be achieved by the user tapping the *change force* button, which will lead one to a new dialog (Figure 9, right). Here, the user can rotate an arrow around a representation of the object in the center of the scene, and tap accept. Shortly after, the rigid body, which had encountered itself on a resting state, will start its motion towards the selected direction of the force, accelerating in this same direction.

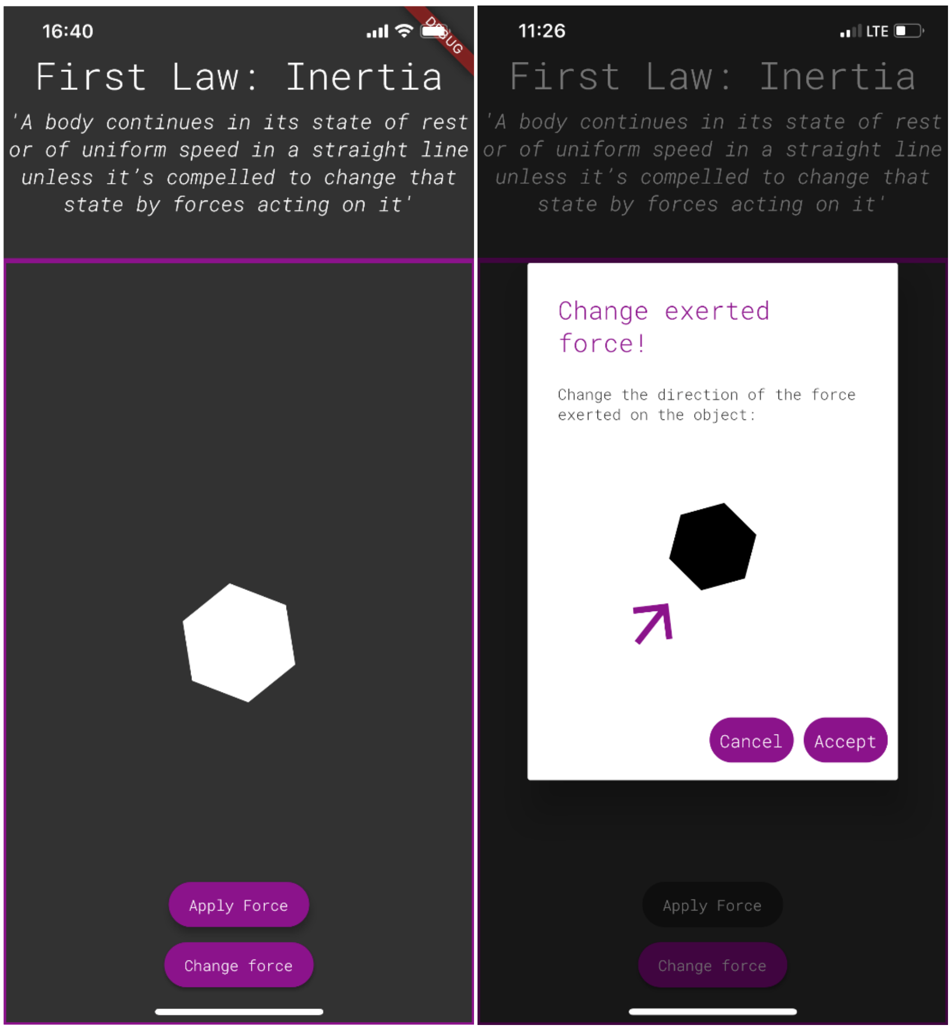

16:40
First Law: Inertia
'A body continues in its state of rest or of uniform speed in a straight line unless it's compelled to change that state by forces acting on it'
Apply Force
Change force
11:26
First Law: Inertia
'A body continues in its state of rest or of uniform speed in a straight line unless it's compelled to change that state by forces acting on it'
Change exerted force!
Change the direction of the force exerted on the object:
Cancel
Accept
Apply Force
Change force

Figure 9: The Inertia Screen.

(Left) The main inertia screen – (Right) The change force dialog

Another aspect that should be noted is that the *change force* button can be pressed at any time, making it so one can always change the desired direction of the exerted force.

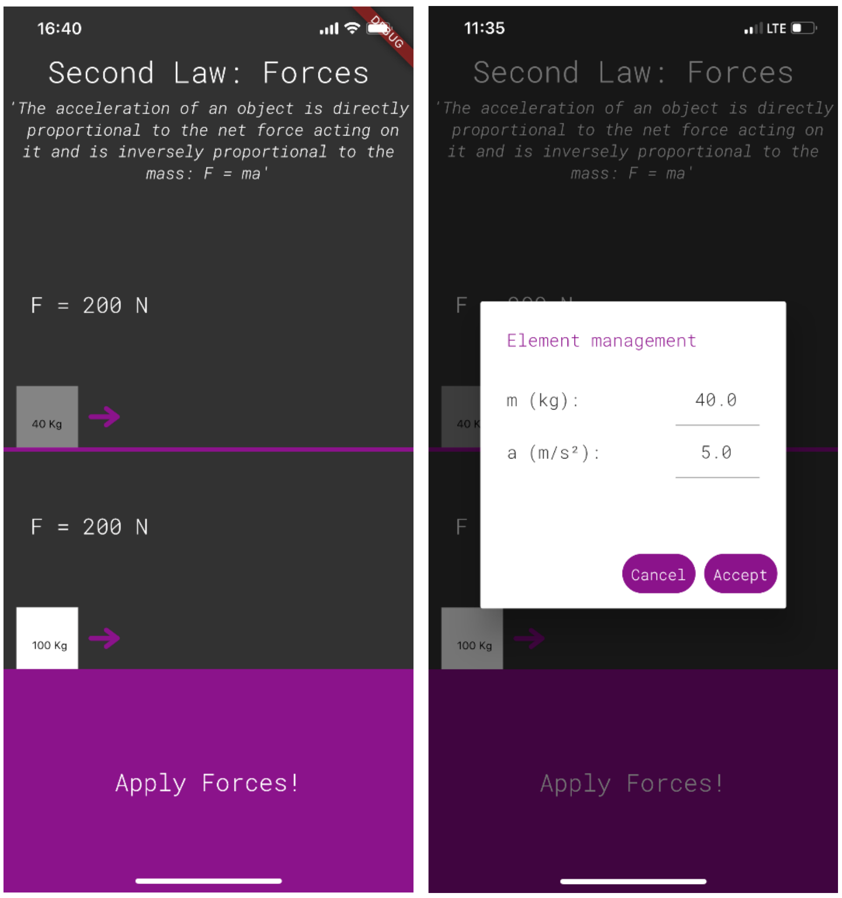
An interesting interaction, which was also created for the user, was the ability to keep applying the force, or being able to stop it. Meaning, that when one decides in which direction the force is going to be exerted, a constant force will keep moving the object in that direction; however, by pressing the *apply / stop force* button the user can influence this force in the desired way. Nonetheless, if the user determines to stop the force, and because of the law of inertia, the object will keep moving in the direction it was previously, but this time with a constant velocity.

The previous argument generates an intuitive loop which helps the user understand how the law works. The user can see how an object performs while being subjected to forces, while at the same time noticing that an object will conserve its velocity when no new forces are being applied.

4.2.2. The Forces Screen

Concerning the *Forces Screen*, the idea was to have a comparison of how forces affect the acceleration of an object. This was decided to be explained by comparing how forces acted on two different objects.

As seen in Figure 10 (Left), two `PhysicalCanvas` were placed on the screen, one after the other, occupying one quarter of the screen each. At the left side of each canvas, a square, represented by a `PhysicalPolygon`, was placed. Both the mass of the square and the desired applied acceleration were made editable, so the user could change them by tapping the canvas. Moreover, the mass of the object was represented with the opacity, the more massive, the opaquer the body is; making so a visual differentiation of masses was also possible.



'The acceleration of an object is directly proportional to the net force acting on it and is inversely proportional to the mass: F = ma'

m (kg): 40.0

a (m/s²): 5.0

Figure 10: The Forces Screen.

(Left) The main forces screen – (Right) The element management dialog

On the other side, when the user taps one of the two displayed canvases the element management dialog is opened (Figure 10, Right). In this dialog one can, as user, change the desired mass and acceleration of the object in question. This also changes amount of force that is going to be applied in order to achieve the desired acceleration. The reasoning behind this, in order to demonstrate the second law, as for the user to be able to edit both factors which are normally displayed together in the equation (acceleration and mass), while letting the device compute the needed force.

Lastly, the force that was going to be generated, for an object of the stated masses and with the desired acceleration, was displayed by both a vector and a numerical representation. Both of this representations of the force give visual feedback to the user in order for him to understand the proportionality between the acceleration and force. Nonetheless, the drawn vectors were made to become visually bigger when the force also becomes larger, producing an effect which allows the user to further understand this proportionality with images, not only with numbers.

Furthermore, the user also gets to experience the difference in accelerations created by applying a force of an object. Since the objects will start moving when the user decides to exert the forces, he can also observe the difference in the motion of objects with different masses. For example, how an object with double the mass of the first can still have the same acceleration, if the generated force is also doubling. In this case the user will be able to see a bigger vector in the more massive object, but will also be able to appreciate how both bodies move exactly the same.

Apart from all this, the systems created to be both friction- and gravitation- less.

4.2.3. The Reactions Screen

With both the first two Newtonian Laws being described, only the law describing the reactive force created after a collision between two objects was left to be developed. The mentioned screen also had a description of the law at the upper part of it, while being followed by three quarters of the screen enveloped by the `PhysicalCanvas`.

Inside this `PhysicalCanvas`, the list of rigid bodies was initially instantiated with several different `PhysicalWalls`, to give the objects something else to collide with. Furthermore, the whole canvas was wrapped in a gesture detector, which would allow the user to tap anywhere inside of it. This tap will in turn generate a `PhysicalParticle` at the desired position with a random restitution value.

The `PhysicalParticles` created when tapping the screen are added to the `RigidBody2D` list, which was given as a parameter to the canvas at instantiation time. Because of how *Dart* manages objects, being that they are passed by reference, new entries can always be added to the list without the need to repeat the instantiation of the `PhysicalCanvas`. These particles will be then drawn by the canvas, and react to the other objects inside it.

The mentioned descriptions can be appreciated in the following figure:

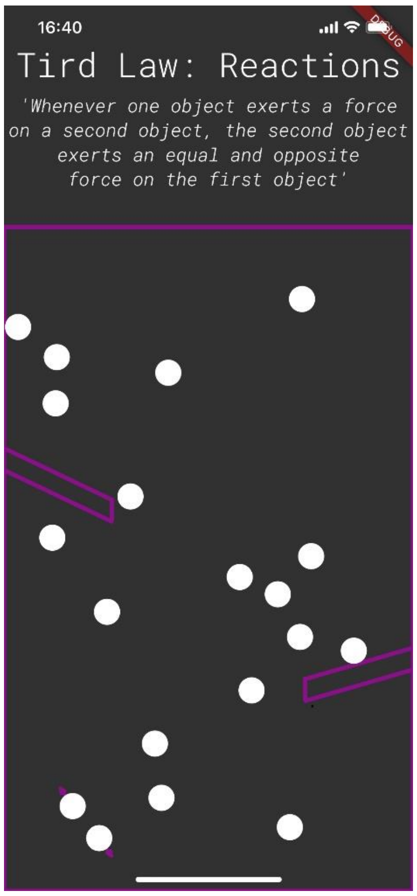

16:40
DEBUG
Tird Law: Reactions
'Whenever one object exerts a force on a second object, the second object exerts an equal and opposite force on the first object'

Figure 11: The Reactions Screen.

The phone’s gyroscope was then used in order to generate a constant force in the direction that the device was rotated to, when inside the *Reactions Screen*. Consequently, this force was used to influence all the objects in scene.

It should be stated that the force exerted when tilting the device is a constant force with the same magnitude for all objects, so if the objects had different masses, they would have different accelerations. Because of this, a force applied the same way it was in this screen should not be considered to act like the gravitational pull; since, if it was like gravity, all objects would be then subjected to the same acceleration, resulting in a variable force when accounting to the variable masses of the objects.

Apart from the mentioned elements of the third page, one of the most important aspects representing forces being generated in opposite directions was the fact that, utilizing the `onCollision` parameter, a function was created and passed on to the engine. This callback function was in charge of examining the impulse generated during each collision and drawing the direction of the force vectors generated by the collision.

Lastly, the Reactions screen was the one which used the engine the most, because of the number of objects and collisions in the scene. In figure number 12 a short sequence diagram can be appreciated.

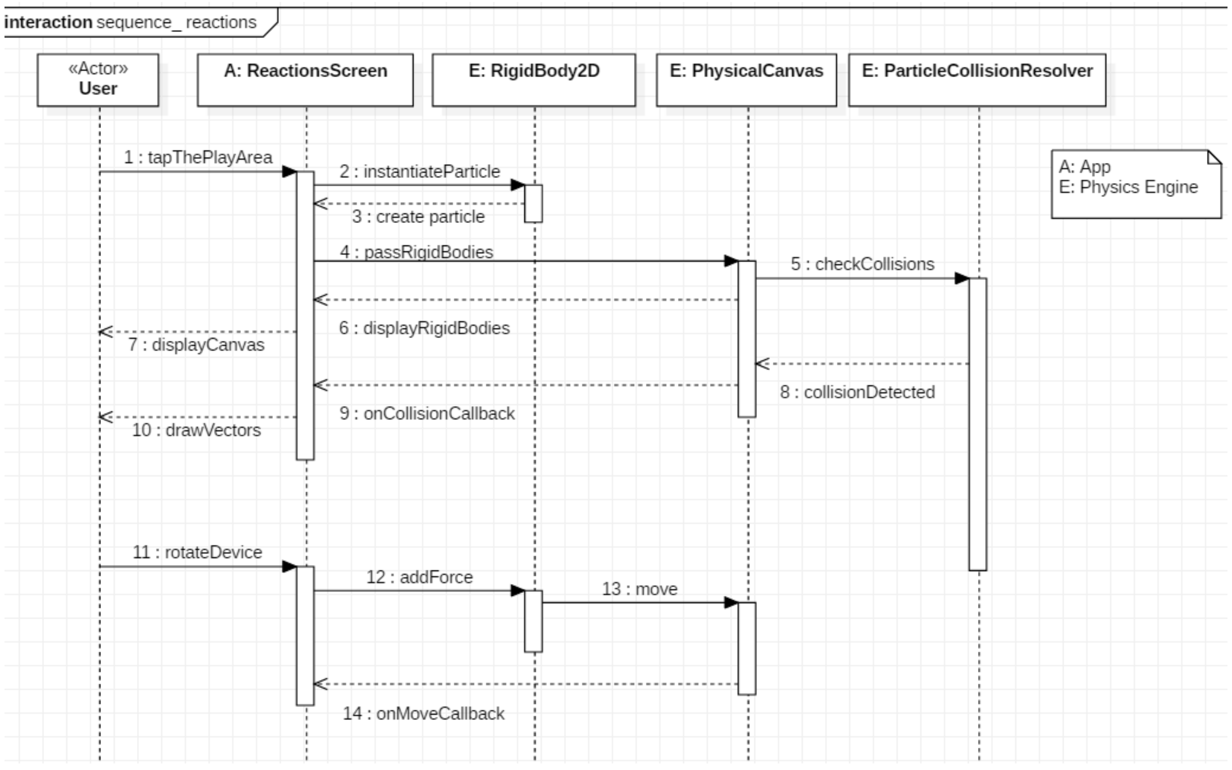


Figure 12: Sequence diagram for the Reactions Screen

Commencing with the interaction of the user, a `PhysicalParticle` is instantiated. This particle has, as stated before, a position, velocity, acceleration, etc.; and is added as an entry to the list of `RigidBody2D`. Shortly after, during the instantiation of the `PhysicalCanvas`, this list is passed to the canvas, making it draw all rigid bodies and start refreshing every one-sixtieth of a second. Every time the screen is refreshed, all the particles are also moved to their new positions, while the `ParticleCollisionResolver` checks if any collision has taken place. If a collision was detected, the resolver also resolves the collision, assigning new acceleration and velocities; but also calling the `onCollisionCallback`, which draws the vectors of the collision.

At the same time, the user is also allowed to perform the other possible interaction: tilting the device; adding in turn a new force. This force is added every one-sixtieth of a second to each particle and depends on how much the phone was rotated. It is also only added with the application time of a sixtieth of a second, but every time the canvas refreshes, a new check for the rotation is also done, resulting in a new added force. This loop will be called endlessly until one either changes screen or shuts the application down.

5. ANALYSIS OF RESULTS

After the laborious job of creating a whole functioning two-dimensional, rigid body focused physics engine; plus going through the arduous task of creating an application in order to identify the benefits of such an engine, one does come to an important question of what this solved. A direct answer to this question might be taken as subjective, since the experience of each developer may vary when encountering such a problem as the one at the core of this thesis. However, the author and developer of this project will try, during the lengths of this chapter, to best explain his point of view and experiences.

Along the lines of this chapter, it will be described what the discoveries during the development of the engine were; as well as explaining the reader which were the benefits and problems of working with such an engine. As a consequence, the whole analysis will be made from the point of view of a developer, who needs to create an application in a similar use case scenario.

As a start for the analysis of the results created by the experimentation with the engine, a huge, if not the most important, argument is how much time and code having a physics calculation tool saves. While implementing the application, it was incredible to see how most of the time spent was in the development of the UI, whereas very little, if not none, was needed in order to calculate physical movements. At the same time, during this development cycle, the engine was able to take on all the physical calculations, making it so the developer had no need to create extra support functions or methods in order to create realistic physical reactions. Examples to this, was how much code the previously mentioned algorithms like *SAT* or the *Sutherland – Hodgman* took in the engine. If one wanted to create such an application without an engine, all of these algorithms would have to be coded inside the application itself, amounting to a huge increase in the code's length.

This also can guide one to the next point of analysis, which is the existence of such engines for other platforms. Examples like the afore declared *Box2D* and *dyn4j* exemplify the presence of a market for such libraries. So, it is only natural to conclude that *Dart*, and by aggregation *Flutter,* needs such an API.

However, a pressing argument cannot be denied by a skillful debater: with enough skill, knowledge and time one can program such functions by oneself, inside of the app. As shown by this thesis, the development of the physical functions can be done having enough experience in the field. This can be claimed because this thesis shows that even an undergraduate can achieve such an API, be it a simplified version of one.

This last point can lead one to another conclusion, being that the *time* plays an important factor. The code that has to be developed is equivalent to the whole engine, which means that if a developer needs to create another application that fits the same physical concepts, the same code has to be recreated. So, if this is true, why not simply have a program that already deals with all the trouble? This is the reason libraries exist for different programming languages, providing solutions to almost every problem.

Together with the aforementioned argument, *knowledge* is also a huge roadblock. Not only one needs to be comfortable with the language one is using, but without a tool that overtakes the physical aspect, one needs to also be proficient enough in this aspect of physics. Obviously, physics is a whole different field, and a bold statement could be that most programmers lack the experience developing physically accurate motion. Because of this, such an engine could provide the ideal scenario for app developers, who might be oblivious to the field of physics.

Nevertheless, with previous daring declarations being potentially true, why aren't there any official implementations of two- or three-dimensional physics APIs in *Flutter*? A fair argument to be made against the creation of an engine might be, that mobile applications do not need one, being backed up its lack of existence in this branch of development.

Mobile applications do not normally have the need for physics to be integrated, because there might not be a real point of it. There might be no actual reason to want physical simulation in an application only containing formularies, or lists with a couple of buttons, for example.

But, the human being is always trying to improve. Together with the creation of a physics engine, new, attractive animations could be achieved by utilizing physics simulation. One should think for a moment back to the example app: Imagine all of the animations accomplished there; every single movement does not have to represent an object, it could represent a button, a colored division, a two-dimensional scrollbar.

The previous statement is also not even taking into account that use case scenarios exist where physical simulation is needed. Learning applications or games are examples of possible products that could be done with such an engine. One could be able to completely design an application using *Flutter*'s material design, while at the same time accomplishing the needed physical representations!

6. CONCLUSION

All in all, with the long journey finally over, which was meticulously set forth at the beginning of this thesis, one needs to turn back and analyze the goals that were seeded; take a look at the hypothesis and observe if it still holds true.

Because of the previous simple inquiry, this last chapter of the thesis will emphasize what can be learned from each section of this project, while trying to generate a conclusion. The research goals, question and hypothesis will be taken into the spotlight in order to see what went smooth, or in which aspects a similar future work can watch upon in order to improve.

Firstly, an overview of all the chapters in the thesis is in order. During the theoretical framework, which of course took a large portion of the thesis, the core principles and algorithms were learnt. Without this, the development of the physics engine would have been impossible, since not knowing the physical formulas, or the mathematical principles, only would have brought speculation. This of course does not exclude the fact that the studied principles might not be the most effective, common or efficient ones; they were, though, the ones chosen to create the engine. Consequently, a deeper analysis could have brought forth newer algorithms unknown to the writer. Nevertheless, since this thesis was planted to be a proof of concept that such an engine is both possible and needed; the employed methods were good enough to achieve a descent API.

Examples of algorithms that might perform better in other circumstances could be the GJK method for convex polygon collision detection; the usage of AABBs or object-oriented bounding boxes for general collision detection. The latter two of these are, of course, extremely useful for game development, were a collider for a sprite can either be a rectangle, or an amalgamation of several jointed bounding boxes.

Apart from the already mentioned methods, something that would be good to inform the reader is about both concave polygons as well as constraints. The former of the two should be self-explanatory as to what they are, but what is not as obvious is how they have to be computed completely different to convex polygons. Making the algorithms needed for such a collision detection completely different to those employed in the thesis. Because of this reason, they were excluded.

As for constraints, they are basically rigid bodies that kind of *stick* to each other, allowing for the creation of more complex structures. These were also explored during the thesis, but were not included in the final product because of both time and complexity.

However, as seen in the actual length of the thesis adding more aspects to the engine would increase the time investment exponentially, while at the same time not furthering the validity of the arguments; so, they were decided to be cut off. This does mean, though, that future work could involve similar aspects.

The two final improvements that would be interesting to apply to the developed engine would be the separation of the broad phase, as explained in the collision detection chapters; and the finding of ways to fix the tunneling problems when objects move too fast. Because of the same reasons stated in the paragraph above, this solutions were not implemented. These are, still, important factors for further development of the engine.

Nevertheless, taking this bleak look at what could be improved is not the only factor that will be mentioned in this final interpretation of the thesis. Another important facet to take in is how the project was developed smoothly.

When comparing both the chapter providing the concept and the one just directly following it, explaining the development, one can easily see how the concept was applied just as planned. Both the creation of the engine and the mobile application ended with satisfactory results, which can be clearly seen and appreciated when using the created app. Not only this, but the engine itself can be employed as planned when programming an application with the *Flutter* SDK. Because of this reasons, it would be fair to state that both the research and development elements of the thesis were successful in achieving the goals that were wanted to be met.

The last aspect that would be crucial to criticize, is about the partial subjectivity of the experimentation with the engine. The root to this, is that the person testing the engine was the same programmer who developed it, meaning that the engine itself would still need a lot of work in order to be commercial or broadly distributed.

This, however, does not contradict the usefulness of the results. Yes, the developer himself tested the engine while creating the application, but the engine was thought to be a prototype. It is intended as a proof of concept of the value of one such API, be it a more complete one, for the *Flutter* SDK. Consequently, the fact that *at least* one such scenario exists, where an engine like this one would be useful, could already be considered an important discovery and an argument pro its creation.

Now then only two core pieces of information remain to be answered: the analysis of both the hypothesis and the goals of the thesis.

First of all, the goals will be observed. As stated before, this thesis serves as a proof of concept that a two-dimensional physics simulation engine is needed for the API. This is, of course, because it was proven with the experimentation of a possible real-world scenario. This thesis did serve to develop the aforementioned engine, plus the implementation of the mobile application using it; marking the completion of the primary goal.

As for the complementary goals, every single one specifying the investigation, exploration and study of the physical factors, the needed algorithms for development, the interactions between rigid bodies and collision responses, were covered. This includes from the first to the third specific goals and can be appreciated all throughout the theoretical framework.

The implementation of the 2D physics engine into the *Flutter* SDK, as well as the application of the previously mentioned concepts in said engine should be obvious to perceive, because the engine was developed and used.

The sixth goal, the justification and exploration of the developed engine's architecture, can be seen in the chapter explaining the development of such an engine. Moreover, the engine's architecture, methods and functionality are not only explained, but displayed with code fragments and diagrams.

At last, both the integration of the engine and the analysis of this integration can be understood while reading the last two chapters before this conclusion; as well as by taking a look at the developed application. So, it is not outlandish to state that these goals were also met.

The final mention would be then to explore what happened to the hypothesis. However, it was already mentioned many times before that this thesis does serve as a proof of concept of the need of a physics engine for the *Flutter* SDK. This is, again, reinforced by the fact that during the development of the application no physical aspects had to be worked upon; there was no need for implementing this physical characteristics manually, just because there was a physical simulation API to fall back upon. Furthermore, this also reduced the amount of code and time invested in programming the app drastically.

In the hypothesis the lack of knowledge was also stated, of both the needed algorithms as well as the necessary physics, which can be proven through the amount of work that was invested during this thesis in order to create such an engine. Every single concept and algorithm, that was applied in order to create the engine, would have to be manually coded by the developer of an application, when considering the option of not having this extra tool.

All in all, this thesis will serve to create a statement: physical engines might be needed in more fields that they are, to the current day, used on. Not only this, but the remnants of this thesis could already be both utilized for creation of new innovative apps, as well as allowing further development of this engine. Hence, before closing this thesis, the writer would like to politely address the reader and ask:

Would you use a physics simulation engine for the creation of mobile apps,
or would you rather develop everything needed by yourself?

REFERENCES

1. Augustyn, Adam; et al. (2019 [Last edited]). Torque. In Encyclopedia Britannica. https://www.britannica.com/science/torque.
2. Baraff, David (1997). An Introduction to Physically Based Modeling: Rigid Body Simulation I—Unconstrained Rigid Body Dynamics [PDF File]. Retrieved from Carnegie Mellon University – Robotics Institute. https://www.cs.cmu.edu/~baraff/sigcourse/notesd1.pdf.
3. Beck, Kevin (2018). How to Calculate the Angular Velocity. Retrieved from https://sciencing.com/convert-rpm-linear-speed-8232280.html.
4. Bittle, William (2010, January 1). SAT (Separating Axis Theorem). Retrieved from http://www.dyn4j.org/2010/01/sat/.
5. Bittle, William (2010, April 13). GJK (Gilbert – Johnson – Keerthi). Retrieved from http://www.dyn4j.org/2010/04/gjk-gilbert-johnson-keerthi/.
6. Bittle, William (2011, November 17). Contact Points Using Clipping. Retrieved from http://www.dyn4j.org/2011/11/contact-points-using-clipping/.
7. Eberly, David H (2010). *Game Physics – 2nd ed.* Amsterdam, Holland: Elsevier.
8. FAQ (n.d). Retrieved from https://flutter.dev/docs/resources/faq.
9. Flutter Documentation (n.d). Retrieved from https://flutter.dev/docs.
10. Gaul, Randy (2013, June 17). How to Create a Custom 2D Physics Engine: Oriented Rigid Bodies. Retrieved from https://gamedevelopment.tutsplus.com/tutorials/how-to-create-a-custom-2d-physics-engine-oriented-rigid-bodies--gamedev-8032.
11. Hardie R.P.; Gaye R.K. [Translation] (1991). *Complete Works (Aristotle) – Fourth Edition* [PDF File]. Princeton, N.J..
12. Kenwright (2012). *Game Physics: A Practical Introduction*. n.p: Kenwright.
13. Merriam-Webster Inc (n.d). “Physics”. Retrieved from https://www.merriam-webster.com/dictionary/physics.

14. Merriam-Webster Inc (n.d). Simulation. Retrieved from https://www.merriam-webster.com/dictionary/simulation.

15. Nykanen, Niilo (2013, June 14). Basics of Angular Acceleration and Rotational Moment of Inertia. Retrieved from http://blog.rw-america.com/blog/bid/304231/Basics-of-Angular-Acceleration-and-Rotational-Moment-of-Inertia#:~:targetText=Angular%20acceleration%20(%CE%B1)%20can%20be,Rad%2Fsec%5E2).

16. Pronost, Nicolas (n.d). Game Physics: Numerical Integration" [Power Point Slides]. Retrieved from Utrecht Universiy at https://perso.liris.cnrs.fr/nicolas.pronost/UUCourses/GamePhysics/lectures/lecture%205%20Numerical%20Integration.pdf.

17. "pub.dev" (n.d). Retrieved from https://pub.dev/.

18. Ramtal, Dev; Dobre, Adrian (2014). *Physics for JavaScript Games, Animation, and Simulations: with HTML5 Canvas*. New York, USA: Apress.

19. RJS Tech (2018, June 4). Flutter Custom Paint Tutorial | Build a Radial Progress. https://medium.com/@rjstech/flutter-custom-paint-tutorial-build-a-radial-progress-6f80483494df.

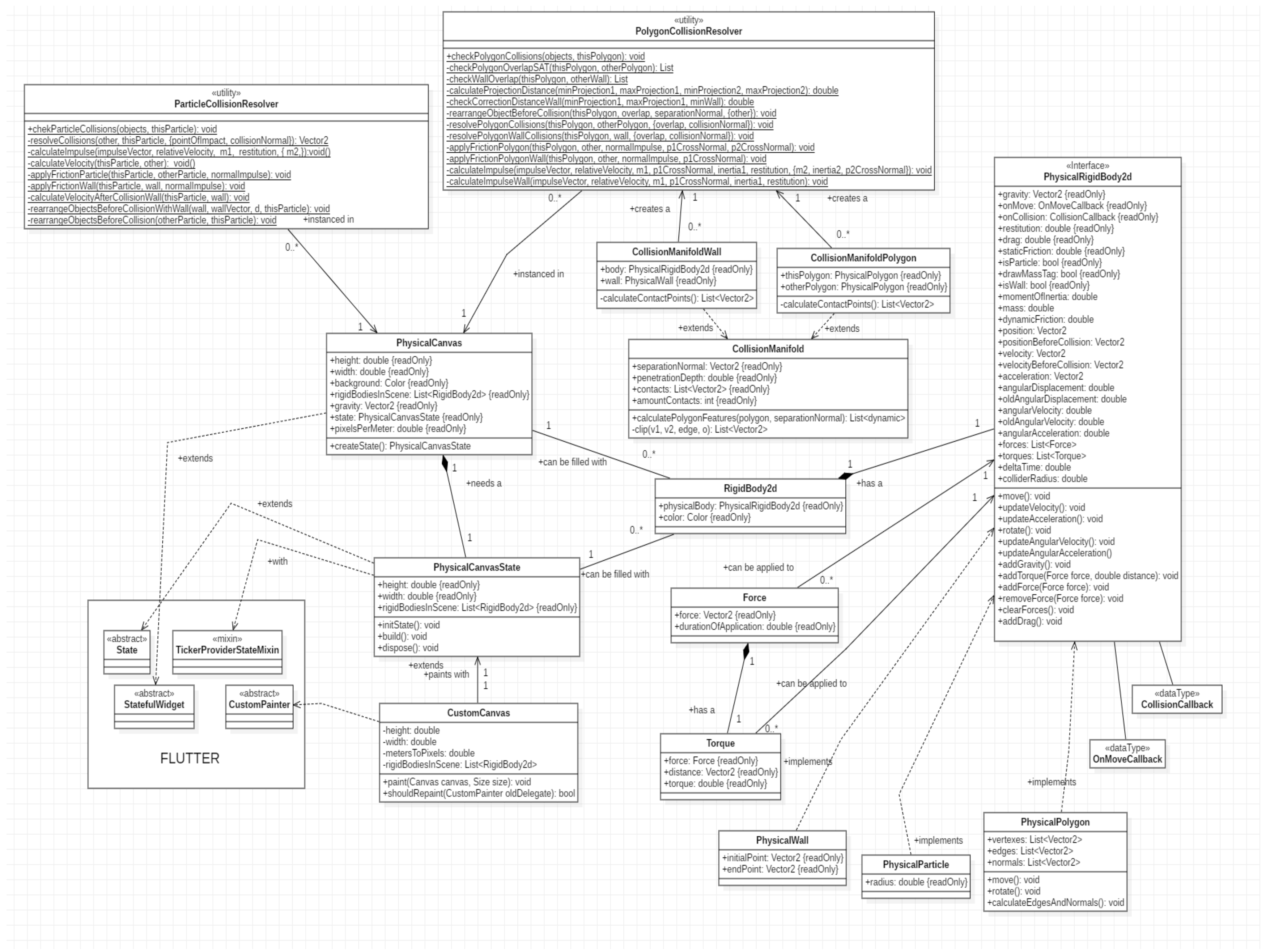
20. Serway, Raymond A.; Jewett, John W. Jr. (2014). *Physics for Scientists and Engineers with Modern Physics – Ninth Edition* [PDF File].

21. Sutherland, Ivan E.; Hodgman, Gary W. (1974). Reentrant Polygon Clipping. In Communications of the ACM Volume 17 Number 1 (1974).

22. Weisstein, Eric W. (n.d). Area Moment of Inertia. Retrieved from http://mathworld.wolfram.com/AreaMomentofInertia.html.

APPENDIX

A1: Complete class diagram of the developed physics API.



«utility»
PolygonCollisionResolver
+checkPolygonCollisions(objects, thisPolygon): void
-checkPolygonOverlapSAT(thisPolygon, otherPolygon): List
-checkWallOverlap(thisPolygon, otherWall): List
-calculateProjectionDistance(minProjection1, maxProjection1, minProjection2, maxProjection2): double
-checkCorrectionDistanceWall(minProjection1, maxProjection1, minWall): double
-rearrangeObjectBeforeCollision(thisPolygon, overlap, separationNormal, {other}): void
-resolvePolygonCollisions(thisPolygon, otherPolygon, {overlap, collisionNormal}): void
-resolvePolygonWallCollisions(thisPolygon, wall, {overlap, collisionNormal}): void
-applyFrictionPolygon(thisPolygon, other, normalImpulse, p1CrossNormal, p2CrossNormal): void
-applyFrictionPolygonWall(thisPolygon, other, normalImpulse, p1CrossNormal): void
-calculateImpulse(impulseVector, relativeVelocity, m1, p1CrossNormal, inertia1, restitution, {m2, inertia2, p2CrossNormal}): void
-calculateImpulseWall(impulseVector, relativeVelocity, m1, p1CrossNormal, inertia1, restitution): void
«utility»
ParticleCollisionResolver
+chekParticleCollisions(objects, thisParticle): void
-resolveCollisions(other, thisParticle, {pointOfImpact, collisionNormal}): Vector2
-calculateImpulse(impulseVector, relativeVelocity, m1, restitution, { m2,}):void()
-calculateVelocity(thisParticle, other): void()
-applyFrictionParticle(thisParticle, otherParticle, normalImpulse): void
-applyFrictionWall(thisParticle, wall, normalImpulse): void
-calculateVelocityAfterCollisionWall(thisParticle, wall): void
-rearrangeObjectsBeforeCollisionWithWall(wall, wallVector, d, thisParticle): void
-rearrangeObjectsBeforeCollision(otherParticle, thisParticle): void
+instanced in
0..*
1
+instanced in
0..*
1
+creates a
1
0..*
+creates a
1
0..*
CollisionManifoldWall
+body: PhysicalRigidBody2d {readOnly}
+wall: PhysicalWall {readOnly}
-calculateContactPoints(): List<Vector2>
CollisionManifoldPolygon
+thisPolygon: PhysicalPolygon {readOnly}
+otherPolygon: PhysicalPolygon {readOnly}
-calculateContactPoints(): List<Vector2>
+extends
+extends
CollisionManifold
+separationNormal: Vector2 {readOnly}
+penetrationDepth: double {readOnly}
+contacts: List<Vector2> {readOnly}
+amountContacts: int {readOnly}
+calculatePolygonFeatures(polygon, separationNormal): List<dynamic>
-clip(v1, v2, edge, o): List<Vector2>
PhysicalCanvas
+height: double {readOnly}
+width: double {readOnly}
+background: Color {readOnly}
+rigidBodiesInScene: List<RigidBody2d> {readOnly}
+gravity: Vector2 {readOnly}
+state: PhysicalCanvasState {readOnly}
+pixelsPerMeter: double {readOnly}
+createState(): PhysicalCanvasState
1
+can be filled with
0..*
1
+needs a
1
RigidBody2d
+physicalBody: PhysicalRigidBody2d {readOnly}
+color: Color {readOnly}
1
+has a
1
0..*
1
+can be filled with
PhysicalCanvasState
+height: double {readOnly}
+width: double {readOnly}
+rigidBodiesInScene: List<RigidBody2d> {readOnly}
+initState(): void
+build(): void
+dispose(): void
+extends
+paints with
1
1
CustomCanvas
-height: double
-width: double
-metersToPixels: double
-rigidBodiesInScene: List<RigidBody2d>
+paint(Canvas canvas, Size size): void
+shouldRepaint(CustomPainter oldDelegate): bool
+extends
+extends
+with
«abstract»
State
«mixin»
TickerProviderStateMixin
«abstract»
StatefulWidget
«abstract»
CustomPainter
FLUTTER
«Interface»
PhysicalRigidBody2d
+gravity: Vector2 {readOnly}
+onMove: OnMoveCallback {readOnly}
+onCollision: CollisionCallback {readOnly}
+restitution: double {readOnly}
+drag: double {readOnly}
+staticFriction: double {readOnly}
+isParticle: bool {readOnly}
+drawMassTag: bool {readOnly}
+isWall: bool {readOnly}
+momentOfInertia: double
+mass: double
+dynamicFriction: double
+position: Vector2
+positionBeforeCollision: Vector2
+velocity: Vector2
+velocityBeforeCollision: Vector2
+acceleration: Vector2
+angularDisplacement: double
+oldAngularDisplacement: double
+angularVelocity: double
+oldAngularVelocity: double
+angularAcceleration: double
+forces: List<Force>
+torques: List<Torque>
+deltaTime: double
+colliderRadius: double
+move(): void
+updateVelocity(): void
+updateAcceleration(): void
+rotate(): void
+updateAngularVelocity(): void
+updateAngularAcceleration()
+addGravity(): void
+addTorque(Force force, double distance): void
+addForce(Force force): void
+removeForce(Force force): void
+clearForces(): void
+addDrag(): void
+can be applied to
0..*
1
Force
+force: Vector2 {readOnly}
+durationOfApplication: double {readOnly}
1
+has a
1
+can be applied to
0..*
1
Torque
+force: Force {readOnly}
+distance: Vector2 {readOnly}
+torque: double {readOnly}
+implements
PhysicalWall
+initialPoint: Vector2 {readOnly}
+endPoint: Vector2 {readOnly}
+implements
PhysicalParticle
+radius: double {readOnly}
+implements
PhysicalPolygon
+vertexes: List<Vector2>
+edges: List<Vector2>
+normals: List<Vector2>
+move(): void
+rotate(): void
+calculateEdgesAndNormals(): void
«dataType»
CollisionCallback
«dataType»
OnMoveCallback

HAW
HAMBURG

Affidavit

I assure that I have written the present work independently without outside help and that I have not used any other sources or aids than those indicated. The passages taken literally from other works, or passages borrowed from the sense of the word, are clearly indicated by references.

Hamburg, 26th February 2020

Simón Hoyos Cadavid