Simplified Mass Spring Model for Deformation Effects in Real Time

Benedict Dietrich

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2015

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, September 28, 2015

Benedict Dietrich

Contents

Abstract

Soft body simulation (also called soft body dynamics) is a field in computer graphics that deals with the deformation of objects. It is an essential part in the animation industry to create more realistic and believable effects. Over the last years it became possible to run soft body simulations in real time. This is because of less calculation intensive models and more powerful hardware. Soft body simulation was also used more and more in computer games, first for effect simulation, that only served to improve the look and feel of the computer game, then also for gameplay changing effects while playing.

The main focus of this master's thesis is the implementation of a soft body simulation that works in real time and could be used in a computer game. The thesis provides an overview of known soft body models, explains important parts of a game engine, a detailed look in the implementation and test results for the implementation.

Kurzfassung

Soft-Body Simulation (auch bekannt als Soft-Body Dynamics) kommt aus dem Feld der Computergrafik und beschäftigt sich mit der Deformation von Objekten. Sie ist ein essenzieller Teil in der animations Industrie um realistische und glaubhafte Effekte zu erzielen. Über die letzten Jahre wurde es möglich diese in Echt-Zeit zu simulieren. Der ausschlaggebende Grund für das waren neue, weniger rechenaufwendigere Modelle und stärkere Hardware. Soft-Body Simulation wurde auch vermehrt in Computerspielen verwendet, am Anfang nur zur Darstellung von Effekten, welche das Computerspiel optisch verbessert haben, dann aber auch für Effekte, welche die Spielmechanik des Computerspiels zur Laufzeit, verändert haben.

Der Hauptfokus der Masterarbeit liegt in der Implementierung einer Soft-Body Simulation, die in Echt-Zeit funktioniert und in einem Computerspiel verwendet werden kann. Die Arbeit gibt einen Überblick der bekannten Modelle für Soft-Body Simulation, erklärt die wichtigsten Teile einer Spiele-Engine (engl. game engine) und gibt einen detaillieren Blick in die Implementation und Testresultate.

Chapter 1

Introduction

There are organic and non-organic materials that can change their shape over time because of material behavior or an external force. Soft body simulation (also known as soft body dynamics or deformable objects simulation) is a field in computer graphics that deals with the motion and properties of deformable objects [12].

With the evolution of computer hardware, the development of newer models and more efficient methods soft body simulation has become more and more interesting. It is used in the fields of computer animation, character animation, computer games and surgical training [12][7, p. 6]. In the field of computer animation it helps by simulation materials (fluids, human tissue, fabrics;) and effects (melting, bending, squashing;) that are hard to animate by hand. This also helps in the field of character animation by simulation skin, hair and clothing. In computer games it can immerse the player more into the game by providing realistic effects during playing [7, p. 6]. This master's thesis focuses on the field of computer games and how soft body simulation can be achieved in computer games.

1.1 Master's thesis and project target

The target of the master's thesis is to list current models and methods that are practical for soft body simulation, evaluates their strengths and weaknesses and decides based on these what model fits best for computer game. The thesis should offer the reader an easy and informative introduction to the topic. The project to this thesis implements the findings and provides additional results.

1.2 Personal motivation

The author of this master thesis has a strong interest in computer games and all the techniques and components that are available to create and improve

computer games. Soft body simulation is a topic that establishes itself in the area of physics simulation and became a fix part of physic engines like Havok [13] and Bullet [14]. Soft body simulation is more and more seen and experienced in computer games today. Surprisingly not only used in AAA computer games [1], but also in independent computer games [2]. This raised the interest of the author to get into the topic of soft body simulation. The thesis should offer an easy, yet sufficient introduction to the topic of soft body simulation for any interested reader.

[1] It is pronounced as triple A and is a term to describe games with a high development and promotion budget.

[2] It is commonly referred to as indie games and it stands for games without support from a publisher.

Chapter 2

State of the art

The research for modeling deformable objects is growing for over the last 40 years [7, p. 10]. The following chapter provides an overview of the methods and models, that are practical for soft body simulation. It gives a short explanation of each model and lists their strengths and weaknesses. After that, this master thesis lists important aspects of computer games and game development. It provides an explanation of this aspects to clear up any misunderstandings, that might occur in later chapters. In the end it provides a few examples of computer games, that use soft body simulation to improve their look and feel. Soft body deformation itself separates in:

- *Elastic deformation*: The shape of the object deforms, when a force is applied and returns to the original shape, when the force is removed [7, p. 2].
- *Plastic deformation*: The shape of the object deforms, when a force is applied and partially returns to its original shape, when the force is removed [7, p. 3].
- *Fracture deformation*: The shape of the object deforms, in irreversible ways (breaking, shattering, torn off;) and does not return into its original shape, once the force is removed [7, p. 3].

The master's thesis focuses on elastic deformations.

2.1 Deformation models

The following section reviews existing models that qualify for soft body simulation. Each of these models are based on physical laws. This section is the base for the selection process in the beginning of section 4.1.

2.1.1 Finite element method (FEM)

The FEM models the deformable as a continuous connected volume of solid elements (see Fig. 2.1). The higher the number of the solid elements, the

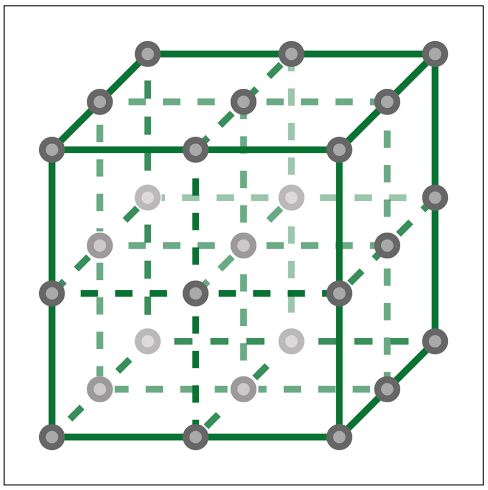


Figure 2.1: Simple model showing a genera setup of a finite element model. The gray circles are the solid elements and the green lines and dashed lines are the connection between the solid elements. The dashed lines show hidden edges and internal connections [6].

more realistic and accurate is the simulation. If an external force is applied to the object, the energy spread over these solid bodies through the whole object [6][7, p. 12].

Advantages

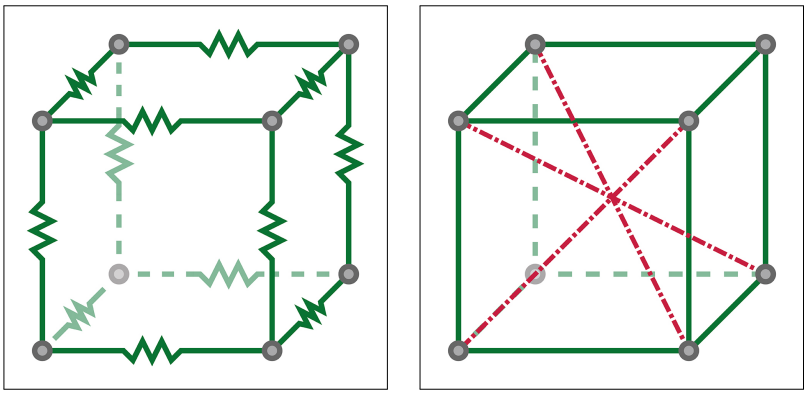
It is the most accurate physical model and it gives the most realistic deformation result of all the methods and models. With this method it is possible to simulate large elastic deformations like melting.

Disadvantages

The biggest disadvantage of this method, is the lack of efficiency. The method is only efficient for elastic deformations which stay within a small range. The calculation for larger deformations is a heavy processing task and isn't efficient enough to show in real time [7, p. 12].

2.1.2 Mass spring system

This method requires a discrete model of the object. It generates deformable objects out of the discrete model by using mass points (also referred to as nodes or particles) and elastic springs that connect them (see Fig. 2.2 (a)). These springs obey some variant of Hooke's law [12, 6, 7]. Compared to the finite element methods, the deformable object is hollow and can collapse



(a) (b)

Figure 2.2: The image on the left (a) shows the spring (green lines and green dashed lines) and nodes (gray circles) that make up the surface of the object. The image on the right (b) shows the structural spring (red dot-dash line) that are inside the object to keep the object from collapsing into itself [6].

into itself. It adds internal structural springs, in order to prevent it from collapsing (see Fig. 2.2 (b)).

Advantages

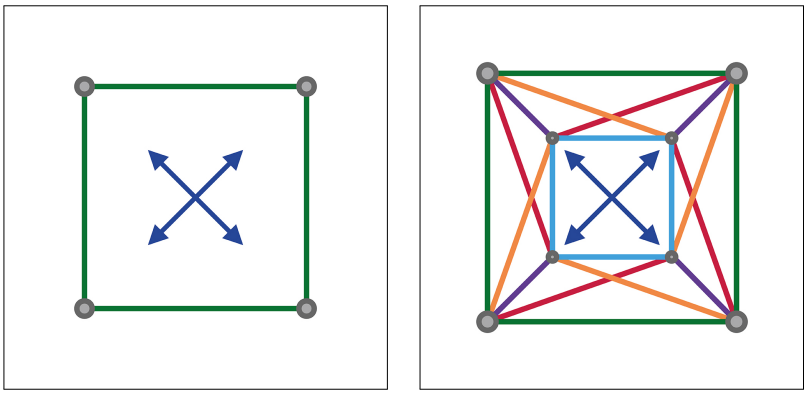
This mass spring system is easy to construct, display and understand. The individual springs of a model can have different parameters to display different kinds of behavior. It is also a widely used model in computer games.

Disadvantages

It requires additional springs to prevent the object from collapsing. The number of these additional springs grows with the complexity of the discrete model. For objects and materials that require a large deformation it isn't practical (effects like melting and different fluids) [7, p. 11].

2.1.3 Pressure model

This model is similar to the mass spring model. The deformable object is build out of a discrete model by using mass points and springs to connect them. The model gets rid of the internal structural springs by replacing them with an internal pressure. This pressure acts from the center of the object on all the mass points and keeps the object from collapsing (see Fig. 2.3 (a)). Characteristically for this model is, the deformable objects behave similar to a balloon [7, p. 13][4].



(a) (b)

Figure 2.3: The image on the left (a) shows the pressure force (blue arrows) inside of the object. The surface is similar to the surface of the mass spring model (see Fig. 2.2 (a)) [4]. The image on the right (b) shows the combined model. The orange and red lines are springs, that prevent sheering. The purple lines are springs that connect the center of the soft body with the surface of the soft body. The blue lines in the center are the soft body the pressure force applies on [7, p. 36].

Advantages

The pressure model does not require additional structural springs and provides very convincing results for gas filled objects.

Disadvantages

It makes every object appear bloated and gives it a balloon like look. The usage is limited to gas filled objects. The model requires an inside volume and an enclosed surface to work [7, p. 13].

2.1.4 Combined model

This model is a combination of the mass spring system and the pressure model. The combined model has a core that acts like the pressure model. This core is enclosed by an additional layer that is connected to the core with structural springs like in the mass spring model (see Fig. 2.3 (b)).

Advantages

The model acts less like the pressure model and the additional layer of springs helps to give the model less of a balloon like behavior.

Disadvantages

Additional structural springs are required to prevent sheering between the pressure core and the shell. The balloon like behavior is not entirely gone [7, p. 16].

2.2 Computer games

A computer game is a real-time dynamic interactive simulation. This section provides an overview of all the essential aspects, that are important for game development and physics simulation.

2.2.1 Game engine

A game engine is a software framework for the development and creation of video games. Its architecture strongly depends on the scale and the genre of the game it is used for. There is no fixed set of rules how a game engine should look like, but it has become an industry standard, to build it in different layers. To prevent circular dependencies, the upper layers depend on the lower layers and not vise versa. This speeds up the development of games, allows to quickly and easily include new or replace old software libraries. It also makes cross platform development easier. If a physics engine is used in the game, it's only a part of this framework[1]. There are a staggering amount of game engines available on the web, both open source or free to use for non commercial projects [2, p. 11].

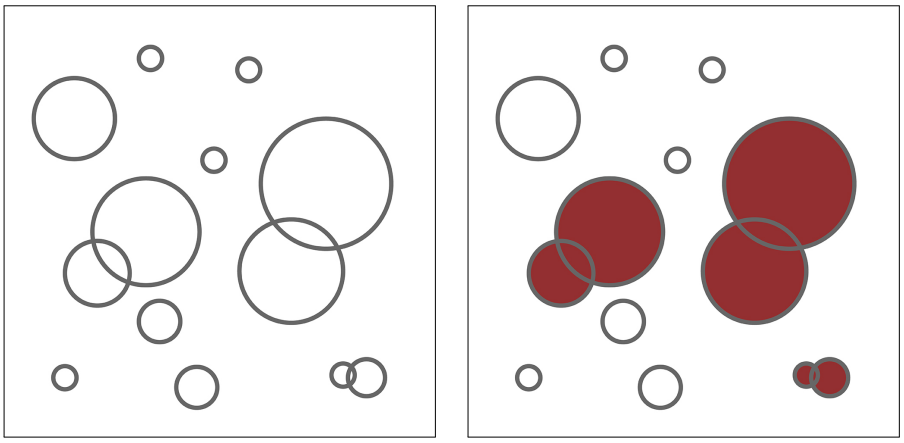
2.2.2 Render engine

The render engine is usually the largest and most complicated part of any game engine. With the render engine the visual scene of the game is created. There are multiple ways and approaches to render the scene and there can also be different render engines for different platforms [2, p. 40].

2.2.3 Frames per second (FPS)

The number of individual frames (images), a device is able to calculate and render on the screen per second. For computer games the frames per second are an important part for the immersion into the game. For computer games, the frames per second should be usually between 30 or 60. A constant framework above 60 ensures smooth animations and no flickering on the screen. A deformation simulation, that is used in a video games needs to be able to deliver a smooth result with these FPS [2, p. 348][15].

[1] For a detailed overview of a game engine architecture see [2, p. 13].



(a) (b)

Figure 2.4: The image on the left (a) shows all the objects in the scenario, some of them are colliding and some of them are not. The image on the right (b) shows the same scenario after the broad phase. The colliding pairs have been identified [1, p. 15].

2.2.4 Timestep

The timestep is the time, that passes during a single update cycle of the game engine. Different elements in a computer game uses this timestep to update themself. This elements can be animations, movement, effects or physics calculations. This timestep can either be variable or set to a fixed value. An advantage of using a variable time step is that it looks smoother but it can cause problems with the game physics, when the timestep gets very small or very big. To counteract this problem the game physic can run on a fixed timestep, while other parts of the game run on a variable timestep. If the timestep for the game physics is variable it needs a limit on how big the timestep can get in order to ensure the simulation does not get out of control [2, p. 348][16, 17].

2.2.5 Interpolation

Through interpolation new data points get calculated out of existing ones. With smaller steps in between two points elements like animation, movement and various other things can run smoother and more precise [2, p. 181]. The section 3.3 provides an overview of the different interpolation methods and compares them.

2.2.6 Collision detection

The task of the collision detection (often refereed to as "physics" in the game development community [2, p. 37]), is to detect colliding objects during the simulation and prevent these from intersecting. Since computer games run in real time, the collision detection needs to be fast and efficient. Each object in the simulation can potentially collide with every other object. This task becomes a problem when the number of objects increases. To make the collision detection more efficient, it is split into two different phases:

- Broad phase,
- Narrow phase.

The broad phase goes through all the objects in the simulation and identifies all object pairs that collide. In order to go through all the objects in the simulation at a decent speed, it needs an efficient algorithm. These colliding object pairs are marked for the narrow phase (see Fig. 2.4). The narrow phase just pays attention to the previously marked object pairs. This phase can use less efficient algorithms because the number of objects has been reduced [1, p. 14].

Collision detection libraries

There are a few well-known collision detection engines:

- *Havok* is the most known industrial-strength physics engine on the market. It has been used in over 400 computer games and the developer claims that the engine has become the gold standard in the game industry [13].
- *PhysX* is another well-known industrial-strength physics engine that is available for free from NVIDIA. Game engines like Unity and Unreal use it as their default physic engine [18, 19].
- *Bullet* is a powerful, free and open-source physics engine. Game companies as well as movie companies use this engine [14, 20].

2.3 Game examples

Soft body simulation can be used in different ways to display different effects. The following games use deformation as a gameplay element or to improve the visual appeal.

2.3.1 Agar.io

Agar.io [11] is a massive multiplayer action game, developed by Matheus Valadares. Players control a single cell represented by a colored circle and can move around in a petri dish like environment. The main objective is to

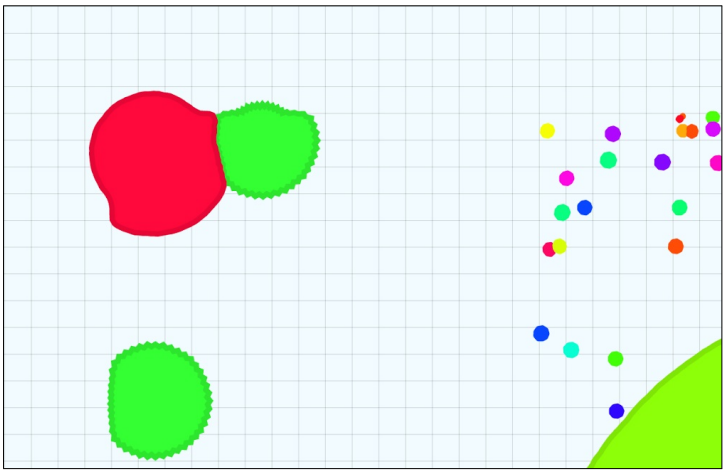


Figure 2.5: The image shows the player (red sphere) pushed into the corner of the level and an object in the level (green spiced sphere) [21].

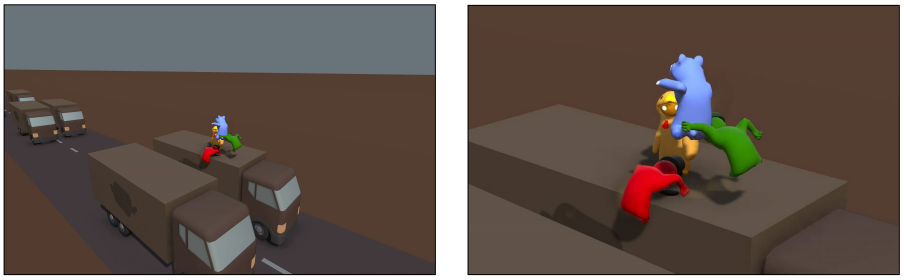
grow bigger by consuming smaller cells that can be also controlled by other players. The deformation simulation is used to show the players which cell is consuming the other, warn players if they are about to hurt themself and it also shows the borders of the level by deforming the circle the players control [21, 22].

2.3.2 Gang Beasts

Gang Beasts [9] is a local multiplayer beat'em-up, developed by Boneloaf and published by Double Fine Production. Players control a human shaped character that looks and feels like it is made out of gelatine or plasticine. The goal of the game is to throw all the other players out of the fighting arena. This game shows how deformation simulation can be combined with a ragdoll-like character [23, 24].

2.3.3 Wreckfest

Wreckfest [10] is a demolition derby-like racing game, developed and published by Bugbear Entertainment. Players control various models of cars and compete in different events against each other. The gameplay does not differ from other racing games, except, that it is very realistic in both look and feel. One of the unique and astonishing feature the game is the soft body damage modeling. Collisions and the deformations are calculated and applied in real-time [25, 26].



(a) (b)

Figure 2.6: The images show the player character and different levels of the game [23].

(a) (b)

Figure 2.7: The image on the left (a) shows a car model that is used on the game and the image on the right (b) shows a crash that can happen in the game [25].

2.4 Summary

The beginning of the chapter lists the potential deformable models with their advantages and disadvantages. This includes the finite element method, mass spring system, pressure model and combined model. After that, the focus shifts to computer games and explains important aspects of these to clear up any misunderstandings. The end of the chapter gives an overview over a few games that include soft body deformation.

Chapter 3

Calculations

The focus of this chapter is on the formulas and calculations for the implementation.

3.1 Newton's laws

Newton's laws consist of three physical laws, that form the foundation for mechanics. The rule, that is interesting for this topic is the second one:

> The vector sum of the forces F on an object is equal to the mass m of that object multiplied by the acceleration vector $\mathbf{a}$ of the object: $\mathbf{F} = m\,\mathbf{a}$ [27].

In general the force F is defined as

$$\mathbf{F} = m\,\mathbf{a}. \tag{3.1}$$

The force $\mathbf{F}$ and the mass of the object m are known. The acceleration a is currently unknown. The velocity $\mathbf{v}$ is defined as

$$\mathbf{v} = \int \mathbf{a}\; dt. \tag{3.2}$$

This equation can be solved for $\mathbf{a}$ and inserted into equation 3.1

$$\mathbf{F} = m\frac{d\mathbf{v}}{dt}. \tag{3.3}$$

The calculation is put into the numerical number space in order to get rid of the integral, turning $\mathbf{F}$ into

$$\mathbf{F} = m\frac{\Delta\mathbf{v}(t)}{\Delta t}. \tag{3.4}$$

This removes the infinity from the integral and makes it easier to calculate. The velocity $\mathbf{v}$ is then used to calculate the position $\mathbf{p}$ with respect to the time dt

$$\mathbf{p} = \int \mathbf{v}\, dt, \tag{3.5}$$

$$\mathbf{p} = \mathbf{v}(t)\, \Delta t. \tag{3.6}$$

This equations are the basic for the following calculations, integrators and the deformation simulation itself [5, p. 10][7, p. 51][27].

3.2 Forces in the simulation

During the simulation the objects experience different forces. $\mathbf{F}$ combines the gravity force $\mathbf{F}_{GR}$, the forces building up in the spring $\mathbf{F}_{SP}$ and external forces $\mathbf{F}_{EX}$ by adding these up

$$\mathbf{F} = \mathbf{F}_{GR} + \mathbf{F}_{SP} + \mathbf{F}_{EX}. \tag{3.7}$$

3.2.1 Gravity force

The gravity force $\mathbf{F}_{GR}$ is a constant force that pulls objects towards the earth based on their mass m by g [7, p. 16]

$$\mathbf{F}_{GR} = m\,\mathbf{g}. \tag{3.8}$$

F_G is calculated by multiplying the mass of the object m with the the currently acting gravity $\mathbf{g}$. $\mathbf{g}$ is the gravitational strength g (on earth $9.81\mathrm{m}/(s)^2$) along a direction $\mathbf{e}$ [5, p. 14]

$$\mathbf{g} = g\,\mathbf{e}. \tag{3.9}$$

In video games it is typical that $\mathbf{F}_G$ is treated as a constant and the objects in the simulation are affected equally regardless of their mass [2, p. 719].

3.2.2 Spring

The springs obey Hooke's Law and $\mathbf{F}_{SP}$ is a linear force that acts upon the two connected mass points (also called node or particle) n_1 and n_2 (see Fig. 3.1). The nodes can act upon the spring by stretching or extending the spring. For easier understanding this is split into two sections, the calculation of the spring force $\mathbf{F}_S$ and the calculation of the damping force $\mathbf{F}_D$ [7, p. 32]

$$\mathbf{F}_{SP} = \mathbf{F}_S + \mathbf{F}_D. \tag{3.10}$$

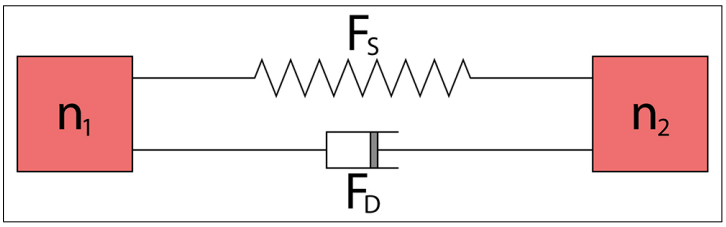


Figure 3.1: The image shows a schematic representation of a spring that connects to mass points n_1 and n_2. The spring force F_S and spring damping force F_D act in between the two mass points [12].

Spring force

The spring force $\mathbf{F}_S$ is calculated by

$$n_{12} = n_2 - n_1 \tag{3.11}$$

$$\mathbf{F}_S = (||n_{12}|| - l)\ k_e\ \mathbf{e}_{12}. \tag{3.12}$$

n_1 and n_2 are the positions of the two associated mass points of the spring. l is the resting length of the spring. The spring tries to maintain this length. n_{12} is the current length of the spring. k_e is the elasticity factor of the spring, it is a constant factor that states the springs characteristics. $\mathbf{e}_{12}$ is the directional vector from n_1 to n_2. $\mathbf{e}_{12}$ defines the direction the spring is able to act in. $\mathbf{F}_S$ acts on n_1 and on n_2 in the opposite direction

$$n_{12} = n_2 - n_1 \tag{3.13}$$

$$\mathbf{F}_{S1} = \mathbf{F}_S \tag{3.14}$$

$$\mathbf{F}_{S2} = -\mathbf{F}_S. \tag{3.15}$$

$\mathbf{F}_{S1}$ acts on n_1 and $\mathbf{F}_{S2}$ acts on n_2. The spring can have the following states [7, p. 33]

$$||n_{12}|| - l \begin{cases} = 0 & \texttt{spring resting}, \\ > 0 & \texttt{spring extended}, \\ < 0 & \texttt{spring contracted}. \end{cases}$$

Damping force

The damping force resists the motion and give the spring a natural moving behavior [7, p. 33]. The damping force $\mathbf{F}_D$ is calculated with

$$\mathbf{F}_D = (\mathbf{v}_2 - \mathbf{v}_1) \cdot \left(\frac{n_{12}}{||n_{12}||} \right) k_d. \tag{3.16}$$

$\mathbf{v}_1$ and $\mathbf{v}_2$ are the velocities acting in the two associated mass points of the spring. k_d is the damping coefficient of the spring [7, p. 34].

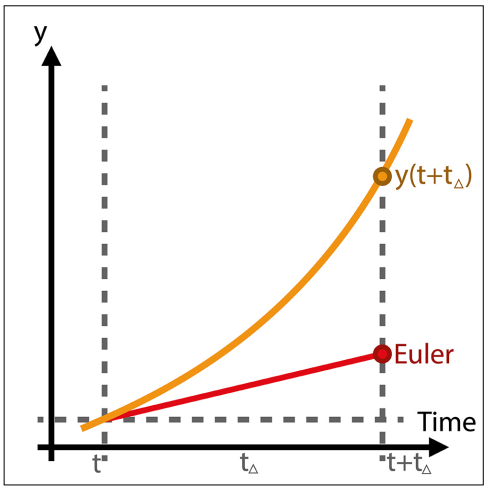


Figure 3.2: Graphic showing the Euler integrator compared to the example function y [7, p. 46].

3.2.3 External forces

The external forces $\mathbf{F}_E$ are additional forces that are applied to the object.

3.3 Integrators

Integrators take care of the interpolation (see chap. 2.2.5). This section shows the different iterators, that are used in the implementation. To solve this problem, an ordinary differential equation (ODE) is used. An ODE contains one independent variable and its derivatives. This type of equations are used to describe, how a dynamic system behaves over time, when given an initial state [28][7, p. 44]. To illustrate the advantages and disadvantages of each method, a simple example is constructed with a single time step. In the example, an arbitrary function $y(x)$ is used. The range of this function starts at time step t and ends at time step dt. The variable t_{delta} is the time that has passed between this two points t and dt. The result of the function $y(x)$ is a curve and each of the following methods has a different level of approximation to this curve.

3.3.1 Euler integration

The Euler calculation method is the simplest and fastest integrator, but also the method with the biggest deviation (see Fig. 3.2). The velocity $\mathbf{v}$ is represented in the equation

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \Delta t\ \mathbf{v}'(t). \tag{3.17}$$

$\mathbf{v}_0$ represents the initial velocity at time step t

$$\mathbf{v}_0 = \mathbf{v}(t). \tag{3.18}$$

At time step t the initial velocity is known. $\mathbf{v}_1$ represents the derivative at time step dt

$$\mathbf{v}_1 = \Delta t\ \mathbf{v}'(t). \tag{3.19}$$

$\mathbf{v}_1$ is calculated with the acceleration $\mathbf{a}$

$$\mathbf{v}_1 = \mathbf{a}(t)\ \Delta t. \tag{3.20}$$

The acceleration $\mathbf{a}$ calculation with the equation 3.1 and solve it for $\mathbf{a}$

$$\mathbf{a} = \frac{\mathbf{F}}{m}. \tag{3.21}$$

The position $\mathbf{p}$ is represented in the equation

$$\mathbf{p}(t + \Delta t) = \mathbf{p}(t) + \Delta t\ \mathbf{p}'(t). \tag{3.22}$$

The equation is again split in two parts

$$\mathbf{p}_0 = \mathbf{p}(t), \tag{3.23}$$
$$\mathbf{p}_1 = \Delta t\ \mathbf{p}'(t). \tag{3.24}$$

$\mathbf{p}_0$ is the initial position and $\mathbf{p}_1$ is calculate with the velocity

$$\mathbf{p}_1 = \mathbf{v}(t)\ \Delta t. \tag{3.25}$$

The result of the Euler integrator shows a big deviation to the actual result of the example curve (see Fig. 3.2) [7, p. 46].

3.3.2 Midpoint integration

The midpoint integration is a symmetric estimate method with a higher per-step accuracy. The method calculates the derivative in the center of the interval at $dt/2$ and at the end of the interval at time step dt. The midpoint method includes the slope at $dt/2$ resulting in a lower deviation at time step dt (see Fig. 3.3). The velocity $\mathbf{v}$ is represented with the equation

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \Delta t\ \mathbf{v}'(t) + \frac{\Delta t^2}{2!}\mathbf{v}''(t). \tag{3.26}$$

Like with the Euler integrator, the function of the midpoint integration can be split in different parts

$$\mathbf{v}_0 = \mathbf{v}(t), \tag{3.27}$$
$$\mathbf{v}_1 = \mathbf{v}'(t) = \mathbf{a}(t)\ \Delta t. \tag{3.28}$$

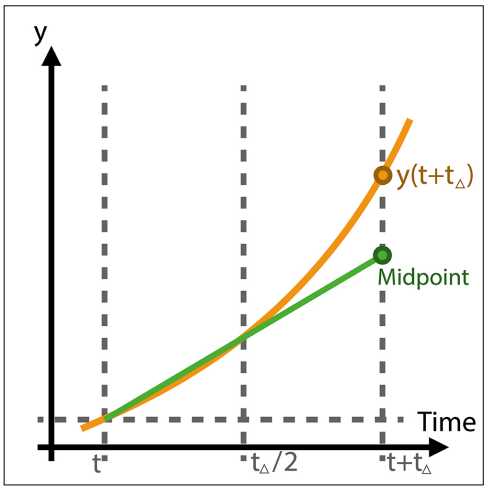


Figure 3.3: Graphic showing the midpoint integrator compared to the example function y. The result at time step dt is closer then it was with the Euler integrator [7, p. 48].

v_0 is the initial velocity at time step t and v_1 is the derivative velocity in the period Δt. Next the velocity at time step $\Delta t/2$ is calculated

$$\mathbf{v}_2 = \mathbf{v}'(t + \Delta t) = \mathbf{v}(t) + \mathbf{a}(t)\ \Delta t. \tag{3.29}$$

With these equations the result for the final velocity is

$$\mathbf{v}(t + \Delta t) = \mathbf{v}_0 + \frac{\mathbf{v}_1 + \mathbf{v}_2}{2}. \tag{3.30}$$

The position $\mathbf{p}$ is calculated by the equation

$$\mathbf{p}(t + \Delta t) = \mathbf{p}(t) + \Delta t\ \mathbf{p}'(t) + \frac{\Delta t^2}{2!}\mathbf{p}''(t). \tag{3.31}$$

The equation is split up in different parts

$$\mathbf{p}_0 = \mathbf{p}(t), \tag{3.32}$$
$$\mathbf{p}_1 = \mathbf{p}'(t) = \mathbf{v}(t)\ \Delta t, \tag{3.33}$$
$$\mathbf{p}_2 = \mathbf{p}'(t + \Delta t) = \mathbf{p}(t) + \mathbf{v}(t)\ \Delta t. \tag{3.34}$$

The position at time step dt is

$$\mathbf{p}(t + \Delta t) = \mathbf{p}_0 + \frac{\mathbf{p}_1 + \mathbf{p}_2}{2}. \tag{3.35}$$

The result of the Midpoint integrator compared to the Euler integrator is more accurate to the actual result, but also the calculation expense has increased (see Fig. 3.3) [7, p. 47].

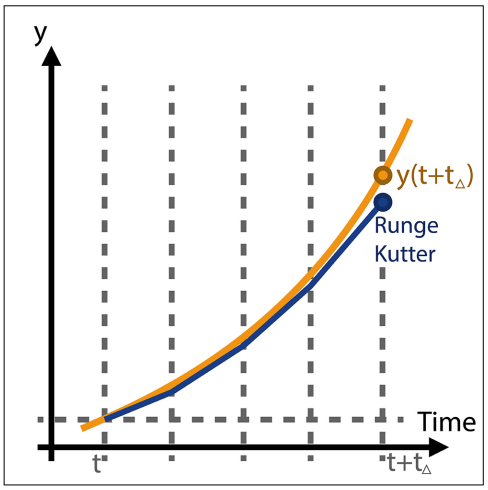


Figure 3.4: Graphic showing the Runge Kutter fourth order integrator compared to the example function y. The result at time step dt is closer then it was with the Euler integrator and in the Midpoint integrator [7, p. 49].

3.3.3 Runge Kutter fourth order integrator

The Runge Kutter fourth order integrator calculates four derivatives and is the most accurate integrator, compared to the Euler and the Midpoint integrator (see Fig. 3.4) [7, p. 49]. The velocity $\mathbf{v}$ is represented with the equation

$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \Delta t\ \mathbf{v}'(t) + \frac{\Delta t^2}{2!}\mathbf{v}''(t) + \frac{\Delta t^3}{3!}\mathbf{v}'''(t) + \frac{\Delta t^4}{4!}\mathbf{v}''''(t). \quad (3.36)$$

The equation is split up in $v_0 \dots v_4$

$$\mathbf{v}_0 = \mathbf{v}(t), \quad (3.37)$$
$$\mathbf{v}_1 = \mathbf{a}(t)\ \Delta t, \quad (3.38)$$
$$\mathbf{v}_2 = \mathbf{v}_0 + \frac{\mathbf{v}_1}{2}, \quad (3.39)$$
$$\mathbf{v}_3 = \mathbf{v}_0 + \frac{\mathbf{v}_2}{2}, \quad (3.40)$$
$$\mathbf{v}_4 = \mathbf{v}_0 + \mathbf{v}_3. \quad (3.41)$$

$\mathbf{v}_0$ is the initial velocity at time step t, $\mathbf{v}_1$ is the derivative velocity in the period dt. $\mathbf{v}_2$ is the derivative of the Euler integration in the period $dt/2$ based on the previous step. $\mathbf{v}_3$ is the derivative velocity of the second approximation based on the $\mathbf{v}_2$ in the period $dt/2$. $\mathbf{v}_4$ is the final resulting velocity change from $\mathbf{v}_0$ to $\mathbf{v}_3$. This makes the new velocity

$$\mathbf{v}(t+\Delta t) = \mathbf{v}_0 + \frac{1}{6}\Delta t\left(\mathbf{v}_1 + 2\mathbf{v}_2 + 2\mathbf{v}_3 + \mathbf{v}_4\right). \quad (3.42)$$

The position $\mathbf{p}$ is calculated with the equation

$$\mathbf{p}(t+\Delta t) = \mathbf{p}(t) + \Delta t\mathbf{p}'(t) + \frac{\Delta t^2}{2!}\mathbf{p}''(t) + \frac{\Delta t^3}{3!}\mathbf{p}'''(t) + \frac{\Delta t^4}{4!}\mathbf{p}''''(t). \quad (3.43)$$

The equation is split up in $\mathbf{p}_0 \dots \mathbf{p}_4$

$$\mathbf{p}_0 = \mathbf{p}(t), \quad (3.44)$$
$$\mathbf{p}_1 = \mathbf{p}(t)\ \Delta t, \quad (3.45)$$
$$\mathbf{p}_2 = \mathbf{p}_0 + \frac{\mathbf{p}_1}{2}, \quad (3.46)$$
$$\mathbf{p}_3 = \mathbf{p}_0 + \frac{\mathbf{p}_2}{2}, \quad (3.47)$$
$$\mathbf{p}_4 = \mathbf{p}_0 + \mathbf{p}_3. \quad (3.48)$$

The calculation for the new position is

$$\mathbf{p}(t+\Delta t) = \mathbf{p}_0 + \frac{1}{6}\Delta t\left(\mathbf{p}_1 + 2\mathbf{p}_2 + 2\mathbf{p}_3 + \mathbf{p}_4\right). \quad (3.49)$$

The Runge Kutter fourth order integrator is the most accurate integrator, but also the most calculation expensive integrator, compared to the two previous integrators (see Fig. 3.4) [7, p. 49].

3.3.4 Position update

The equation for the new position is

$$\mathbf{p}(t+\Delta t) = \mathbf{p}(t) + \Delta t\ \mathbf{p}'(t). \quad (3.50)$$

The equation is split in two parts

$$\mathbf{p}_0 = \mathbf{p}(t), \quad (3.51)$$
$$\mathbf{p}_1 = \Delta t\ \mathbf{p}'(t). \quad (3.52)$$

$\mathbf{p}_0$ is the initial position at the time t and $\mathbf{p}_1$ is calculate with the velocity at the time t and Δt

$$\mathbf{p}_1 = \mathbf{v}(t)\ \Delta t. \quad (3.53)$$

3.3.5 Comparison

All three integrators are compared in the aspect of efficiency and accuracy. For the efficiency, the calculation effort is the main criteria and for the accuracy, the deviation to the example function is the main criteria.

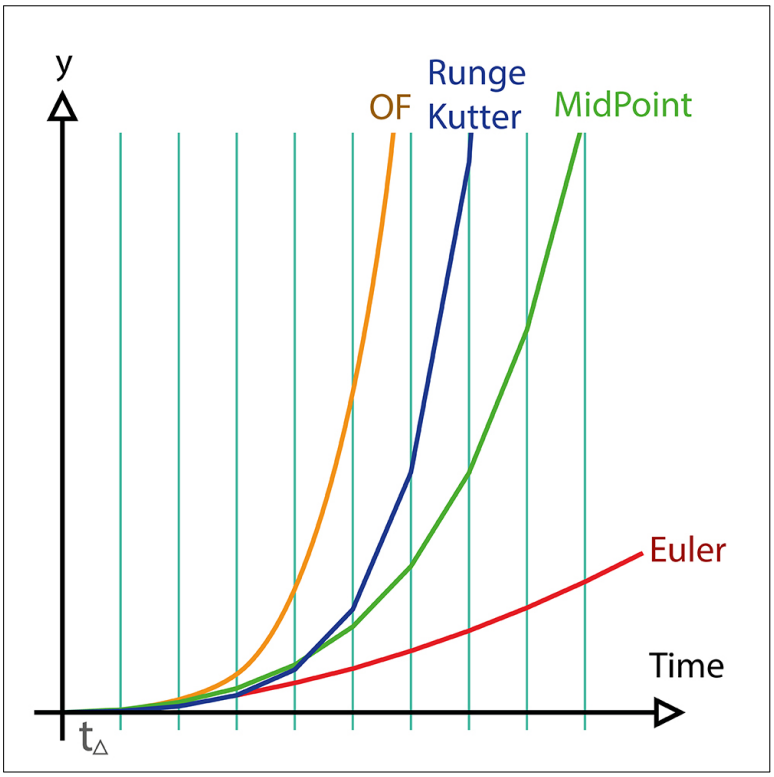


Figure 3.5: The image shows the different integrators compared. The orange line is the original function (OF). The red line is the Euler integration, the green line is the mid point integrator and the blue line is the Runge Kutter fourth order integrator.

Efficiency

The Euler integrator is the most efficient integrator out of the three integrators. It only needs one derivative per step. The midpoint integrator uses two steps to calculate the velocity and position. It requires roughly twice as much computations as the Euler integrator. The Runge Kutter fourth order integrator uses four steps to calculate velocity and position. It requires roughly four times as much computation as the Euler integrator.

Accuracy

The Runge Kutter fourth order integrator is the most accurate integrator out of the three integrators (see Fig. 3.5). It manages to get closest to the actual result.

3.3.6 Conclusion

The choice of integrator method depends on the field it is used in. One criteria are the resources that are available during the simulation and calculation. In a rendering for a computer animation, it may not matter if the calculations take longer, when the result justifies that in the end. In a real time application, like a computer game it does matter. So the integrator is chosen by the resources that are available. If the available resources are minimal the Euler integrator is enough to ensure the application runs in real time, but when more resources are available, a more accurate integrator, like the Midpoint or Runge Kutter integrator offer better results.

3.4 Summary

The main focus of this chapter, is to explain all the calculations, that are required for the implementation. The chapter begins with Newton's law and explains the second law of motion. Then, it lists all the forces that are required. After that, it explains the three different integrators, that handle the interpolation in the implementation. At the end of the chapter, there is a comparison of the three integrators in efficiency and accuracy.

Chapter 4

Implementation and design

This chapter presents the selection process for the deformation model, the algorithms the implementation uses and the structure and components of the implementation itself. In the end, it shows the results of the implementation.

4.1 Deformation model selection

In order for the deformation model to work well in computer games, it needs to fulfill a couple of requirements.

Performance

Important for the model selection is, that the simulation is not resource intensive. Computer games run in real time and nothing is more frustrating and immersion breaking for the player, when the frame rate drops, even if it only lasts for a few seconds. This disqualifies the finite element method. It is the most accurate and realistic model, but also the most expensive one to calculate.

Simplicity

The model should be easy to understand and easy to use. The three remaining models achieve this, because these models share similarities, but the pressure model is the simplest model of them all. It does not require any structural springs, nor does it require a mass spring hull to be modeled around it.

Usability

Computer games have different genres, some are similar and some or completely different. The deformation model should be able to be used for different scenarios. The pressure model delivers the best and most believable

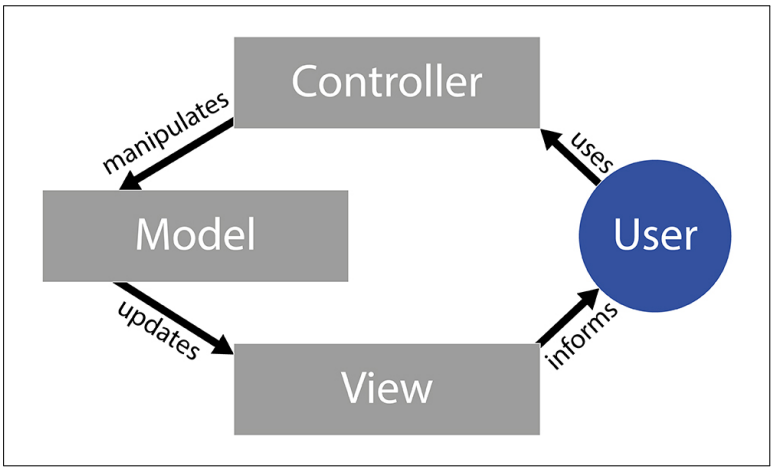


Figure 4.1: The interaction cycle between the MVC and the user. The user enters different inputs to the controller. The controller manipulates the model, based on this inputs. The view changes, based on the manipulations in the model and presents the changes to the user. [29].

result for gas filled objects, but not so good results for other types of objects. The combined model shares some of the same flaws, that the pressure model has, so the decision is in favor of the mass spring system.

Result of the selection

The mass spring system is the favorite, because of the wider range of usability compared to the pressure model and combined model. The mass spring system has its drawbacks with larger deformations, but in the general context of computer games is the most suitable model.

4.2 Design

The implementation uses the Model-View-Controller (MVC) pattern. This pattern is ideal for real time simulations because it splits up the project into three individual parts, the model, the view and the controller. The pattern establishes clear dependencies between the different parts, a clear interaction between the different parts (see Fig. 4.1) and gives each part a clear role:

- The *model* stores all the information, that is needed for the simulation to run. This includes information about the environment the simulation takes place in, information about the boundaries of the simulation and information about each object, that is in this environment.
- The *view* is the on screen representation of the model. It serves as an interface for the user during the simulation. This gives the user

a better understanding, what happens in the simulation and allows interactions with the simulation with the keyboard, mouse and other input devices.
- The *controller* processes the input form the user and manipulates the model, according to these inputs.

In the implementation, the model is the most important part. In the project it contains all of the important parts for the simulation [7, p. 57].

4.3 Used algorithms

This section lists all the algorithms, that the implementation uses.

4.3.1 Explanation of used terms

A short overview of terms, that appear throughout this section, to clear up misunderstandings.

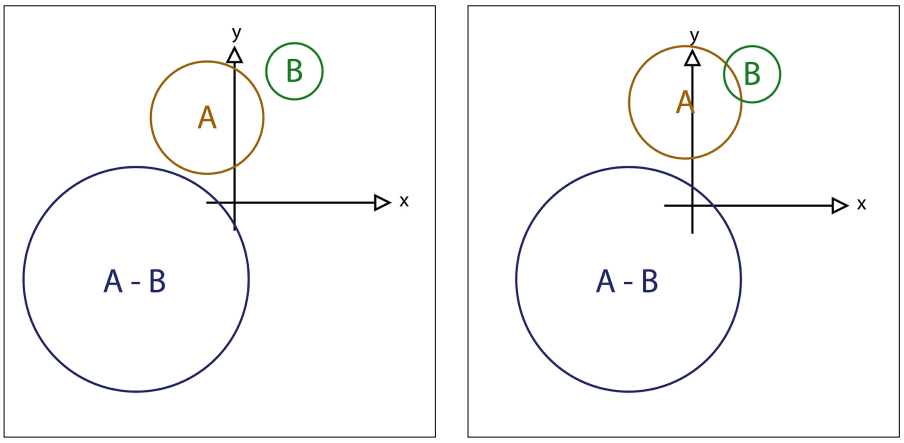
- *Point*: Defines a location in space, for this implementation it is in two dimensional space (also refereed to as 2D space) [2, p. 166]. In general, it is represented by a position *pos* and for this implementation, it can also have a velocity *vel* and a force f.
- *Ray*: It is an infinite line, that is represented by a position *pos* and a direction *dir* [2, p. 213].
- *Line segment*: It is a straight line, bound by two points A and B at both ends [2, p. 214].
- *Plane*: It is a flat surface, that stretches into infinity. It is represented by a position *pos* and a normal *norm*. The normal points in the direction the plane faces [2, p. 215].

4.3.2 GJK algorithm

The GJK algorithm is a simple and very efficient algorithm to determine the shortest distance between two convex sets and if there is a collision between these sets. The algorithm is named after its inventors E. G. Gilbert, D. W. Johnson and S. S. Keerthi[2, p. 670]. There are many papers and presentations about this algorithm, but the most easiest explanation is a blog entry by Casey Murantori that contains an instructional video with the title “Implementing GJK” [30]. The GJK algorithm relies on the geometric operation, called the Minkowski difference. This operation creates a result set S of points from the two original sets A and B. The resulting set contains every point from A subtracted with every point from B

$$S = A - B, \tag{4.1}$$

$$A - B = \{\mathbf{a} - \mathbf{b} \mid \mathbf{a} \in A, \mathbf{b} \in B\}. \tag{4.2}$$



(a) (b)

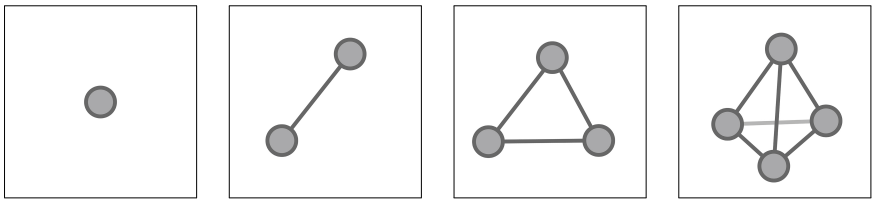
Figure 4.2: Both images contain the two convex sets A and B together with the resulting set $A - B$. The image on the left (a) shows the case when there is no collision (origin is not inside $A - B$) and the image on the right (b) shows the case when there is a collision (origin is inside $A - B$) [2, p. 671].

The most important part of the set of points created by the Minkowski difference is the hull. The interior points of S are uninteresting for the future steps. If the origin of the coordinate system is inside the hull of R, it means that the two set are colliding (see Fig. 4.2(b)). If the origin is outside of this hull, then there is no collision between the two sets (see Fig. 4.2 (a)).

To find out if the origin is inside, an additional shape, called simplex, is build up inside of the hull of S. The simplex starts from a single point on the hull of S (see Fig. 4.3 (a)). It then extends towards the origin by adding another point from the hull of S to itself in the direction d, this creates a line (see Fig. 4.3 (b)). The new point has to be the farthest point of S in d to assure that the algorithm runs as fast as possible. In a 2D scenario the simplex gets up to three points big and tries to enclose the origin with a triangle shape(see Fig. 4.3 (c)). In the 3D scenario the simplex gets up to four points big and tries the same with a tetrahedron shape(see Fig. 4.3 (d)). The algorithm mostly consists of two essential functions:

- Support function,
- Simplex function.

The function 4.1 returns the furthest points from a set in a specified direction. This algorithm can also be used to skip calculating the whole Minkowski difference and instead calculate just the important points. For this the support function 4.2 uses the function 4.1 to get the furthest point from A in the direction d and subtracts it from the furthest point in B in the opposite direction $-d$. This step avoids the storage of the whole Minkowski difference



(a) (b) (c) (d)

Figure 4.3: The different simplex states. The image on the far left (a) shows the initial state of the simplex. It consists of a single point. The second image from the left (b) shows the simplex after a point has been added. The third image from the left (c) shows the simplex after to points have been added to the simplex. In a 2D scenario the simplex does not get bigger. The image on the right (d) shows the simplex after four points have been added. In a 3D scenario the simplex does not get bigger [2, p. 671].

Algorithm 4.1: Returns the furthest point from a set of points in a specified direction by using the dot product.

```
1:  getFurthestPoint(points, d)
    Returns the furthest point from points in the direction d.
2:      p ← points[0]
3:      maxdot ← dot(p, d)                          ▷ calculate dot product
4:      for i ← 1, ..., length(points) do           ▷ iterate over all the points
5:          if (dot(points[i], d)) > maxdot then
6:              p ← points[i]
7:              maxdot ← dot(p, d)
8:          end if
9:      end for
10:     return p
11: end
```

Algorithm 4.2: Calculates the Minkowski difference for a specified direction.

```
1:  supportFunction(A, B, d)
    Returns the Minkowski difference in the direction d for A and B.
2:      return getFurthestPoint(A, d) − getFurthestPoint(B, -d)
3:  end
```

and always returns points, that would be on the hull of S in the direction d. The simplex function determines the search direction and adds points to itself or removes them if required. The simplex expands until it either manages to enclose the origin, which signals it is a positive collision or until it can no longer expand to a new position on the hull, signaling a negative

collision. But even a negative collision provides useful data, the shortest distance between S and the origin is the same as the shortest distance between A and B [1, p. 399][2, p. 670][30].

By picking an intelligent staring point for the GJK algorithm it can speed up, but not for much, because it finds a solution quite fast even with a bad starting point [30].

4.3.3 Plane-point-position test

During the simulation the planes act as borders and keep objects from falling into infinity. This algorithm 4.3 determines if a point p is in front, behind or on a plane b by using the dot product.

Algorithm 4.3: Determines if a point p is in front, on top or behind a plane b [1, p. 207]

```
1: positionPointPlane(p, b)
   Returns 0 if the p is in front of b, 1 if on the b and 2 ifs behind the b.
2:     r ← dot(b.norm, (n.pos + length(n.pos)))
3:     if (r > 0) then
4:         return 0                                  ▷ in front of the plane
5:     else if (r < 0) then
6:         return 2                                  ▷ behind the plane
7:     end if
8:     return 1                                      ▷ on the plane
9: end
```

4.3.4 Plane-Plane-intersection test

This test determines the intersection point between the point p and plane $pl1$. The implementation needs this test when a point is behind the plane and needs to be moved back to the surface of the plane. For the test to work, it converts p into a plane. The position of the new plane $pl2$ is the same position as p. For the normal of $pl2$ the velocity of p is normalized and rotates by 90°. This is the preparation step of the algorithm 4.4, that to determine the intersection point between.

4.3.5 Point-in-triangle test

This test uses the barycentric coordinate system. This coordinate system makes all kind of triangle tests easier by using a parametrized space. Triangles are a common shape in real-time applications because of these reasons:

- Triangles are the simplest type of a polygon with a total of three vertices.

Algorithm 4.4: Returns the intersection point between the two planes $pl1$ and $pl2$. p is a reference to a point and it becomes the intersection point [1, p. 209].

```
1:  intersectionPlanePlane(pl1, pl2, p)
2:      d11 ← dot(pl1.n, pl1.n)
3:      d12 ← dot(pl1.n, pl2.n)
4:      d22 ← dot(pl2.n, pl2.n)
5:      denom ← d11 · d22 − d12 · d12
6:      d1 ← dot(pl1.n, pl1.p)
7:      d2 ← dot(pl2.n, pl2.p)
8:      k1 ← (d1 · d22 − d2 · d12) / denom
9:      k2 ← (d2 · d11 − d1 · d12) / denom
10:     p ← k1 · pl1.n + k2 · pl2.n
11: end
```

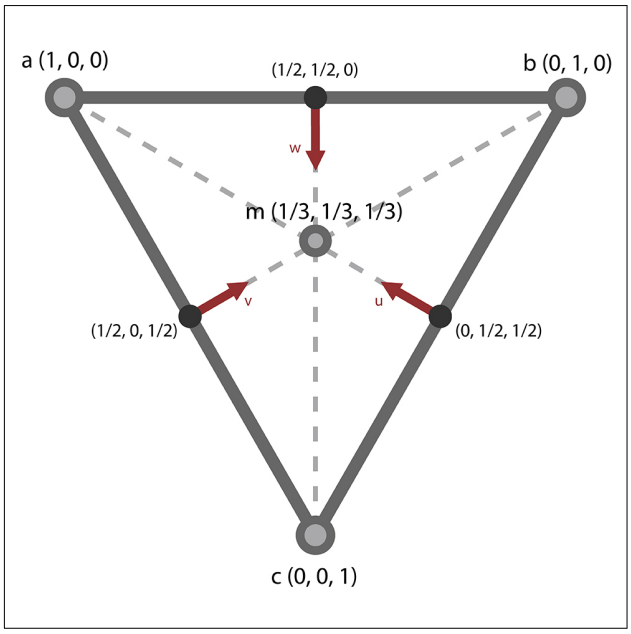


Figure 4.4: The image shows a triangle in the barycentric coordinate system with the corner points a, b and c, the center points m and the different axis u, v and w. The barycentric coordinates for the points in the image are the same for each triangle [31].

- The surface of a triangle is always planar.
- Triangles remain triangles under most transformations.
- Most of the commercial and non commercial render engines are designed around triangles [2, p. 447].

In the case of a triangle all the corner points a, b, c and the center of the triangle m are inside this space (see Fig. 4.4). The test transfers all the points of the triangle and the test point p in the barycentric coordinate system and checks if p is within the corner points of the triangle a, b and c (see Alg. 4.5) [1, p. 46].

Algorithm 4.5: Deermines if the point p is inside the corner points of the triangle b, b and c using barycentric coordinates [1, p. 46].

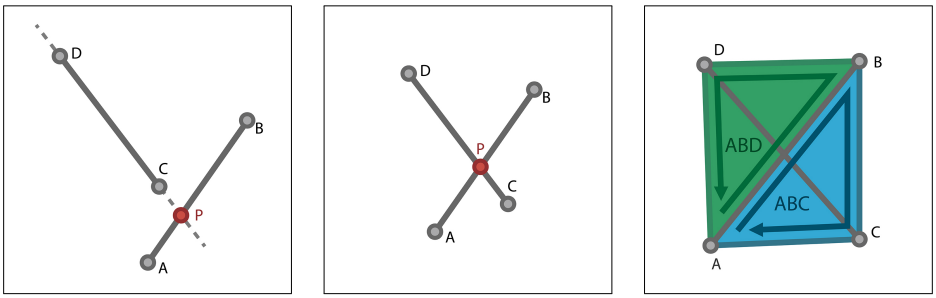
```
1:  pointInTriangle(p, a, b, c)
    Returns true if p is inside t else it returns false.
2:      v0 ← b − a
3:      v1 ← c − a
4:      v2 ← p − a
5:      d00 ← dot(v0, v0)
6:      d01 ← dot(v0, v1)
7:      d11 ← dot(v1, v1)
8:      d20 ← dot(v2, v0)
9:      d21 ← dot(v2, v1)
10:     denom ← d00 · d11 − d01 · d01
11:     v ← (d11 · d20 − d01 · d21) / denom
12:     w ← (d00 · d21 − d01 · d20) / denom
13:     u ← 1 − v − w
14:     if (v >= 0 AND w >= 0 AND u < 1) then
15:         return true
16:     end if
17:     return false
18: end
```

4.3.6 Line-line-intersection test

This test determines, if the two line segments AB and CD overlap, for this it needs to figure out if A and B are on different sides of CD and if C and D are on different sides of AB. The test for C and D is to verify, that the triangles ABD and ABC wind in different directions (see Fig. 4.5). Is the signed area of a triangle positive it winds counterclockwise, if the signed area is negative it winds clockwise [1, p. 152]. The algorithm 4.6 calculates the signed area. The algorithm 4.7 then uses the function to determine if two line segments intersect.

4.3.7 Area of a polygon

In the 2D scenario the area of the polygon is calculated with the following algorithm.



(a) (b) (c)

Figure 4.5: The points AB and CD are the end points of two line segments, the point P marks the intersection point. The image on the left (a) shows the situation when there is no intersection. The image in the middle (b) shows the situation when there is an intersection. The image on the right (c) shows the two created triangles ABD (green triangle) and ABC (blue triangle) with their wind [1, p. 152].

Algorithm 4.6: Calculates the signed area of a triangle from the provided corner points a, b and c of a triangle [1, p. 152].

```
1: SIGNEDAREATRIANGLE(a, b, c)
   Returns 2 times the signed triangle area.
2:     return (a.x − c.x) · (b.y − c.y) − (a.y − c.y) · (b.x − c.x)
3: end
```

Algorithm 4.7: Test if the two line segments AB and CD have an intersection. If there is an intersection point it is stores in p which is a reference to a point. a and b are the endpoints of AB and c and d are the endpoints of CD [1, p. 152].

```
1: INTERSECTIONLINELINE(a, b, c, d, p)
   Returns true if AB and CD have an intersection and false if not.
2:     a1 ← signedAreaTriangle(a, b, d)
3:     a2 ← signedAreaTriangle(a, b, c)
4:     if (a1 · a2 < 0.0) then                ▷ wind in different directions
5:         a3 ← signedAreaTriangle(c, d, a)
6:         a4 ← signedAreaTriangle(c, d, b)
7:         if (a3 · a4 < 0.0) then
8:             t ← a3 / (a3 − a4)
9:             p ← a + t · (b, − a)
10:            return true
11:        end if
12:    end if
13:    return false
14: end
```

Algorithm 4.8: Calculates the area of the 2D polygon p [32].

```
1:  CALCULATEAREA(s)
    Returns the area of s.
2:      tx ← 0                          ▷ temporary variable in x direction
3:      ty ← 0                          ▷ temporary variable in y direction
4:      s ← length(p)
5:      for i ← 0, ..., s do
6:          tx += p[i].x · p[(i + 1) % s].y
7:          ty += p[i].y · p[(i + 1) % s].x
8:      end for
9:      return (tx − ty) / 2
10: end
```

4.4 Implementation

The implementation of the test project is written in the programming language C++. The project has been split into different classes to separate the functionality. This makes it easier to understand, explain and update.

4.4.1 Used libraries

The following libraries have been used in the test project.

GLM

A C++ mathematics library based on the *OpenGL*[1] shading language *GLSL*[2]. *GLM* offers all the functionalists that *GLSL* has and also provides extensions in various fields. It is a recommended library for physic simulation, software rendering and image processing [35].

GLFW

Is an open source, cross-platform, lightweight utility library for using *OpenGL*. It offers support for multiple windows, creates *OpenGL* context and manages inputs from mouse, keyboard and joysticks. *GLFW* has an active development status and that makes this library surpass others [36].

[1] *OpenGL* (Open graphics library) is a cross-language and multi-platform API for creating 2D and 3D content [33].

[2] *GLSL* is a shading language based on the programming language C [34].

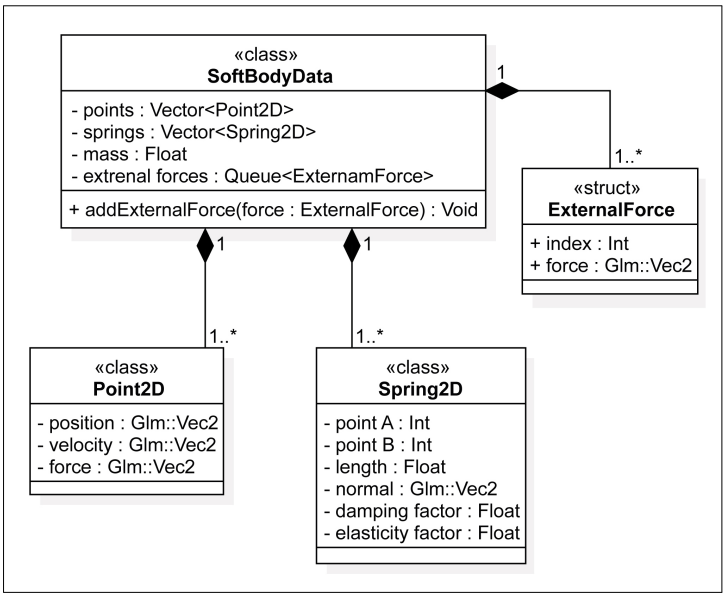


Figure 4.6: The image shows the important functions and variables of the implantation of the Point2D, Spring2D and SoftBodyData class. Getter and setter functions are not shown in the image.

4.4.2 Test project

This section provides an overview of the most important classes of the implemented test project and gives insight of their structure and functionality. Each and every class has functions to read, write and update their variables.

Point2D

Is one of the two most basic classes of the simulation. It is the element, that has been referred to as mass point, node or particle in previous sections. It stores the following variables:

- *position*,
- *velocity*,
- *force*.

The *position* stores the current position of the Point2D. The *force* stores the sum of all the forces, that are acting on the Point2D. The *velocity* stores the current acting velocity on the Point2D.

Spring2D

Is the second most basic class and it connects two Point2D objects, which define the two end points of the spring. To identify the associated Point2D objects, an index is stored in the class. It also stores all the parameters, that define the characteristics of the spring:

- *point A*,
- *point B*,
- *length*,
- *normal*,
- *damping factor*,
- *elasticity factor*.

The variables *point A* and *point B* are the indexes of the Point2D objects. The *length* is the resting length of the spring. The spring tries to maintain this length during the simulation. The normal is the direction in which the spring is able to apply its force. The *damping factor* and *elasticity factor* are the material factors that control the behavior of the spring.

SoftBodyData

This class combines Point2D objects and Spring2D objects to a soft body for the simulation:

- *points*,
- *springs*,
- *mass*,
- *external forces*.

The vector *points* holds all the Point2D objects. The index of these Point2D object in the vector is the index mentioned in section 4.4.2. The vector *springs* holds all the Spring2D objects. The *mass* stores the total weight of the entire soft body. The queue *External forces* holds all the forces, that act from the outside on the SoftBodyData object. The function *addExternal-Force()* adds these to the queue. An external force consists of a $force$ and a *index*, that is a index to the Point2D object.

WorldData

This class defines the environment of the simulation and it contains the variables:

- *gravity*,
- *softBodies*,
- *borders*.

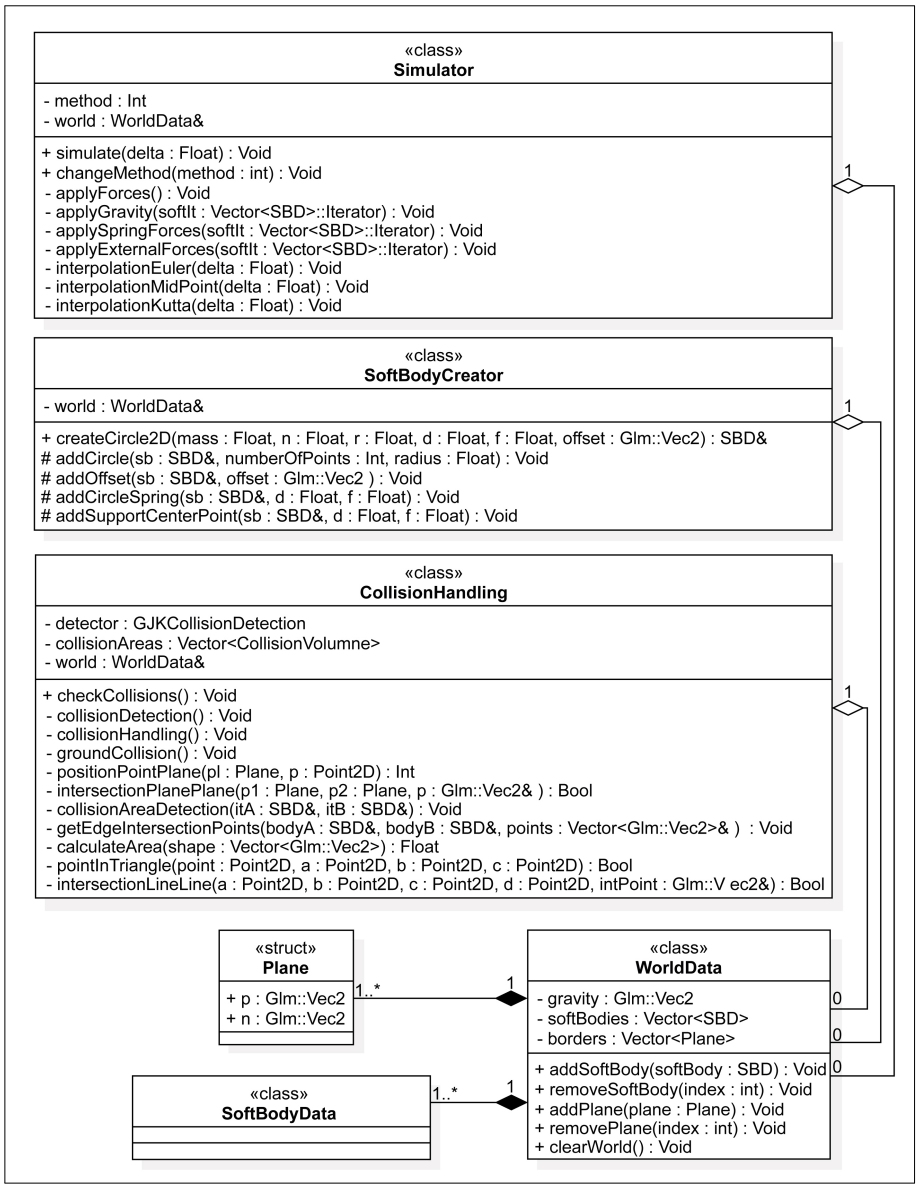


Figure 4.7: The image shows the important functions and variables of the implantation of the Simulator, SoftBodyCreator, CollissionHandling and WorldData class. To save space SoftBodyData is shortened to SBD. Getter and setter functions are not shown in the image.

The variable *gravity* is the acting gravity in the environment. It affects all the active SoftBodyData objects during the simulation. The vector *softBodies* hold all the SoftBodyData objects. The vector *borders* store the borders of the simulation and it is a plane object (see Fig. 4.7).

Simulator

This class calculates all the forces and applies them together with the external forces to the associated Point2D objects. The class contains the variables:

- *method*,
- *world*.

The variable *method* states the integration method the class uses. The variable *world* stores a reference of the WorldData object and over it the class has access to the SoftBodyData objects. This class also holds the integrators, mentioned in section 3.3.

The function *simulate* updates the every object in *world* with the selected interpolation method. The whole apply functions calculate the forces that occur during the simulation and apply them to the individual SoftBodyData objects.

CollisionHandling

The class detects the collisions in the simulation and also reacts to these collisions. It contains the variables:

- *detector*,
- *collisionAreas*,
- *world*.

The *detector* is an implementation of the GJK algorithm (see Fig. 4.8). The vector *collisionAreas* holds all the detected collisions. The variable *world* is a reference to the WorldData object and allows the class to get access to the SoftBodyData objects. The class contains all sorts of intersection tests, that are mentioned in section 4.3. The function *checkCollisions()* calls the two functions *collisionDetection()* (broad phase) and *collisionHandling* (narrow phase).

In *collisionDetection()* all the SoftBodyData objects are compared with each other with the *detector*. If there is a positive collision the two SoftBodyData objects are marked and stored in the *collisionAreas* vector.

In *collisionHandling* the points, that are intersecting another are extracted from a SoftBodyData couple with the point-in triangle test from section 4.3.5. These points are added to a new shape, that will become the collision shape. Then, the spring of the SoftBodyData couples are checked for intersections with the Line-line-intersection test from section 4.3.6. These intersection points are added to the collision shape as well. It calculates the area of the collision shape with algorithm from section 4.3.7. The area sets the base for the separation force. It adds the separation force as an external force to the SoftBodyData object and the direction of the separation force face away from the other SoftBodyData object.

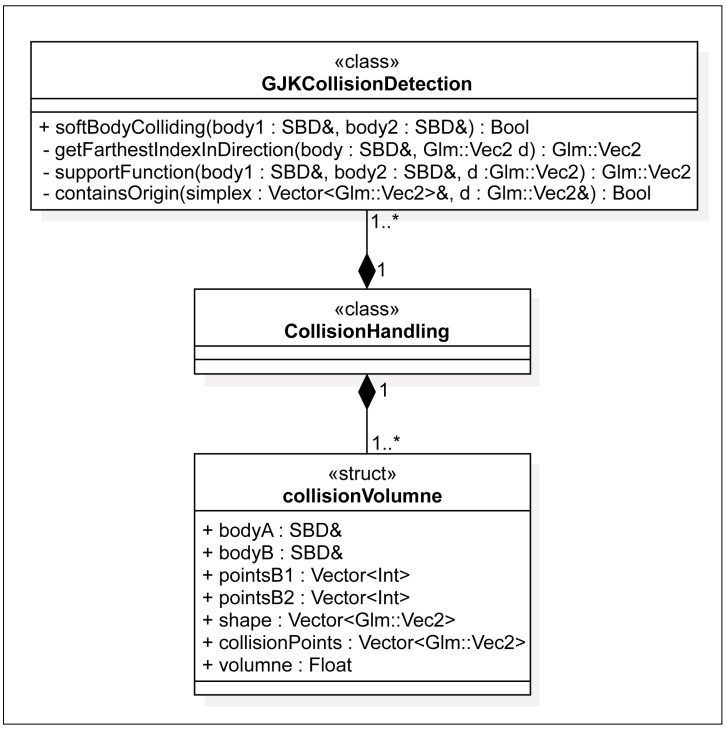
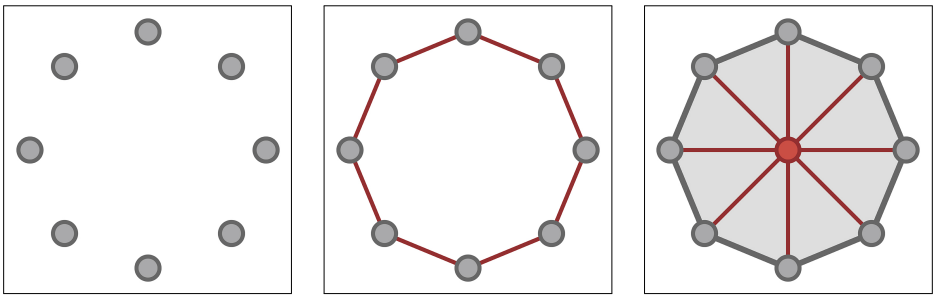


Figure 4.8: The image shows the important functions and variables of the implantation of the Simulator, SoftBodyCreator, CollissionHandling and WorldData class. To save space SoftBodyData is shortened to SBD. Getter and setter functions are not shown in the image.

The collision with the border planes works a bit different. In the beginning of the *narrow phase* the individual Point2D objects of each SoftBodyData object is tested against the border planes with the plane-point-position test from section 4.3.3. If the point is behind the plane the plane-plane-intersection test from section 4.4 determines the intersection point. Then the position of the Point2D object is set to the intersection point and the velocity is reflected against the normal of the plane.

SoftBodyCreator

The purpose of this class is to automate the creating of SoftBodyData objects. If offers different functions to create different types of SoftBodyData objects and adds these to the WorldData. To do so, it stores a reference to the WorldData object and adds the new SoftBodyData object to it with the *addSoftBody()* function of WorldData (see. Fig. 4.7). The class implements the creation process for a circle shape. The function *createCircle2D()* call the three following functions for the creation process. The function *createCircle()* creats points on the hull of the circle shape and adds thouse points to



(a) (b) (c)

Figure 4.9: The image on the left (a) shows the soft body with the initial hull points. The image in the middle (b) shows the springs (red lines) that connect the hull points. The image on the right (c) shows the structural springs (red lines) inside the hull with a center point (red circle).

a SoftBodyData object *sb* (see Fig. 4.9 (a)). The function needs the number of point p that are on the hull and the radius r of the circle (see Alg. 4.9). In the next step it calls the function *addCircleSpring()*. This function adds

Algorithm 4.9: Adds Point2D objects in a circle shape to a SoftBodyData object.

```
1: addCircle(sb, p, r)
   Adds Point2D objects to sb in a circle shape.
2:     for i ← 0, ..., p do          ▷ create the requested number of points
3:         angle ← i · ((2 · π) / p)
4:         sb.addPoint(r · sin(angle), r · cos(angle))
5:     end for
6: end
```

Spring2D objects to *sb* and conects the existing Point2D objects (see Fig. 4.9 (b)). Additional to *sb*, the function also requires the damping coefficient d and the elasticity factor f of the spring (see Alg 4.10). The last step is to

Algorithm 4.10: This algorithm adds springs between the individual points to create a circle shape.

```
1: addCircleSpring(sb, d, f)
   Adds Spring2D objects to sb to create the outside hull.
2:     n ← length(sb.points)
3:     for i ← 0, ..., n do
4:         sb.addSpring(i, (i + 1) % n, d, f)
5:     end for
6: end
```

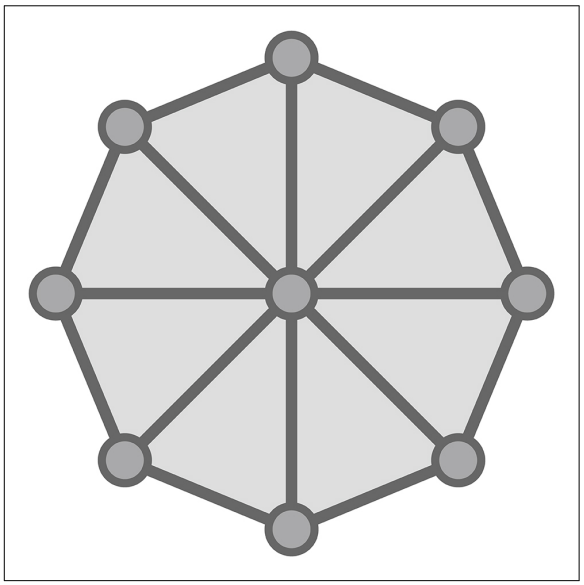


Figure 4.10: The soft body, that is used in the test phase of the simulation. It is the same soft body from section 4.4.2.

add the structural support spring to the inside of *sb* (see Fig. 4.9 (c)). The function *addSupportCenterPoints()* does this (see Alg. 4.11).

Algorithm 4.11: This function adds an internal structure to SoftBodyData object. It works for convex shapes.

```
1: addSupportCenterPoint(sb, d, f)
   Adds a center point to sb and connects it with the hull points.
2:     center ← average (sb)
3:     n ← sb.getLength()                ▷ Get length before adding center
4:     sb.addPoint(center)
5:     for i ← 0, . . . , n do
6:         sb.addSpring (i, n, d, f)
7:     end for
8: end
```

4.5 Project testing

After the implementation, the project is tested to determine what is possible and where the limits of the simulation are.

Table 4.1: The table shows the maximum number of soft body elements, which the simulation is able to handle, before the frames per second drops below 60. The test was done with collisions and without collisions of the individual soft body elements. For each test case the scenario was repeated 5 times and the results averaged.

	With collision	*Without coll.*
Euler integrator	132	3534
Mid point integrator	72	1817
RK fourth order integrator	36	912

4.5.1 Test object

The test object is a circle with eight hull points and one middle point (see Fig. 4.10). The section 4.4.2 shows the whole creation process in detail.

4.5.2 Test scenario

The test scenario will push the implementation to its limits and figure out how many soft body objects it can handle. The setup consists of one border plane. It functions as a ground where the soft body objects pile up. New soft body objects spawn above the ground plane and fall towards it and onto existing soft body objects. New soft body objects are added until the FPS fall under 60. This test repeats for each integrator (see Sec. 3.3). In the second test the collision detection between the individual soft body objects is turned off. The rest of the setup stays the same and the test is repeated for each integrator.

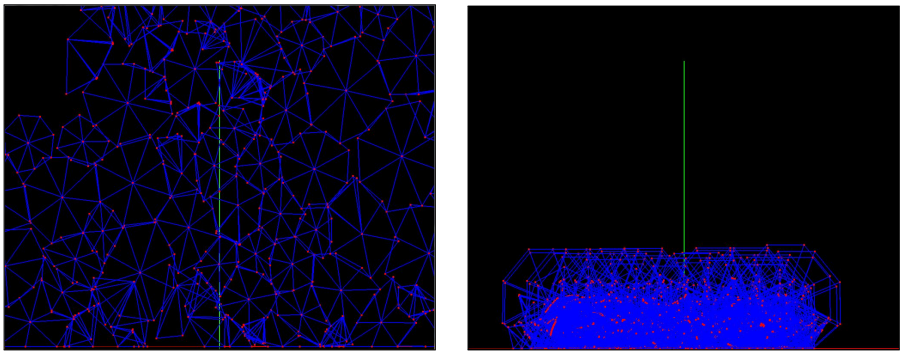
Test hardware

The test runs on a HP ZBook 15 business laptop with the following configuration:

- *Operating system*: Windows 7 Professional 64-Bit
- *Processor*: Intel Core i7-4800MQ CPU @ 2.70 GHz
- *RAM*: 16 GB

4.5.3 Test result

The table 4.1 shows the maximum number of soft body objects with and without collision detection. The table confirms, what has been mentioned at the end of section 3.3.6. The Euler integrator manages to handle the most soft body elements and the Runge Kutter fourth order integrator manages the least elements. The result also shows, that when the collision between



(a) (b)

Figure 4.11: The image on the left (a) shows the test scenario with collisions, during the simulation. The soft body objects pile up and the heavier objects squeeze and deform the lighter ones. This leads to unwanted deformations and a couple of soft body objects fold into itself. This pressure is also responsible that a couple of soft body objects overlap. The image on the right (b) shows the test scenario without collisions. The soft body objects fall towards the ground where the objects swing until they take on a resting position.

the soft body objects is turned off, the simulation is able to handle way more elements. This is possibly the reason why soft body simulation is used more often as an effect simulation, than as a gameplay element.

4.6 Summary

This chapter starts with the selection of the deformation model. It lists important aspects, on which the selection is based on. It then clarifies the MVC pattern and how the implementation applies it. The next section focuses on algorithms, a special focus is on the GJK algorithm. Then it shows and explains the architecture of the implementation, it goes into detail on the most important classes. The last section of this chapter describes the hardware, the test setup and results and explains them. The test proofs the assumptions, that the Euler integrator is the less resource intensive one of the three. The comparison between the simulation with and without collision proves, that simulations without collisions allow an increase of objects. This fact makes it more attractive to use soft body simulation mainly for effects, which improves the look and feel of computer games. In the context of limited resources the realistic deformation simulation is not efficient enough in comparison to its calculation effort.

Chapter 5

Conclusion

The master's thesis deals with the topic of soft body simulation and provides an overview of potential model to achieve it. It evaluates these models in the aspect of game development and finds a fitting model based on these aspects. It provides a detailed look into the calculations, that are necessary to implement a basic soft body simulation. Furthermore, it shows how an implementation can look like and what kind of algorithms are important.

A drawback of the thesis is, that the 3D part of the implementation is missing and just the 2D part has been covered. The collision detection and handling has been more difficult than expected in the beginning and in the end the 3D part was cut from the project. The view is functional but lacks in features. This can be avoided by using a game engine like Unreal or Unity.

5.1 Improvements and future work

A possible improvement is to use a parallel computing method. For example OpenCL or Cuda can be used to calculate the simulation on the graphics card (GPU). Tzvetomir Vassilev and Roumen Rousev published a paper in 2008 for a data structure and algorithm, that calculates a mass spring system on the GPU [8].

Another improvement is to include a adaptive refinement of the mass spring model described in an article from Hutchinson Dave, Preston Martin and Hewitt Terry publishd in 1996. In the article they described an approach to create visually pleasing results for less computation costs [3].

5.2 Personal statement

A debatable point of the implementation is the usefulness of a self implemented soft body simulation, compared to a already existing physic simulation. The size of the project and the motivation of the creator factor into this a lot. If a project is small and just needs a simple deformation, it may

be too much to include an existing physics engine or to pay the licenses for one. On the other side, if the project is big, it is a good idea to include a existing physic engine. Physic simulation can also be very theoretical and mathematical. This may discourage one or another, but if the resources, knowledge and motivation is there it is definitely an option.

Chapter 6

Content of the CD-ROM

6.1 PDF documents

Pfad: /

Dietrich_Benedict_2015.pdf Master's thesis

6.1.1 Literature

Pfad: /Literature

*.pdf . . . . . Copies of the used Literature as PDF documents.

6.1.2 Video

Pfad: /Video

Implementing_GJK_by_Casey_Muratori.mp4 Copies of the GJK explanation video.

6.1.3 Online literature

Pfad: /Online

*.pdf . . . . . Copies of the used web sides.

6.2 Project

Pfad: /Project

readme.txt . . . . . Short explanation on how to use the project.
Libraries/ . . . . . Contains the libraries the project requires.
Souce Code/ . . . . . Contains the source code of the project.
Executable/ . . . . . Contains the executable of the project.

Screenshots/ . . . . . Contains screenshots of the project.

6.3 Images

Pfad: /Images

*.pdf . . . . . Vectorgraphics
*.jpg . . . . . Graphics

References

Literature

[1] Christer Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004 (cit. on pp. 8, 9, 27–30).

[2] Jason Gregory. *Game Engine Architecture, Second Edition*. 2nd. Natick, MA, USA: A. K. Peters, Ltd., 2014 (cit. on pp. 7–9, 13, 24–28).

[3] Dave Hutchinson, Martin Preston, and Terry Hewitt. "Adaptive Refinement for Mass/Spring Simulations". English. In: *Computer Animation and Simulation '96*. Ed. by Ronan Boulic and Gerard Hégron. Eurographics. Springer Vienna, 1996, pp. 31–45. URL: http://dx.doi.org/10.1007/978-3-7091-7486-9_3 (cit. on p. 41).

[4] Maciej Matyka. "How To Implement a Pressure Soft Body Model". In: (2004) (cit. on pp. 5, 6).

[5] H. Mitter. *Mechanik*. Bibliogr. Institut Mannheim, 1989 (cit. on p. 13).

[6] A. Nealen et al. "Physically Based Deformable Models in Computer Graphics". In: *Computer Graphics Forum* 25 (2006), pp. 809–836 (cit. on pp. 4, 5).

[7] M. Song. *Interactive Elastic Two-Layer Soft Body Simulation with OpenGL*. LAP Lambert Academic Publishin, 2010 (cit. on pp. 1, 3–7, 13–19, 24).

[8] Roumen I Rousev Tzvetomir I Vassilev. "Algorithm and Data Structures for Implementing a Mass-spring Deformable Model on GPU". In: (2008) (cit. on p. 41).

Games

[9] Boneloaf. *Gang Beasts*. Windows, Linux, OS X, Playstation 4, Playstation Vita. 2015 (cit. on p. 10).

[10] Bugbear Entertainment. *Wreckfest*. Windows. 2015 (cit. on p. 10).

[11] Matheus Valadares. *Agar.io*. Browser, Windows, Linux, OS X, Android, iOS. 2015 (cit. on p. 9).

Online sources

[12] URL: https://en.wikipedia.org/w/index.php?title=Soft_body_dynamics&oldid=681715625 (visited on 09/28/2015) (cit. on pp. 1, 4, 14).

[13] URL: http://www.havok.com/products/physics (visited on 01/07/2015) (cit. on pp. 2, 9).

[14] URL: http://bulletphysics.org/wordpress/ (visited on 01/07/2015) (cit. on pp. 2, 9).

[15] URL: http://www.technologyx.com/featured/understanding-frame-rate-look-truth-behind-30v60-fps/ (visited on 08/26/2015) (cit. on p. 7).

[16] URL: http://gafferongames.com/game-physics/fix-your-timestep/ (visited on 03/02/2015) (cit. on p. 8).

[17] URL: http://gamedev.stackexchange.com/questions/1589/when-should-i-use-a-fixed-or-variable-time-step (visited on 08/26/2015) (cit. on p. 8).

[18] URL: https://developer.nvidia.com/gameworks-physx-overview (visited on 09/18/2015) (cit. on p. 9).

[19] URL: https://en.wikipedia.org/w/index.php?title=PhysX&oldid=682614948 (visited on 09/28/2015) (cit. on p. 9).

[20] URL: https://en.wikipedia.org/w/index.php?title=Bullet_(software)&oldid=676518643 (visited on 09/18/2015) (cit. on p. 9).

[21] URL: http://agar.io/ (visited on 09/14/2015) (cit. on p. 10).

[22] URL: https://en.wikipedia.org/w/index.php?title=Agar.io&oldid=682613178 (visited on 09/14/2015) (cit. on p. 10).

[23] URL: http://gangbeasts.com/ (visited on 08/26/2015) (cit. on pp. 10, 11).

[24] URL: https://en.wikipedia.org/w/index.php?title=Gang_Beasts&oldid=679242281 (visited on 08/28/2015) (cit. on p. 10).

[25] URL: http://nextcargame.com/ (visited on 08/26/2015) (cit. on pp. 10, 11).

[26] URL: https://en.wikipedia.org/w/index.php?title=Wreckfest&oldid=676203737 (visited on 08/26/2015) (cit. on p. 10).

[27] URL: https://en.wikipedia.org/w/index.php?title=Newton's_laws_of_motion&oldid=682609121 (visited on 09/28/2015) (cit. on pp. 12, 13).

[28] URL: hhttps : / / en . wikipedia . org / w / index . php ? title = Ordinary _ differential_equation&oldid=678790331 (visited on 09/28/2015) (cit. on p. 15).

[29] URL: https://en.wikipedia.org/w/index.php?title=Model%C3%A2%C2%80%C2%93view%C3%A2%C2%80%C2%93controller&oldid=681473142 (visited on 09/17/2015) (cit. on p. 23).

[30] URL: http : / / mollyrocket . com / 849 (visited on 09/17/2015) (cit. on pp. 24, 27).

[31] URL: https : / / en . wikipedia . org / w / index . php ? title = Barycentric _ coordinate_system&oldid=674534623 (visited on 09/22/2015) (cit. on p. 28).

[32] URL: http : / / www . wikihow . com / Calculate - the - Area - of - a - Polygon (visited on 09/22/2015) (cit. on p. 31).

[33] URL: https : / / en . wikipedia . org / w / index . php ? title = OpenGL & oldid = 682654009 (visited on 09/28/2015) (cit. on p. 31).

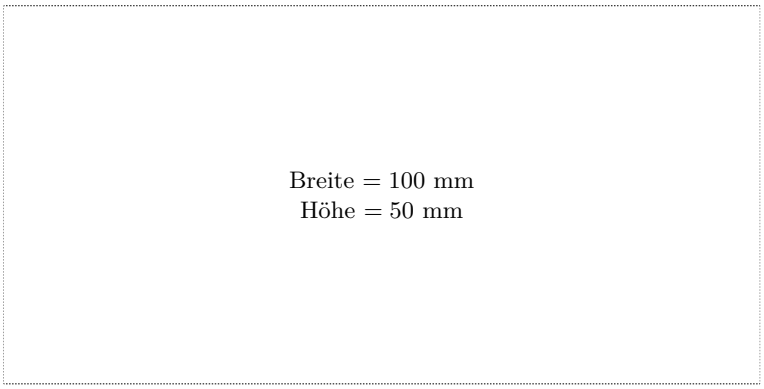
[34] URL: https://en.wikipedia.org/w/index.php?title=OpenGL_Shading_Language&oldid=680733601 (visited on 09/15/2015) (cit. on p. 31).

[35] URL: http://glm.g-truc.net/ (visited on 09/09/2015) (cit. on p. 31).

[36] URL: http://www.glfw.org/ (visited on 09/09/2015) (cit. on p. 31).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —