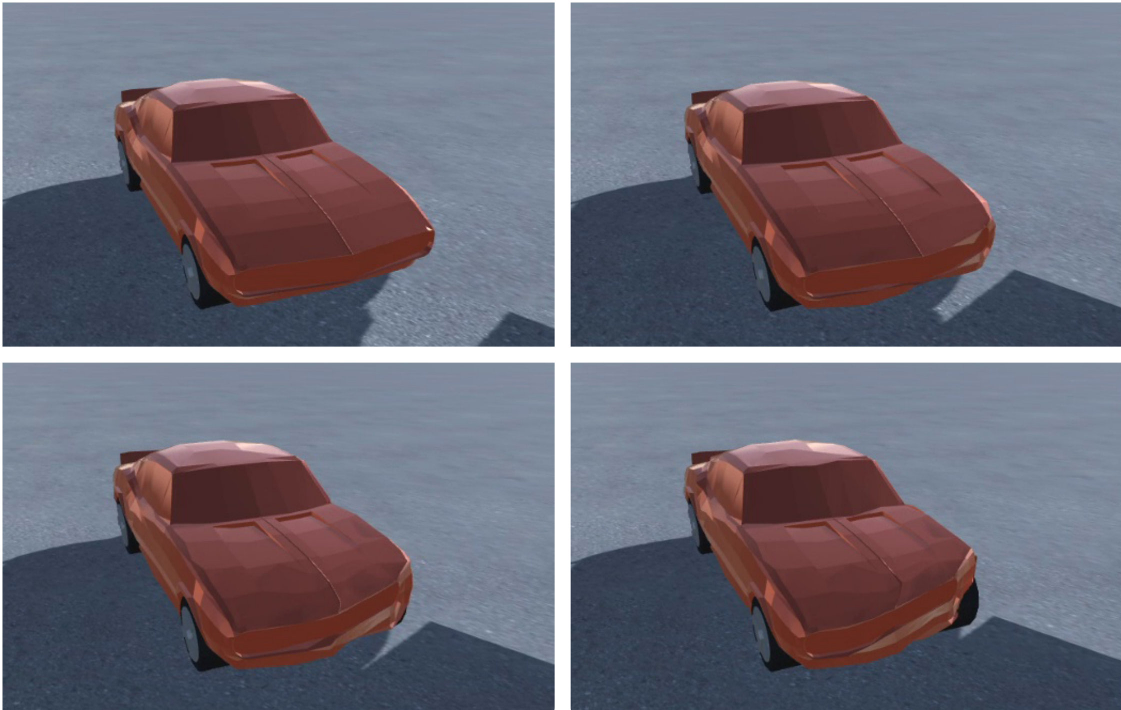

Real-Time Plastic Deformation of Car Bodyworks

Master's thesis in Computer science and engineering

Tom Ille

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2020

Real-Time Plastic Deformation of Car Bodyworks

TOM ILLE

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Real-Time Plastic Deformation of Car Bodyworks
TOM ILLE

© TOM ILLE, 2020.

Supervisor: Marco Fratarcangeli, Department of Computer Science and Engineering, Chalmers
Examiner: Name, Department

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Four stages of a car being deformed by a collision with an obstacle. The upper right image shows the original state of the car body. The other three images show the increasing deformation of the cars front-left as it collides further with an obstacle. The obstacle is not rendered to increase the visibility on the deformation.

Typeset in LaTeX
Gothenburg, Sweden 2020

Real-Time Plastic Deformation of Car Bodyworks
Tom Ille
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Realism in video games is furthered each year. Particularly in racing and driving games, visual realism in the deformation of cars play a larger and larger role for the immersion in a virtual world. With the improvements of modern hardware, a new physically based simulation approach for the deformation of object has emerged. In this thesis a prototype is developed that aims to implement such a deformation system for car bodyworks. One of the many challenges is to generate visually appealing deformations, while remaining within the constraints of the real-time context. There is a variety of deformation techniques in the body of research. At their core, most work similarly. The deformable object is discretized into smaller units, so called particles. These particles are subject to the forces of the virtual environment and thus adjust the superordinate deformable object. The method of choice for this thesis' prototype is position-based deformation, as it has many advantages for a real-time context. In position-based deformation, the particles are interconnect via constraints, which adjust their positions in relation to one-another. These constraints are solved each frame by an iterative Gauss-Seidel solver.
It was integrated into a deformation module which is used by the physics engine *Unity* to compute deformation results. This configuration proved successful as it makes use of the strengths of both a third-party physics engine and a more performant module for time-critical algorithms. The prototype was developed based on an agile software development philosophy and was continuously improved and optimized. The prototype was analyzed regarding the computational performance and the visual results. Depending on its configuration, the system computes deformations within 5-150 ms per frame on an Intel i5-8500 CPU. The results suggest that the performance can be enhanced by using a more sophisticated solver method and by utilizing the GPU. The visual results are promising, but suggest that properties must be distributed thought an object in a non-uniform manner. This can generate a more visually interesting result, as it mimics the existence of vehicle parts that are varying in their structural rigidity.

Keywords: computer graphics, plastic deformation, position based deformation.

Acknowledgements

I would like to thank Marco Fratarcangeli for the ongoing support and advice throughout the extended creation of the thesis.

Tom Ille, Gothenburg, June, 2020

Contents

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
1.1 Purpose and research questions . . . 2
1.2 Delimitations . . . 2
1.3 Requirements . . . 3
1.3.1 Performance . . . 3
1.3.2 Stability . . . 3
1.3.3 Integratability . . . 3
1.3.4 Controllability . . . 4

2 Background **5**
2.1 History of Real-Time Deformation . . . 5
2.2 Deformation Techniques . . . 5
2.2.1 Physically Based Methods . . . 6
2.2.1.1 Mass-Spring System . . . 6
2.2.1.2 Finite Element Method . . . 6
2.2.1.3 Finite Difference Method . . . 7
2.2.1.4 Finite Volume Method . . . 7
2.2.2 Position-Based Methods . . . 7

3 Theory **9**
3.1 Deformation . . . 9
3.2 A Simplified Car Model . . . 9
3.3 Choosing the Deformation Method . . . 10
3.3.1 The Mass-Spring Model . . . 10
3.3.2 Physically-Based Continuum Methods . . . 10
3.3.3 Position-Based Deformation . . . 11
3.4 Surface Mesh and Tetrahedral Mesh . . . 11
3.4.1 Surface Mesh . . . 11
3.4.2 Tetrahedral Mesh . . . 11
3.5 Barycentric coordinate system . . . 12
3.6 Constraints . . . 14
3.6.1 Distance constraints . . . 14
3.6.2 Volume constraints . . . 15

3.6.3 Constraint Stiffness . . . 16
3.6.4 Collision constraints . . . 16
3.7 Solver . . . 17
3.7.1 Gauss-Seidel Solver . . . 17
3.7.2 Jacobi Solver . . . 17
3.8 Plasticity . . . 18
3.9 Position-Based Dynamics . . . 19

4 Methodology **21**
4.1 Requirements . . . 21
4.2 Development Process . . . 21
4.3 Tools . . . 22
4.3.1 Tools for Agile Development . . . 22
4.3.2 Functionality Tools . . . 22
4.3.3 Choosing a Physics Engine . . . 23
4.3.4 *Unity* . . . 23
4.4 Testing Projects . . . 24
4.5 Summary . . . 26

5 Execution **27**
5.1 Overview . . . 27
5.2 The Set-Up . . . 27
5.3 Execution flow . . . 28
5.4 Preparation and Initialization . . . 30
5.4.1 Importing mesh data from *TetWild* . . . 30
5.4.2 Constraint Generation . . . 31
5.4.3 Surface Mesh to Tetrahedral Mesh Mapping . . . 32
5.4.4 Barycentric Mapping . . . 32
5.4.5 Serialization . . . 33
5.4.6 Code Architecture . . . 34
5.5 Deformation Loop . . . 35
5.5.1 Collision Handling . . . 35
5.5.1.1 Coarse Collision Detection . . . 36
5.5.1.2 Fine Collision Detection and Collision Response . . . 36
5.5.2 Constraint Solving . . . 37
5.5.3 Solver . . . 37
5.6 Parallelization . . . 37
5.7 Force Feedback . . . 38
5.8 Summary . . . 39

6 Results **41**
6.1 Critical Goal Assessment . . . 41
6.2 Visual Analysis . . . 43
6.3 Performance Analysis . . . 44
6.3.1 Methodology . . . 44
6.3.2 Initialization . . . 45
6.3.2.1 Parallelization . . . 46

6.3.2.2 Barycentric Mapping 47
6.3.2.3 Data Serialization 48
6.3.3 Deformation Loop 48
6.4 Outlook and Future Work 50
6.4.1 Utilizing the GPU 50
6.4.2 A Parallel Gauss-Seidel Solver 51
6.4.3 Simulated Force Feedback from Obstacle Collisions 51
6.4.4 Saving and Loading Deformation States 51
6.4.5 Weight-Painting for Deformation Parameters 52

7 Conclusion **55**

Bibliography **57**

List of Figures

1.1 Different aesthetics of crashes in *Beam.NG drive* and *Wreckfest* . . . 4

3.1 Spheres represented by a surface mesh and a tetrahedral mesh. 12
3.2 Barycentric mapping in two dimensions. 13
3.3 Position update of a point that is barycentrically mapped to a triangle. 14
3.4 Distance preservation of an edge via a distance constraint. 15
3.5 Tetrahedral volume preservation via a volume constraint. 16
3.6 Projection of collision constraints. 17

4.1 Simplified order of *Unity* events. 25
4.2 Testing projects with varying complexity. 26

5.1 An overview of the scene hierarchy of the prototype. 28
5.2 A flow diagram of the prototype. 29
5.3 File formats for tetrahedral and surface mesh files. 31
5.4 File format of serialized tetrahedral mesh data. 34
5.5 Exemplarily serialized data. 34
5.6 Excerpt of the flow chart focusing on the deformation loop. 35
5.7 Comparison between *AoS* and *SoA*. 38

6.3 Selection of meshes for performance analysis. 45
6.4 Load distribution for different impacts on a Mercedes-Benz CLS-Class. 52
6.5 Mock-Up of a parameter painting tool. 53

List of Tables

6.1 Computation time of a serial initialization. 46
6.2 Comparison of a serial and a parallel initialization step. 46
6.3 Computation time of a parallel initialization. 47
6.4 Direct mapping quotas in different meshes. 47
6.5 Comparison of parallel initialization and parallel initialization with direct mapping. 47
6.6 Computation time of a parallel initialization with direct mapping. . . 48
6.7 Comparison of parallel initialization with direct mapping and data serialization. 48
6.8 Average computation times of serial deformation loops of different meshes. 49
6.9 Average computation times of parllel deformation loops of different meshes. 49
6.10 Comparison of serial deformation loop and parallel deformation loop. 50

1

Introduction

Consumers have a never-ending demand for increased realism in video games. This applies to both realism in gameplay mechanics and computer graphics. The demand is fueled by ever-increasing hardware capabilities for ever-decreasing costs. This development does not exclude driving games. To make driving errors more meaningful it is not enough to penalize the player through the gameplay. For a full immersion, the game needs a visually appealing damage model.
Initiated by *Destruction Derby* [4] in 1995, visually detailed damage models started to find their way into racing games. *Destruction Derby* was far ahead of competitors as such damage models opened up a lot more opportunities to further graphical realism in driving games. It took another ten years until 2005, when *Rigs of Rods* [18] was published. *Rigs of Rods* is an open-source physics engine, which employed soft-body physics and marked the beginning of modern physics-based simulation techniques in video games. Prior to the development of physics-based deformation methods, deformations of vehicles were realized by displaying precomputed or manually created, damaged car parts on collision. Artists prepared different deformed versions of a part of the bodywork (i.e. a door) which were then swapped with the original model on collision. While this method had great success in the past, it lacks in detail, variety and visual plausibility. In recent years therefore, some game developers started to emphasize simulation-based deformation techniques. Prominent examples are *BeamNG.drive* (2015) [26], which was developed by a group of *Rigs of Rods*-contributors and *Wreckfest* [36] (2018). Both games employ extensive and sophisticated visual damage models. Despite the rise of such techniques in recent years, published information about them are sparse due to the commercial nature of the projects. This thesis aims to investigate the development of a simulation-based deformation system for real-time applications. The report describes the steps taken to successfully develop a prototype implementing such system, as well as the discoveries made.
The structure of this thesis is as follows:

- Chapter 1 Introduction. The remainder of this chapter presents the research question, the delimitations and the requirements of this thesis.
- Chapter 2 Background. A synopsis of the history of dynamic deformation in driving games. Video games that are known for prominently featuring deformations of cars are presented. The existing body of research is presented and the most commonly used deformation techniques are shown.
- Chapter 3 Theory. The previously presented deformation techniques are analyzed regarding their practicality for this thesis. The Position-Based defor-

mation is chosen as a basis for the prototype development. Common concepts from the fields of physics, mathematics and computer graphics are explained and contextualized within the thesis.

- Chapter 4 Methodology. Development tools and methodologies during the prototype development are presented and motivated.
- Chapter 5 Execution. The development of the prototype is described in detail. The individual features' evolution throughout time is explained. Challenges that were met throughout the prototype development are presented and solutions are motivated.
- Chapter 6 Results. The final prototype is evaluated based on the delimitations and requirements described in chapter 1 Introduction. The visual quality of the deformations and the performance of the used algorithms are extensively assessed.

1.1 Purpose and research questions

The aim of the thesis is to develop an understanding of how to develop a deformation model for real-time applications. Real-time dependant requirements are considered to ultimately answer the following two research questions.

> *Which factors must be considered when designing and developing a real-time deformation system for car bodyworks in video games? How can such a method be incorporated into a modern game engine?*

This question was used to generate a set of resulting requirements. The requirements were then used to iteratively develop a prototype, with the intent to answer the research questions. Discoveries made throughout the prototype development were used to reconsider the individual requirements. The final prototype answers aspects of the research questions in differing detail. Section 6.4 offers suggestions for future work that can further develop the aspects described in the upcoming sections.

1.2 Delimitations

The goal for the prototype is to create a deformation model for the bodyworks of cars. This is limited to the metal of the bodyworks. Other parts and materials of the car, such as the glass of windows, the rubber of wheels or the interior are only modeled primitively as rigid bodies or are completely omitted. The virtual environment of the car was also modeled primitively. Obstacles the car can collide with, were modeled as primitive shapes. The target platform for the prototype is a Windows PC.

1.3 Requirements

The deformation method in this thesis was developed for real-time, interactive applications such as video games. This circumstance yields a number of requirements. Generally, the plausibility of the approximation is of higher importance than physical correctness and completeness. The highest focus was on the performance and efficiency of the used algorithms. Minor sacrifices in visual quality and physical correctness were therefore justifiable by enhancing the computation speed or conserving computing resources. The following sections lay out a set of requirements for a deformation technique that is sufficient for a real-time context as well as possible approaches that could help meeting such requirements.

1.3.1 Performance

One of the most important aspects is the efficiency of the algorithms in use. Commonly video games have a desired frame rate of 60 frames per second (fps). Any loss of frame rate below this limit is clearly noticeable by the consumer according to –todays standards. Consequently, one frame has approximately 17 ms to be fully computed and rendered onto the screen. This includes lighting, rendering, AI computations, user input and animations, besides the deformation. This is no simple task and the code has to be as time-efficient as possible. Increasing the performance can be done using code parallelization, efficient data structures or by utilizing the graphics hardware.

1.3.2 Stability

The deformation technique needs to be unconditionally stable. This means that the method can not corrupt over time or in the case of unpredicted forces. It is a known phenomenon in the field of simulation-based deformation, that some techniques can become unstable over time. As the user may drive the same car for an unknown duration this behaviour has to be prevented strictly. This requirements can mostly be met by choosing techniques and algorithms that are known to be stable.

1.3.3 Integratability

Current video games almost exclusively use surface-meshes to represent their three-dimensional objects. The deformation method therefore needed to work with arbitrary surface-meshes after being integrated into the games system. It needed to be avoided to design the deformation technique to be dependant on a certain characteristic of a surface-mesh. It needed to also be able to correctly detect collisions with common shapes such as cubes or spheres. It was also required that the code of the deformation technique was written in a way that allowed the addition of new features. For example, modern video games use collision systems that are more complex than just primitives. Improving the collision system needed to not alter any other code of the deformation technique.

1.3.4 Controllability

The method was designed to be used in an interactive, real-time environment. Therefore, the technique needed to offer a set of adjustable parameters which could predictably tweak the deformation system. This was important because individual games and projects have different artistic and aesthetic goals. This point is illustrated at the example of *BeamNG.drive* and *Wreckfest* in figure 1.1.

Figure 1.1: An example of two crashes presented in two aesthetically different manners. The crash in *BeamNG.drive*[26] (left) is much cleaner with less debris being propelled off the cars. The crash in *Wreckfest*[36](right) deforms the car less, but features more visual effects and debris. This difference impacts how frequent and encouraged crashes are in the specific game and how impactful they look and feel.

2

Background

This chapter presents a synopsis of the history of real-time deformation in driving games. The evolution from *Destruction Derby* in 1995 through *Wreckfest* in 2018 is described. The classic approach to dynamic deformation as well as a collection of current deformation methods are presented.

2.1 History of Real-Time Deformation

The traditional approach to deformation in cars simulation is a combination of a physical simulation and an application of manually created assets from artists. Once a collision between an object and a deformable car is detected, the affected exterior parts get switched to a deformed version. The earliest video game employing deformation of car exteriors is *Destruction Derby* (1995)[4]. While the deformation technique is rather simple, it was very ahead of its times, no other racing games used such techniques. The following years, more and more racing and driving games started to feature car deformation systems. Notable examples are *Street Legal Racing: Redline* (2003) [13], *FlatOut* (2004) [16] and *TOCA Race Driver* (2003) [10], which all used developed damage and deformation models for their times and are well known for the unique integration of the damage models into their gameplay.
Only in recent years, game developers started to replace the traditional approach to deformation with a simulation-based approach. This is due to the rise of both demand for more realistic graphics and the availability of computing power in graphics hardware. Two very notable games employing modern techniques are *Wreckfest* (2018) [36] and *BeamNG.drive* (2015) [26]. Both employ highly sophisticated simulation based deformation techniques, which are adjusted to fit the aesthetic concept of the game. As the specifics of their deformation systems are not published due to the commercial nature of the games, it is hard to say which set of algorithms they used to develop their systems. It can however be said that they are forerunners of using simulation based deformation techniques in a real-time context and in the future more and more driving and racing games will employ similar techniques to further graphical quality and realism.

2.2 Deformation Techniques

There are a number of deformation techniques used in the fields of computer graphics, computer animation and computer simulation. They are able to describe a vast number of physical phenomena related to deformation of materials. Many models

can plausibly simulate elasticity, plasticity, elasto-viscosity, ductile [12] and brittle fracture [8] among others.
Generally, all deformation techniques have a similar concept. The deformable object is discretized into smaller entities. Different approaches name these entities differently but in this thesis I will refer to them as *particles*. The particles are used to simulate the effect of internal and external forces on the object. The specifics of the discretization as well as the process of handling external and internal forces and influences depend on the specific technique. In the following sections, a selection of the most prominent deformation methods is presented.

2.2.1 Physically Based Methods

The use of a physically based approach for the animation of deformable materials has been proposed by Terzopoulos in the late 1980s [1][3][2]. Terzopoulos and his colleagues started off a development in computer animation, which has yielded a large body of research and is still actively researched on today. These methods manipulate objects by utilizing Newton's second law of motion. Forces, that are applied to an objects particles influence their acceleration and velocity and thus their position directly. transforming the position of a particle then results in the deformation of the object.

2.2.1.1 Mass-Spring System

The mass-spring model is an early and simple model for deformation [5][7]. It still is a common method to create simple simulations for deformable materials. The model handles the deformable object as a set of particles which are connected via springs. Material characteristics can be achieved by adjusting the spring stiffness and by adjusting the weights of the particles. This model is very lightweight, which makes it intuitive, easy to implement and computationally fast. It does however have a number of significant drawbacks in terms of the accuracy and the ease of reproducing the desired material behaviour. The way the spring network is setup greatly influences the deformation behaviour of the object. Furthermore, achieving the desired material behaviour can be hard, since the spring stiffness is the only adjustable parameter in the model. Thus, certain specific material characteristics may not be achievable. Volumetric effects are very difficult to model with a mass-spring system as well. While adjusting the spring layout can help to model the volume of an object, more complex effects like conservation of volume are not recreatable.

2.2.1.2 Finite Element Method

The finite element method (FEM) is the most commonly used representative of a family of deformation models: the continuum models. In contrast to the mass-spring model, the continuum models handle the object as a continuum with masses, forces and energies distributed throughout. This approach is mathematically more complex but also more physically accurate as it is closer to reality. In order to compute the deformations on an object, it gets discretized into finite, non-overlapping

elements, over which the continuum gets approximated. The object in its rest configuration is considered as an equilibrium, which is acted on by external forces.
In order to find the deformed configuration of the object, the total energy equation must be minimized. This is achieved by solving the resulting partial differential equations (PDE). As the material properties and different forces become more sophisticated, the resulting PDEs becomes more and more complex. The FEM offers a numerical solution to said PDEs. To simplify the PDEs and thus speed up the computations, boundary conditions can be defined which are derived from domain knowledge. They restrict the number of variables and thus speed up the computations. Therefore, to model a given object, all material properties and boundary conditions are required to apply the FEM in an efficient way.
The FEM is very flexible and can be used on arbitrary object shapes but it is especially strong for complex shapes [28]. It can also conquer the above explained shortcomings of a mass-spring system. Furthermore, it can handle volumetric effects and the material characteristics are not significantly dependant on the layout of the discretization of the object. The high computation cost makes it most suitable for offline rather than real-time operations. However, lately there have been attempts to address this issue [9][11][17][23].

2.2.1.3 Finite Difference Method

The finite difference method (FDM) was proposed by Terzopoulost and colleagues [1] when they first suggested the use of physical models to animate deformable bodies. Is is also a continuum model like the FEM, however using the FDM, a deformable object is discretized into surfaces separated by nodal points, rather than volumetric elements.
Basloom [28] states that this method is inferior to the FEM both in terms of its efficiency and in the precision of its approximation. This is also reflected in the fact that here has been almost no research expanding on the method proposed in [1] within the field of computer graphics. Instead, a large set of research has been published utilizing and expanding the FEM, furthermore suggesting that the FDM is truly inferior.

2.2.1.4 Finite Volume Method

The finite volume method (FVM) discretizes the object into a finite set of elements. Some researchers argue, that the method is more suited for 3D purposes than the FEM [28]. Teran et al. [15] state that FVM is computationally cheaper, while sacrificing some visual detail. They argue however, that in a computer graphics context, the ability to simulate a large amount of elements is more valuable than achieving higher realism for a smaller amount of elements.

2.2.2 Position-Based Methods

A serial algorithm for position based dynamics was first introduced to rigidbody and deformation simulation in 2007 [21] which has later been adapted for parallel computations in 2014 [25]. In contrast to physics-based methods, position-based

methods omit the velocity and acceleration layer to directly manipulate the particle position. A particle is purely formalized as point-masses with a position. This is the biggest difference to physics based methods. It allows more directly control on the deformation behaviour, as particle positions are adjusted directly, rather than indirectly through forces. The total potential energy within an object is not reduced via forces, but rather in a position-based energy reduction[25]. Particle constraints restrict the particle positions in relation to other particles or external influences. Satisfying the constraints is equivalent to an energy reduction in other approaches. The position-based approach was designed for real-time and interactive applications. It is computationally less expensive than physics-based methods and allows more control. These advantages emerge from a more lightweight physics integration as well as a commitment to sacrificing physical correctness for better computational performance.

3

Theory

3.1 Deformation

Deformation describes change in the shape, size or volume of an object as a result of a set of external and internal stresses. Typically, an object under stress will deform in two ways. If the stress is below a certain limit, the deformation will be elastic. An elastically deformed object will return back to its original form after the stress is removed. If the stress is above a certain limit, the objects deformation is permanent, even after the removal of the stress. The maximum stress an object can take before deforming plastically is called the elastic limit. Deformations in reality are extremely complex, as they are results of inter-atomic relations. Modelling this behaviour in detail is not suitable for a real-time context. Additionally, deformations are not perfectly elastic or plastic in reality and a number of more complex deformation behaviour has been observed. As a result, deformations in this thesis are represented by a combination of elastic and plastic deformations. This approach is further explained in section 3.8

3.2 A Simplified Car Model

A car consists of hundreds of parts, many of which are produced from different materials to benefit the individual parts' specific purpose. Some parts are light and flexible, others heavy and rigid. Modelling and simulating each individual part would be computationally expensive, time intensive to implement and most importantly not required for visual plausibility. As specified in section 1.2 the car parts modelled as deformable are merely the exterior metal panelling as well as the structural frame of the bodyworks. The glass of the windows, plastic of the headlights, rubber of the wheels and the interior will not be modeled in a physically plausible manner as it would be outside the scope of this thesis. The focus rather lies on developing a prototype that can simulate the deformation of the car exterior in a way that is suitable for real-time interactive applications.

The bodies of modern cars are designed to deform in certain ways to offer maximum protection and security to the passengers in case of a crash. The front and back zones of cars serve as crumple zones to absorb collision energy from frontal impacts. A structural frame, made from steel or other sturdy materials is built to enlarge the surface of the impact on both frontal impacts and side impacts, to evenly distribute load and thus reduce local strain. While implementing the behaviour of said structural parts could bring benefits for the visual results of the deformation simulation

with regards to realism, the focus of the thesis lies on the metal panelling itself. The entire exterior of the car is thus modelled as a material with a constant rigidity. A possible approach on implementing the structurally more rigid parts is examined in section 6.4.

3.3 Choosing the Deformation Method

In section 2.2 the research body of physically based and position-based deformation techniques was explored. The strengths and weaknesses of each technique are now highlighted and related to the requirements, presented in section 1.3. Finally, a method for the development of the prototype of this thesis is chosen.

3.3.1 The Mass-Spring Model

The mass-spring model is a classic approach to deformation modeling. It is real-time applicable and relatively easy to implement. However, it can become unstable in certain conditions. It also has the weakness of being dependant on the layout of the mass-spring network, while having only the spring stiffness parameter to model all of the desired material properties. Volumetric and angular behaviour is complicated to model due to the fundamental inner workings of the mass-spring model. Said behaviours have to be implicitly defined by designing specific mass-spring networks. This inability to accurately and reliably model material behaviour makes the mass-spring model unsuited for the purpose of the thesis.

3.3.2 Physically-Based Continuum Methods

Out of the three physically based continuum methods FEM, FDM and FVM, the FEM is by far the most dominant one in the field of computer animation. The FEM has gotten far more attention in the research over the past decades than the other methods. It offers great flexibility and physical accuracy, while being more efficient than FDM and FVM with arbitrary 3D meshes [28]. There have also been multiple explorations into utilizing an FEM in the simulation of car crashes, such as in [30]. The computational cost for all three methods is unfortunately high. However, there have been multiple attempts to implement the FEM in a real-time domain as expressed in section 2.2.1.2. The volumetric nature of the FEM could be used to model the different parts of the car. The outer layer of the model can have softer properties than the inner layers. This would allow to model the soft metal paneling in combination with the more rigid structural frame. This is similar to an approach proposed by Müller et al. in [11]. The disadvantage here would be that the three-dimensional representation of the object is quite different than the representation of a typical 3D mesh, which is a set of surfaces rather then volumes. This complicates mapping from one to the other, both when spatially discretizing the object, and when applying changes in the deformable model to the 3D mesh. It is still a viable method that was ruled out merely due to the position-based methodology being more suitable.

3.3.3 Position-Based Deformation

The position-based approach is specifically designed for real-time applications. The method is fast, stable and controllable[25], while offering the ability to model complex physical material characteristics. It can handle all material behaviour that physically based methods can handle, while sacrificing some visual detail for better computational performance. It is also relatively simple to implement. The fact that the method works purely on the positions of the particles by omitting the velocity and acceleration layers also proved very convenient when implementing a position-based system as an add-on to a physics engine. All of these advantages make the approach extremely well suited for the purpose of this thesis.

3.4 Surface Mesh and Tetrahedral Mesh

Two types of meshes are frequently referred to in this thesis. They both represent objects in three-dimensional space, but they offer different advantages and disadvantages.

3.4.1 Surface Mesh

Surface meshes represent the object by describing the objects surface. They are also called *3D models* or *render meshes*. The surface is made up of vertices which are connected via faces. Surface meshes are essentially hollow and have therefore no volumetric properties. They are only used to visually represent an object and to render it onto the screen.
The thesis aims to enhance a driving game environment with deformable bodyworks. The car model, that was used as the base of the deformation methods is referred to as the *original surface mesh* in this thesis.

3.4.2 Tetrahedral Mesh

In order to physically influence a virtual object in a detailed manner, a surface mesh is not enough. The object needs to be represented by a volumetric mesh, i.e. a mesh that not only accounts for the surface of the object but the entire volume of it. The simplest such volumetric mesh is a tetrahedral mesh, a mesh that discretizes the volume of an object into perfectly tessellated tetrahedra. This method of splitting up a volume into a set of tetrahedra works nicely with position-based deformation techniques. The corners of the tetrahedra make up the particles while other geometrical properties of the tetrahedra can be used to generate the constraints. Figure 3.1 illustrates the difference between a triangular surface mesh and a tetrahedral volume mesh.

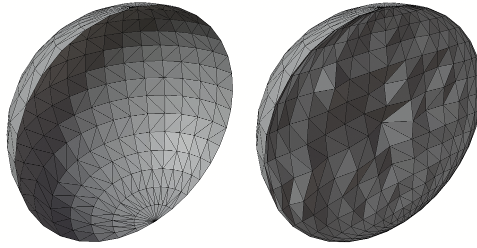

Figure 3.1: The cross-sections of a sphere as a surface mesh (left) and a tetrahedral mesh (right). Notice, that the surface mesh is hollow, while the the surfaces of the inner tetrahedra of the tetrahedral mesh are visible.

3.5 Barycentric coordinate system

3D models in game environments are surface meshes. They are the only meshes that can be rendered by default and are the industry standard. However, modelling deformation behaviour using position-based deformation requires a volumetric, tetrahedral mesh. The tetrahedral mesh is used for a deformation simulation and the surface mesh for rendering. This yields a need for a mapping algorithm, that relates the vertices of the surface mesh to the tetrahedral mesh. Barycentric coordinates can be used to create such algorithm. With barycentric mapping, each point in the surface mesh can be mapped to a tetrahedron in the tetrahedral mesh. When tetrahedra are deformed during a collision, the surface mesh can be changed accordingly to show that deformation on screen.
The barycentric coordinate system is a method of representing points in space. Typically, we use the Cartesian coordinates system, where points are represented in relation to the axes of a coordinate system. Barycentric coordinates however, describe points in relation to the points of a simplex.

A point $\mathbf{p}$ with barycentric coordinates $\mathbf{b}$ is the center of mass of a tetrahedron T. T is comprised of the four vertices $\mathbf{v}_1 - \mathbf{v}_4$ with their respective masses $m_1 - m_4$. The barycentric coordinate component b_i is

$$b_i = m_i/m_T$$

where

$$m_T = T = m_1 + m_2 + m_3 + m_4 \text{ .. the total weight of tetrahedron}$$

Each barycentric coordinate component b_i therefore describes the influence each point $\mathbf{v}_i$ has on point $\mathbf{p}$. Increasing the mass m_i of a vertex $\mathbf{v}_i$ moves the center of mass closer to $\mathbf{v}_i$. This will in turn increase the volume V_i of the opposing sub-tetrahedron that is created by $\mathbf{p}$ and the remaining three vertices. In fact, the proportions of the individual opposite sub-volumes is the same as the proportions of the vertex masses. Figure 3.2 illustrates the principle of the opposite sub-tetrahedra in 2D.

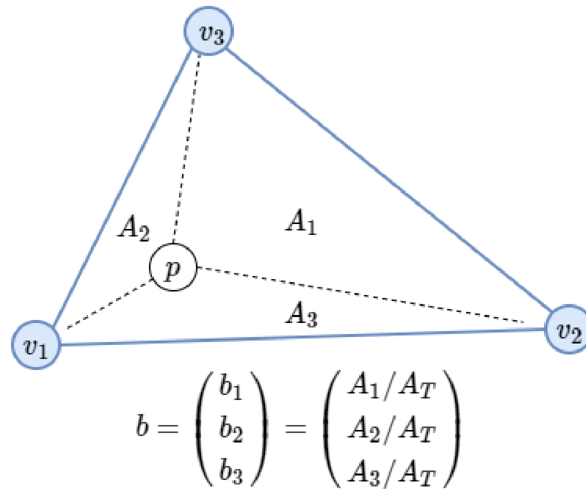

Figure 3.2: Barycentric mapping in 2D. Point $\mathbf{p}$ is mapped to the triangle T. The barycentric coordinate $\mathbf{b}$ is the result the quotient of the area of each sub-triangle A_i and the total area of the triangle A_T.

From this circumstance we can derive a fast method of calculating the barycentric coordinates $\mathbf{b}$ using the volumes of the sub-tetrahedra that are comprised of $\mathbf{p}$ and three vertices. The volume V_T of tetrahedron T is

$$\frac{1}{6} \det D_T \begin{pmatrix} v_{1,x} & v_{1,y} & v_{1,z} & 1 \\ v_{2,x} & v_{2,y} & v_{2,z} & 1 \\ v_{3,x} & v_{3,y} & v_{3,z} & 1 \\ v_{4,x} & v_{4,y} & v_{4,z} & 1 \end{pmatrix}$$

To get the volume V_i of the sub-tetrahedron opposite of vertex $\mathbf{v}_i$ we replace the ith row of D_T by $\mathbf{p}$. For example, for V_2 this results in

$$\frac{1}{6} \det D_2 \begin{pmatrix} v_{1,x} & v_{1,y} & v_{1,z} & 1 \\ p_x & p_y & p_z & 1 \\ v_{3,x} & v_{3,y} & v_{3,z} & 1 \\ v_{4,x} & v_{4,y} & v_{4,z} & 1 \end{pmatrix}$$

The barycentric coordinate $\mathbf{b}$ is therefore

$$\mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} = \begin{pmatrix} V_1/V_T \\ V_2/V_T \\ V_3/V_T \\ V_4/V_T \end{pmatrix} = \begin{pmatrix} D_1/D_T \\ D_2/D_T \\ D_3/D_T \\ D_4/D_T \end{pmatrix}$$

To get point $\mathbf{p}$ from a given tetrahedron T and barycentric coordinates $\mathbf{b}$ we use simply

$$\mathbf{p} = \mathbf{b}_1\mathbf{v}_1 + \mathbf{b}_2\mathbf{v}_2 + \mathbf{b}_3\mathbf{v}_3 + \mathbf{b}_4\mathbf{v}_4$$

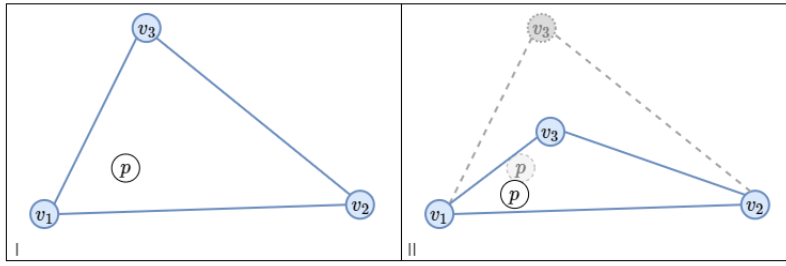

Figure 3.3: A point $\mathbf{p}$ is barycentrically mapped to a triangle (I). As the triangle deforms in (II), the position of $\mathbf{p}$ is adjusted.

3.6 Constraints

Constraints are functions that restrict the positions of groups of particles. Solving the constraints means that all particles are moved such that all constraints are satisfied. The different kinds of constraint functions are manifold and are used to model all behaviour of objects that are deformed via position-based methods. A constraint $C(\mathbf{x}_i, ..., \mathbf{x}_n) = a_0$ restricts a set of position vectors $\mathbf{x}_i, ..., \mathbf{x}_n$ of the particles i. A function of the particles has to equal the given rest value a_0. The way the function is defined and how the rest value is formalized needs to be defined for each type of constraint. Solving a constraint results in a set of displacement vectors $\Delta\mathbf{x}_i$ (commonly referred to as deltas) which move the involved particles such that they satisfy the constraint. In the following sections, constraint types that are relevant to this thesis and their resulting displacement vectors are presented.

3.6.1 Distance constraints

Distance constraint [14] C restricts two particle positions $\mathbf{x}_1$ and $\mathbf{x}_2$ in their distance to one another such that

$$C(\mathbf{x}_1, \mathbf{x}_2) = |\mathbf{x}_{2,1}| = D_0$$

where

$$\begin{aligned} |\mathbf{x}_{2,1}| &= \text{length of } \mathbf{x}_2 - \mathbf{x}_1 \\ D_0 &= \text{the resting distance} \end{aligned}$$

To satisfy the constraint, the two particles have to be either pulled apart or pushed together until their distance equals D_0. The resulting displacement vectors are

$$\Delta\mathbf{x}_1 = -\frac{1}{2}\ (|\mathbf{x}_{2,1}| - D_0)\ \mathbf{n}$$

$$\Delta\mathbf{x}_2 = +\frac{1}{2}\ (|\mathbf{x}_{2,1}| - D_0)\ \mathbf{n}$$

where

$$\mathbf{n} = \frac{\mathbf{x}_{2,1}}{|\mathbf{x}_{2,1}|} \text{ i.e. the normalized vector } \mathbf{x}_{2,1}$$

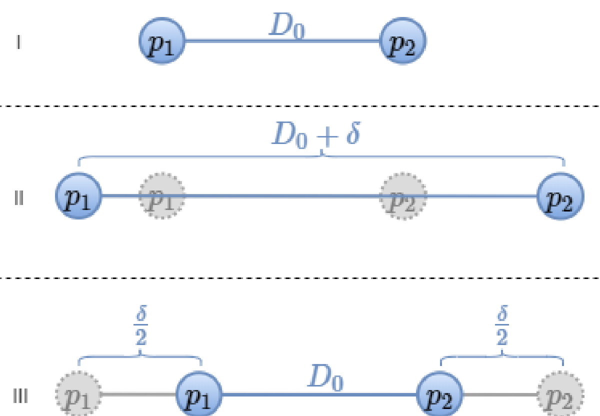

Figure 3.4: A distance constraint between particles $\mathbf{p}_1$ and $\mathbf{p}_2$ (I). When the particles are moved apart in (II), the distance is no longer equal to the resting distance D_0 and the constraint is offended. Both particles are adjusted by half the divergence δ in order to satisfy the constraint (III).

3.6.2 Volume constraints

Volume constraint C restricts the volume of a tetrahedron between the four particle positions $\mathbf{x}_1$,$\mathbf{x}_2$,$\mathbf{x}_3$,$\mathbf{x}_4$ with

$$C(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4) = \frac{1}{6}(\mathbf{x}_{2,1} \times \mathbf{x}_{3,1}) \cdot \mathbf{x}_{4,1} = V_0$$

where

$$\begin{aligned} \mathbf{x}_{2,1} &= \mathbf{x}_2 - \mathbf{x}_1 \\ \mathbf{x}_{3,1} &= \mathbf{x}_3 - \mathbf{x}_1 \\ \mathbf{x}_{4,1} &= \mathbf{x}_4 - \mathbf{x}_1 \\ V_0 &= \text{resting volume} \end{aligned}$$

Similarly to the distance constraint, the displacement vectors are the result of the difference between the current and the desired volume of the tetrahedron. Particles are either moved inwards or outwards in order to adjust the volume. The direction of the displacement vector $\Delta\mathbf{x}$ of a point $\mathbf{x}$ is determined by the opposing faces normal. The magnitude of each displacement is the result of the difference between the current volume V_1 and the desired volume V_0 and the total squared sum of all displacement vector directions.
For position x_1, the displacement vector is

$$\Delta\mathbf{x}_1 = d_1 * (V_1 - V_0) / \sum d_i^2$$

where

$$\begin{aligned} d_1 &= \mathbf{x}_{4,2} \times \mathbf{x}_{4,3} \text{ .. direction of displacement vector} \\ \sum d_i &= |d_1|^2 + |d_2|^2 + |d_3|^2 + |d_4|^2 \text{ .. squared sum of direction magnitudes} \end{aligned}$$

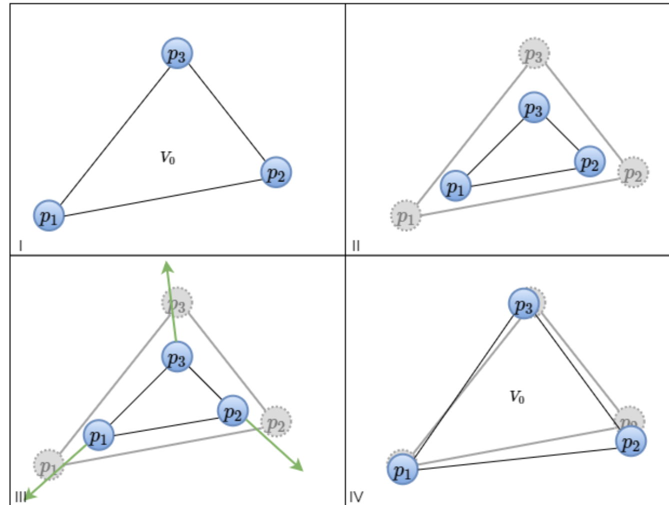

Figure 3.5: Tetrahedral volume preservation via a volume constraint. For clarity, the process is shown in 2D. The volume constraint with resting volume V_0 acts on triangle $\triangle p_1, p_2, p_3$ (I). The triangle is deformed in (II). (III) shows the adjusting vectors (amplified for clarity). The vertex positions are adjusted and the triangles volume is preserved in (IV). Notice that the shape is not preserved.

3.6.3 Constraint Stiffness

Controlling the influence of each type of constraint is important to allow more control over the material behaviour. There are multiple way to adjust the strength of a given constraint, but a very common one is to apply a constraint stiffness. The stiffer a constraint, the stronger their influence on the material behaviour. To apply a stiffness parameter to a constraint we simply multiply the adjustment vector by a stiffness parameter before adjusting the vector position.

$$\mathbf{x}_i = \mathbf{x}_i + \Delta\mathbf{x}_i * s$$

where stiffness $s[0..1]$.

3.6.4 Collision constraints

The last constraint that is used in this thesis is the collision constraint, which is a unilateral (inequality) constraint. This means that it does not specify a geometric value which the particles have to satisfy but rather a general area in space in which particles can not penetrate. Collision constraints represent obstacles in the virtual space which particles can collide with. A mathematical formalization of the displacement vector is difficult, as it depends on the shape of the obstacle and the collision scheme. A common solution for satisfying collision constraints is to project them orthogonally onto the surface of the colliding obstacle [14]. Figure 3.6 depicts the particle projection.

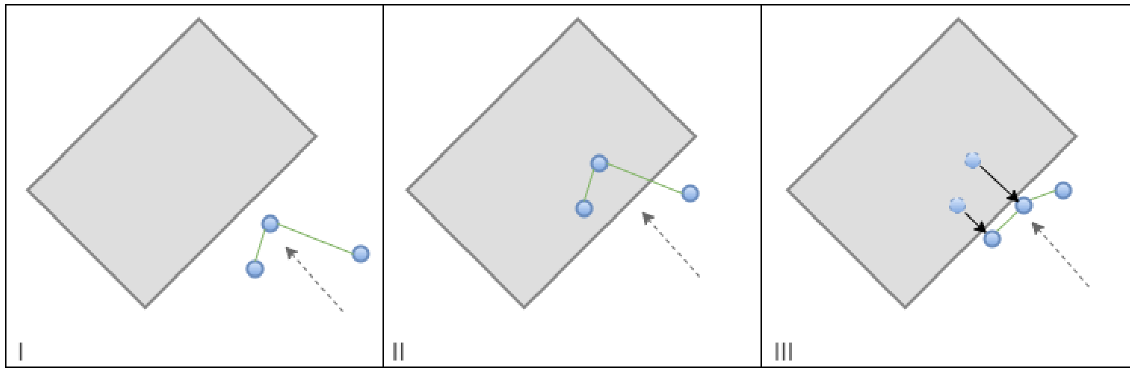

Figure 3.6: Projection of collision constraints. The particles (blue) move towards the obstacle (gray) and penetrate it in (II). They are then orthogonally projected onto the closest surface of the obstacle in (III).

3.7 Solver

The solver is an algorithm that handles the constraint resolution. Solving a large amount of different constraints is by no means trivial. Generally, a solver is a method to find the solution of a system of equations. Specifically in this context, a solver takes sets of functions, such as constraints and iteratively solves them in order to converge to an optimal solution. Typically, a solver needs to do 3-10 iterations over the set of constraints to converge to a sufficient solution. The scheme the solver follows determines how quickly the solver converges, how computationally expensive it is and whether or not it is stable (i.e. whether it actually finds a feasible solution). In this thesis two solvers were implemented. The Gauss-Seidel Solver and the Jacobi-solver. They will be described in the following sections.

3.7.1 Gauss-Seidel Solver

The most common solver is the Gauss-Seidel solver. It iterates over all constraints and solves them one-by-one. The deltas for the vertices that are influenced by a constraint are computed (algorithm 1, line 4) and immediately applied to the vertex position (line 5). This method has a rather quick convergence time but has one crucial flaw. It is inherently serial and thus can not be paralallized in code [29]. The Gauss-Seidel method does therefore not take advantage of modern hardware capabilities. While there are many attempts to parallelize Gauss-Seidel algorithms, they require large overheads in synchronization structures, which make them inapplicable for real-time contexts [33].

3.7.2 Jacobi Solver

The Jacobi solver works similarly to the Gauss-Seidel solver, while using an approach that allows the parallelisation of the execution code. Instead of computing and immediately applying the vertex displacement vectors succesively, the deltas per vertex are summed up to a total vertex delta variable (line 5 - line 10). After the

Algorithm 1 Gauss-Seidel Solver

```
1: procedure SolveGaussSeidel
2:     for all constraints c_i do
3:         for all vertices of c_i do
4:             compute v_j.delta
5:             v_j.position += v_j.delta
6:         end for
7:     end for
8: end procedure
```

vertex deltas are computed, the average vertex deltas are applied to each vertex position (lines 11 - 13). The average is computed via the total vertex delta and the number of constraints that influence this vertex. This requires an initial step that counts and stores the amount of influencing constraints per vertex (lines 2 - 4).

Algorithm 2 Jacobi Solver

```
1: procedure SolveJacobi
2:     for all vertices v_j do
3:         SET constraint_count_j
4:     end for
5:     for all constraints c_i do
6:         for all vertices of c_i do
7:             compute delta
8:             vertex_j.delta += delta
9:         end for
10:    end for
11:    for all vertices v_j do
12:        vertex_j.position += vertex_j.delta/constraint_count_j
13:    end for
14: end procedure
```

3.8 Plasticity

Using the above formalizations will produce a completely elastic deformation. The rest state for each constraint is never changed, which will drive the system towards its initial state. Deformations of real materials are however not purely elastic. They feature both elastic and plastic components. This behaviour can be modeled in the constraint solution step. After a deformation occurs and the rest value a_0 of the constraint is offended, the rest value gets adjusted. The adjustment is done by using a plasticity scalar p to linearly interpolate between the previous rest value a_0 and the current state a_1. This produces

$$C(x_1, ...x_n) = a_0 \cdot p + (1 - p) \cdot a_1.$$

The precise value of p and the interpolation function are not fixed and can be adjusted to model a specific material behaviour.

3.9 Position-Based Dynamics

In position-based dynamics, the above concepts are united. The deformable object is represented as a mesh of particles. These particles are connected via constraints, which are used to model material behaviour. PBD served as the basis for the approach used in this thesis. It was adjusted to better fit the tools and software used for the prototype development and the restrictions that come with those. How and why said adjustments were made is outlined in chapter 5. Position-based dynamics, as originally proposed by Müller et al. [20] works in three steps (see algorithm 3).
Position Prediction. After the initialization (lines 1 - 3) the positions $\mathbf{p}_i$ are predicted for the particles i (lines 5 - 8). These predicted positions are not applied to the particles directly, but are rather used as a base for the following step.
Constraint Generation and Solving. After the initialization, the temporary collision constraints (line 10) are generated and all constraints are solved using the previously predicted positions (lines 11 - 13) via a solver. The particle positions are modified so that they satisfy all constraints as precisely as possible using the solver.
Particle Update. Finally, the actual positions and velocities of the particles are updated (lines 14 - 17) in order to update the particle mesh for the current time step.

Algorithm 3 Position Based Dynamics [20]

1: **for all** vertices i **do**
2: initialize $\mathbf{x}_i = \mathbf{x}_i^0, \mathbf{v}_i \ \mathbf{v}_i^0, w_i = 1/m_i$
3: **end for**
4: **loop**
5: **for all** vertices i **do** $\mathbf{v}_i \leftarrow \mathbf{v}_i + w_i \mathbf{f}_{ext}(x_i)\Delta t$
6: **end for**
7: **for all** vertices i **do** $\mathbf{p}_i \leftarrow \mathbf{x}_i + \mathbf{v}_i \Delta t$
8: **end for**
9: **for all** vertices i **do** generateCollisionConstraints
10: **end for**
11: **loop** iterationCount **times**
12: solveConstraints$(C_1, ..., C_{M+M_{Coll}})$
13: **end loop**
14: **for all** vertices i **do**
15: $\mathbf{v}_i \leftarrow (\mathbf{p}_i - \mathbf{x}_i)/\Delta t$
16: $\mathbf{x}_i \leftarrow \mathbf{p}_i$
17: **end for**
18: **end loop**

4

Methodology

This chapter describes the approach taken to achieve the aim of the thesis. The goal of developing a prototype of a real-time deformation system for car bodies, comes with a set of requirements on the software development process. This asks for an appropriate software development philosophy, which is presented in this chapter. Additionally, the third-party tools and software used in the development of this thesis are presented and motivated.

4.1 Requirements

In the following sections two types of requirements are used to explain challenges faced in the thesis process. These two requirements will be called *prototype requirements* and *software development requirements*. Prototype requirements are what are commonly called software requirements. They describe the requirements on features and functionality of the prototype, among others. These requirements are described in detail in section 1.3. Software development requirements are a set of requirements on the software development process itself. They are a result of both the prototype requirements and the development environment of this thesis.
Many of the questions answered in this thesis were open-ended at the beginning. Specific prototype requirements were therefore difficult to define initially. Furthermore, discoveries throughout the prototype development were expected to modify existing prototype requirements or add new ones. This imposed a set of requirements on the prototype development process. The prototype development process needed to be designed to allow an ever-changing environment with a set of changing prototype requirements.

4.2 Development Process

The nature of this thesis demanded to allow new discoveries to alter existing prototype requirements. A software development philosophy that is known for its adaptivity is agile software development. There are various approaches grouped by agile methodology and most focus on similar philosophies. Agile software development methods are aiming to adjust software development to a quickly changing environment. This is done by having small teams produce granular improvements to the software in a rapidly iterative fashion. The software is frequently released and customer feedback is immediately addressed within the following development cycles. While the team-oriented nature of agile methodology could not be respected due

to this thesis being a solo-project, many other influences have been implemented. To make use of agile methodologies a step was taken at the start of the development. The goal was to create a minimally functional prototype. All features were represented by simplified versions of themselves. This allowed for feature-by-feature improvements of the prototype in an agile manner. The advantage of following this approach was that effects of a new feature were immediately visible and newly emerged requirements could be tracked. This enabled continuous code testing and continuous integration into the final prototype, which are both pillars of the agile methodology. A backlog was kept to keep track of features as well as old and newly emerged requirements. Implementing a quasi-agile philosophy proved useful as the prototype could be gradually improved while new discoveries were tracked along the way.

4.3 Tools

Throughout the prototype development, a set of software and tools was used. Some provided important functionality to the final prototype while others enabled an efficient workflow and an organized process. These tools are now presented and motivated.

4.3.1 Tools for Agile Development

To effectively implement a quasi-agile software development process, a set of tools was required. *Trello*[24] was used to keep a backlog of desired features and requirements which was ordered by urgency. *GitHub* was used to version-control code [22], which allowed for tracking the progression of features, tracing issues and errors as well as enabling an easy option to deploy the prototype onto different systems. The last point was especially important for collecting the run-time data and statistics of the prototype that are presented in chapter 6.

4.3.2 Functionality Tools

In order to implement all the features that were required for the prototype, a few additional tools were needed. One of the requirements of the prototype was, that it could enhance a game environment with deformable car bodies. This implies that existing car models need to be made accessible for PBD. For that purpose, 3D models which are typically triangular or rectangular surface meshes, needed to be converted into volumetric tetrahedral meshes. This task is by no means trivial, which motivated the use of *TetWild* [34], a software which generates tetrahedral meshes from surface meshes. *TetWild* has certain requirements on the surface meshes, which generated a need to modify them. This was done using *Blender*, an open-source 3D modelling software.
Writing code that was efficient and fast, called for two third party libraries. *OpenGL Mathematics* (glm) [38] is a library that provides many functions and classes for 3D geometry as well as matrix and vector manipulation. *Intel ®Threading Building*

Blocks was used to parallelize code and thus make use of all available cores of the CPU.

4.3.3 Choosing a Physics Engine

The prototype was developed using a physics engine. Physics engines provide features that could significantly speed up the initialization of the prototype development. Most engines provide physics and rendering frameworks, that could be build upon as well as having user interface frameworks. These features were essential in the early prototype development as it meant a relatively quick start up time. There is a vast amount of physics engines on the market which could assist the prototype development.
Four different options were tested: *Bullet* [32], *Open Dynamics Engine* [27], *PhysX* [31] and *Unity* [19]. Each test involved creating a minimal prototype that resembled the plan for the thesis' final prototype. The tests were intended to explore the engines different features and properties. The most crucial evaluated points were:

- release under permissive license
- ease of use
- amount of online learning and help resources
- amount of required and optional features
- flexibility of the physics framework

The final point was especially important, as most physics engines do not explicitly support custom deformation for models. The engine therefore needed to be flexible enough to allow the creation of a custom extension to the physics framework. After the test were absolved for each option, the decision was made in favor of *Unity*. This was made mostly due to three points. *Unity* has a large online community, with help and discussion forums and tutorials. Also I have experience using Unity, which allowed for a smooth workflow right away, as well as removing much of the slow learning process. *Unity* also provides many features the other three engines do not and has a graphical interface. All these properties enabled a very quick initialization step. Many features that were required as a base for the prototype already existed and non-existing ones could be implemented quickly due to my prior experience.
Unity had one disadvantage however. It is programmed using C#, a rather slow programming language. This would significantly slow down the deformation computations to a point where they would no longer be sufficient for real-time applications. The solution to this was to develop a library in C++ (DLL) for handling the deformation computations. C++ can achieve much better performance than C# and is used by most other physics engines, that do not use proprietary languages. This solution proved beneficial later on, as it enabled the use of all *Unity* features, like the user input, rendering and rigid body physics capabilities, while the DLL provided the fast deformation computations.

4.3.4 *Unity*

Unity provides many features that were used in this thesis. It is therefore necessary to present how *Unity* functions internally and what features it provides.

The base object in *Unity* is the so called *GameObject*. Every object in *Unity* is a *GameObject* at its core, but *GameObject*s themselves do not provide any inherent behaviour. Components can be attached to *GameObject*s and enhance them with behaviour. By default, each *GameObject* has a transform component, which controls the position, rotation and scale of the object in 3D space, but components can handle much more. *GameObject*s can be nested, so that the child objects inherits some component properties of the parent object. They are extremely powerful and contain all functionality of a game. Among others, they are used to generate sounds, handle user input, control artificial intelligence, rendering and animations. For a complete list of available features, the reader is referred to their online user manual [41]. The user can write custom components, which are called scripts. Scripts are programmed with C# or JavaScript and are used to provide any features that go beyond the features provided by the default components.
Scripts in *Unity* follow a specific order of execution which is determined by *Unity*s core game loop. Event functions are called each frame by *Unity* and trigger component behaviour. By implementing event functions new behaviour can be added to a given script, which is executed at a certain time each frame. For instance, implementing collision events allows a script to react to a collision that is registered internally by *Unity*. A simplified order of executions that is focused on the events relevant to this thesis is depicted in figure 4.1. For a complete overview, the interested reader is referred to the official web manual page on event functions [42].

Unity also provides many features that speed up the development of games and interactive programs, such as a intuitive user interface system, drag-and-drop imports of common 3D, audio and image files. DLLs can be linked easily and immediately used in custom components. Furthermore, *Unity*-projects can be easily built and exported to different platforms.

4.4 Testing Projects

The development of the prototype required the implementation of many interdependent algorithms in an agile manner. As multiple algorithms were often developed at the same time, the origins of bugs became hard to determine at times. This problem was furthered by the fact that the prototype featured large amounts of constraints and forces. To individually assess and test algorithms in an isolated manner, test projects were created. These projects featured minimal, often hard coded data for the algorithms and omitted most other algorithms. For instance, an algorithm that solves constraints was initially implemented in a very primitive environment (see figure 4.2). A singular tetrahedron was created and the algorithms was tested. Once this functionality was as expected, the environment got more and more complex until it could be included in the actual prototype.

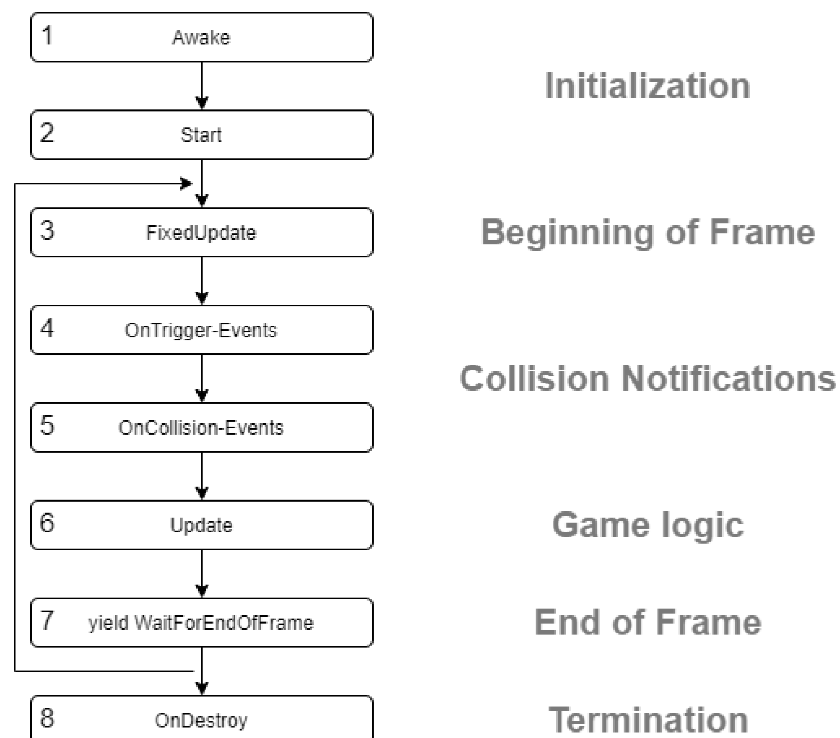

Figure 4.1: Simplified order of *Unity* events. Typically, a script is initialized by implementing *Awake*(1) or *Start*(2). Behaviour that is to be triggered each frame, such as reacting to user input is typically implemented via *FixedUpdate*(3) or *Update*(6). Collision handling behaviour uses the *OnTrigger*(4) or *OnCollision*-events(5).

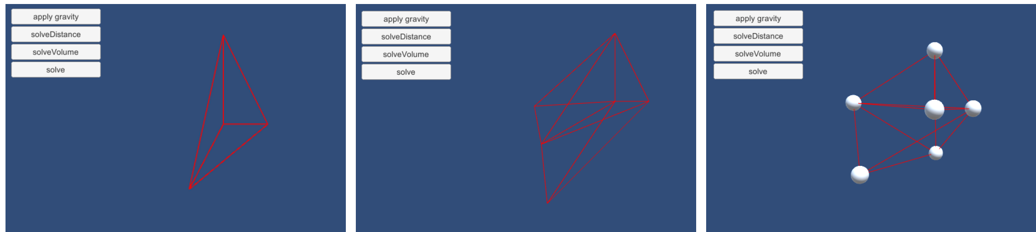

Figure 4.2: Three projects for testing a constraint solver in a progressively more complex environment. The first level of complexity ensures the correct functionality on a singular tetrahedron under gravity. The second level tests the algorithm on multiple connected tetrahedra under gravity. The third level lets the developer freely move the vertices and thus test the model under arbitrary forces.

4.5 Summary

In this chapter, the tools and philosophy for the software development process were described. The requirements that motivated the use of both the tools and the philosophy were outlined. Most notably, this thesis' context called for a flexible development process which could incorporate new findings into the existing process. This requirement on the development process motivated the use of a quasi-agile development methodology. *Trello* and *GitHub* were used to make the agile process more efficient and easier to track and validate. Additionally, the physics engine *Unity* was chosen, which was later used to implement the general game features that the prototype required. The choice for *Unity* was based mainly on *Unity*s wide range of features and on the prior experience I had with the engine. Both aspects allowed for a relatively fast start-up time with the project. *Unity*s internal events were used to control the flow of the game, which proofed as a stable run-time for the prototype. The *Unity* editor could additionally be used to set up the prototype scene as well as test projects, which were used to validate certain, complex algorithms.

5
Execution

The implementation of the prototype was the main part of this thesis and will be thoroughly described in this chapter. The prototype set-up and execution flow are outlined. The evolution and development of each step are explained in their respective sections.

5.1 Overview

The prototype can be separated into two parts. *Unity* serves as the run-time environment. The user loads the surface mesh and the tetrahedral mesh generated by *TetWild*. With these two meshes, the deformation module is initialized. All necessary data for future deformation computations is generated and sent to the second part: the deformation module. Constraints are generated, the surface mesh is barycentrically mapped to the tetrahedral mesh and the positions of all obstacles are sent to the deformation module.
After the initialization phase, the user can control the car via keyboard input. Each frame of the game loop, the position and rotation of the car is sent to the deformation module, which determines whether there are any intersections with obstacles in the scene. If this is the case, the deformation computations begin. Each particle of the tetrahedral mesh that collides with an obstacle is displaced. The solver now iteratively solves the constraints, which are aiming to return to their determined resting state. After the solver runs through a number of iterations, *Unity* queries the deformation module for the new positions of the surface vertices. The surface mesh gets adjusted and changes in the underlying tetrahedral mesh are visible. This loop repeats until the end of the run-time.

5.2 The Set-Up

Unity serves as the run-time environment. Figure 5.1 depicts the set-up of the *Unity* scene with the most relevant *GameObject*s and components. The deformable car is controlled by the attached *Car Controller* script. It reads user input and adjusts the gas and steering. The script also simulates the weight of the car, surface friction, turning radius and other physical properties. The *Car Controller* is part of *Unity*s standard assets pack[40]. The *Surface Mesh GameObject* has two important components. The *Mesh Filter* component contains all vertices and faces of the original surface mesh, while the *Mesh Renderer* component renders them onto the screen. Both components are part of the default *Unity* render pipeline. The car *GameObject*

also holds a set of colliders, which are used for the coarse collision handling layer (see section 5.5.1).
The *Dll Manager GameObject* holds a the *Dll Interface* script, which is used to send data to the deformation module, retrieve data from it and prompt deformation computations each frame. Each obstacle the car can collide with in the virtual world is represented by a *GameObject* which are grouped in the *Obstacles-GameObject*. The *Obstacle Manager* contains references to each obstacle and sends them to the *Dll Interface* script at the run-time initialization step. The data is then sent to the deformation module where it is stored and used for the collision handling.

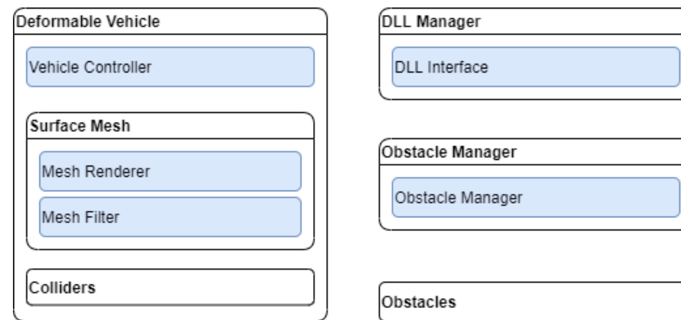

Figure 5.1: An overview of the scene hierarchy of the prototype. The most important *GameObject*s (black) are depicted with their attached components (blue) and child objects.

5.3 Execution flow

The *DLL Interface* script controls the flow of the run-time. Figure 5.2 presents a detailed flow chart with all important functions that are executed in the initialization step and each frame of the run-time. The generation of the tetrahedral mesh is not depicted in the chart, as it can take between a few minutes and multiple hours to fully generate a tetrahedral mesh, depending on the complexity of the mesh. This action should therefore be done before the prototype execution starts.
The prototype was implemented to integrate into *Unity* and thus implements *Unity*s event functions, which are recommended to be used to control the flow of a *Unity* project. This results in the execution flow being strongly influences by the *Unity* order of execution [42].
When the prototype starts, *Unity* internally imports all assets, including the original surface mesh (2). The *Awake* function (3) initiates the initialisation step. The deformation module is supplied with the data needed to prepare the module for the run-time. This data includes:

- the starting position and rotation of the car,
- the surface mesh of the car,
- the position, rotation and size of all obstacles,
- the file path to the tetrahedral mesh file, generated by *TetWild* and
- a set or parameters to control the deformation behaviour

After the deformation module received the data, the tetrahedral mesh is loaded from the specified file path (4) and the constraint generation is prompted on the base of the tetrahedral mesh (5). The mapping between the surface mesh and the tetrahedral mesh is generated(7).
The game loop is handled by *Unity* callbacks as well. The *OnTrigger*-events are used to register collisions between obstacles and the car each frame. The user input gets registered and applied in *Unity*s *Update* function (11) and the new position and rotation of the car are passed to the deformation module (12). The deformation module is then prompted to handle collisions (13), solve the constraints (14) and update the surface mesh (15).

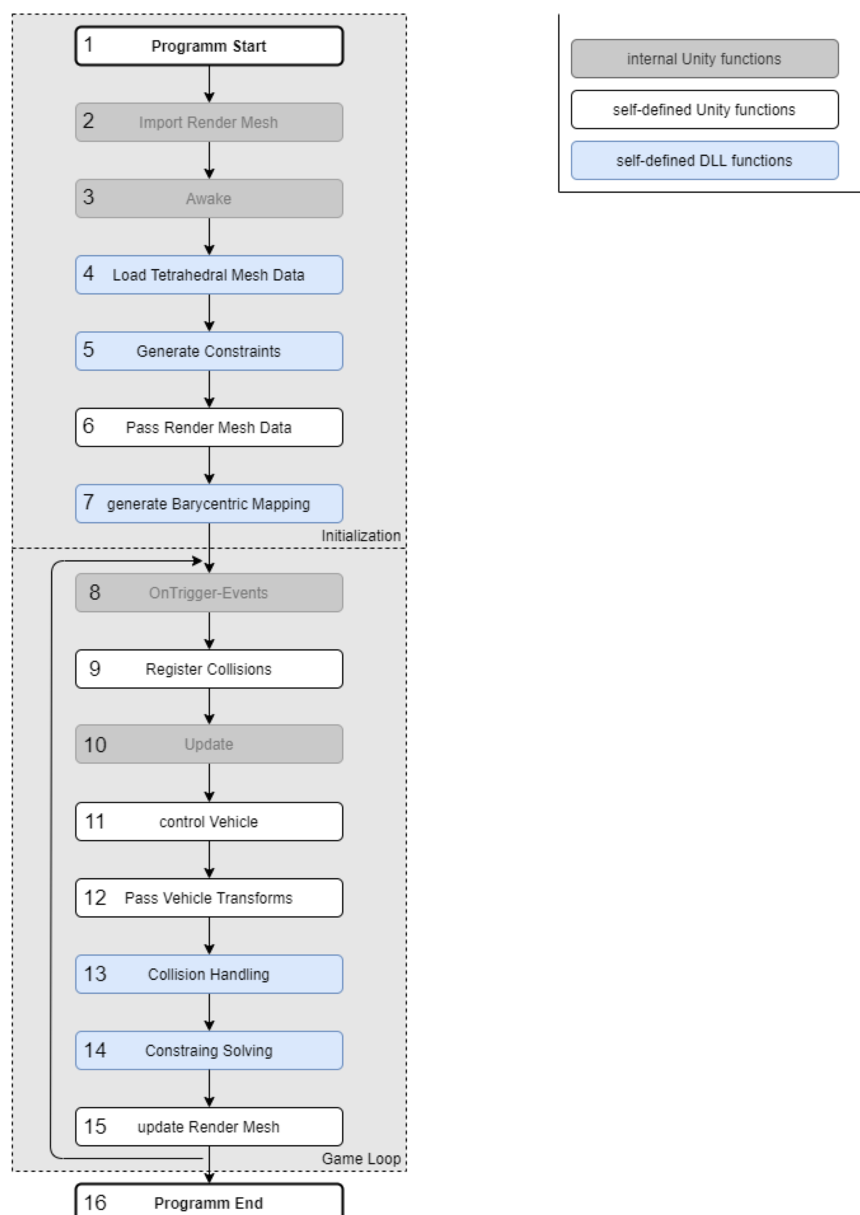

Figure 5.2: A flow diagram of the prototype. The diagram shows the functions of the initialization (2-7) and of the deformation loop (8-15).

5.4 Preparation and Initialization

This sections describes preparation steps before the prototype executes and steps at the beginning of the execution. The original surface mesh is prepared for being used in the prototype. Necessary data is generated, such as different meshes and constraints. These steps are summarized in the flow diagram 5.2 in steps (2)-(7).

For the deformation simulation, the program requires a surface mesh and a tetrahedral mesh. The surface mesh is the original model of the car that is to be used in the simulation. The tetrahedral mesh is later used for the deformation computations. To generate a tetrahedral mesh from the given triangular surface mesh, *TetWild*[35] was used. *TetWild* takes an .obj file (i.e. a surface mesh) and a set of parameters and generates a tetrahedral mesh as well as a triangular mesh of the surface of that tetrahedral mesh. The car models are taken from the web . *TetWild* imposes a set of constraints on the models as a result of its inner workings. Depending on how *TetWild*'s parameters are set up, a too detailed triangular model would either lose the detail in the process or result in a high increase in tetrahedralization time.
Generating an overly detailed tetrahedral mesh causes a significant slowdown of the deformation simulation later down the line, due to its high number of simulated vertices and constraints. In chapter 6, the effects of different mesh resolutions on the execution speed are investigated in detail.
TetWild demands a set of features for the input mesh in order to properly generate a tetrahedral mesh. The input mesh must be watertight, i.e. there can not be any holes in its surface. Additionally, the mesh must be strictly triangular, which can introduce issues depending on the software the mesh was created with. Often, modelling software is based on quad meshes, which can ease the modelling process and provide workflow benefits. Since freely available car models are generally not created with these constraints in mind, the models used in this thesis had to be modified by hand via *Blender*[6]. Unwanted detail was removed, holes were filled and quadratic faces had to be triangulated. These time consuming issues can mostly be eliminated in a production environment. 3D artists can produce watertight models with relative ease and triangulation of quad meshes can be automated before the tetrahedralization step, if necessary.

5.4.1 Importing mesh data from *TetWild*

TetWild generates the tetrahedral mesh and saves its vertices and tetrahedra in a file. Additionally, *TetWild* generates a triangular mesh, which contains all vertices and faces of the surface of the tetrahedral mesh. The prior is required as the core of the deformation algorithms, while the latter can be used to implement a simple method of visualization. This method will be expanded on in upcoming sections.

Neither of the files follow any established standard, which required a custom importer to extract the data from the files. The importer reads the files line-by-line and parses the data to entries of data structures.

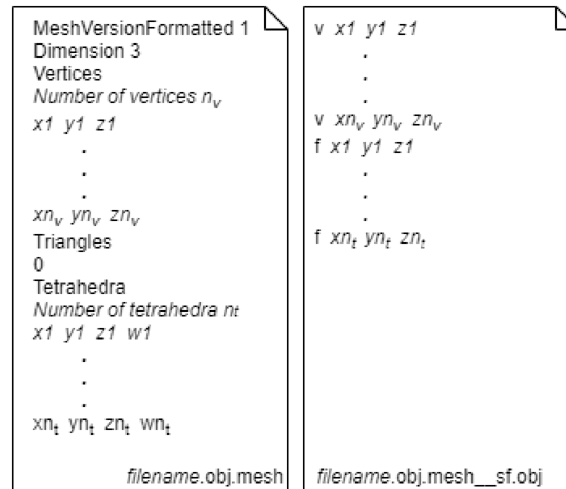

Figure 5.3: The format for the tetrahedral mesh file (left) and the surface mesh file (right).

5.4.2 Constraint Generation

The tetrahedral mesh is now represented in code by a set of 3D vertices and a set of tetrahedra which hold the indices of their respective four vertices. To simulate a deformation a set of constraints is required, which was generated on the basis of the tetrahedral mesh. Volume constraints represent the volume of each tetrahedron which allows one constraint to be generated per tetrahedron (algorithm 4, line 3). Distance constraints represent the four edges of each tetrahedron, i.e. the distances of the four vertices to each other. Since most edges are shared by adjacent tetrahedra, generating four distance constraints per tetrahedron results in a large amount of duplicates, which in turn results in unwanted deformation behaviour. Therefore, a duplicate check is required before adding a distance constraints to the final constraint set (lines 4-8).

Algorithm 4 Constraint Generation

```
 1: procedure GenerateConstaints(t)                    ▷ List t of tetrahedra
 2:     for all tetrahedra t_i do
 3:         generate volume constraint with volume of t_i
 4:         for all edges of t_i do
 5:             if distance constraint of edge does not exist then
 6:                 generate distance constraint
 7:             end if
 8:         end for
 9:     end for
10: end procedure
```

5.4.3 Surface Mesh to Tetrahedral Mesh Mapping

The final step of the initialization is generating a mapping between the tetrahedral mesh and the triangular render mesh. The deformation simulation acts exclusively on the tetrahedral mesh. To show the deformation for the user, changes in the triangular render mesh are required. However, the mapping between the two is not trivial, as not all vertices of the render mesh are present in the tetrahedral mesh and the tetrahedral mesh contains a lot more vertices than the render mesh. This first iteration of a mapping algorithm uses the surface mesh file generated by *TetWild*. This mesh is not identical with the render mesh, hence it can not be used as a final mapping algorithm. However, it can be rendered in Unity, which allows for a simple method of visually validating changes in the tetrahedral mesh during the deformation. It therefore was used in the beginning of the development, before a more advanced mapping algorithm was implemented and enabled debugging of the deformation algorithms without any interference of the mapping algorithm.
The mapping between the surface mesh generated by *TetWild* and the tetrahedral mesh is simple as this surface mesh contains all surface vertices and surface faces of the tetrahedral mesh. The surface vertices are therefore a subset of the tetrahedral vertices, which enables the use of a simple indexing algorithm (algorithm 5). A array is created that has the same size as the array that holds the surface vertices (line 2). For each vertex of the surface mesh, the algorithm finds the index of the vertex in the tetrahedral mesh that is identical to it and stores it in the map (lines 4 - 7). A maximal distance is set for two vertices to be considered equal (line 3). This accounts for floating number rounding errors and vertices that are very similar.

Algorithm 5 Subset indexing

```
 1: procedure IndexSubset
 2:     initialize map with surfaceMesh.vertexCount elements
 3:     initialize maxDistance
 4:     for all vertices s in surfaceMesh do
 5:         for all vertices t in tetrahedralMesh do
 6:             if distance(s, t) ≤ maxDistance then
 7:                 map[s] = t
 8:             end if
 9:         end for
10:     end for
11: end procedure
```

5.4.4 Barycentric Mapping

The direct mapping algorithm explained in section 5.4.3 was an algorithm for quick testing and visualization only. It does not enable the tetrahedral mesh to modify the original render mesh. To allow the deformation of the tetrahedral mesh to impact the render mesh, barycentric coordinates were used. The algorithm finds the index of the closest tetrahedron to each vertex (algorithm 6, line 3) and then computes the barycentric coordinates of the vertex in relation to its closest tetrahedron (line 4).

This closest tetrahedron is determined for each vertex by finding the tetrahedron with the minimal distance between the vertex and the tetrahedrons center.

Algorithm 6 Barycentric Mapping

```
1: procedure GenerateBarycentricMapping
2:     for all vertices v_i in renderMesh do
3:         determine closest tetrahedron t_i
4:         determine barycentric coordinate bc_i
5:     end for
6: end procedure
```

Due to the way *TetWild* generates the tetrahedral mesh, 50%-80% of surface vertices are equal to vertices in the tetrahedral mesh (see section 6.3). Utilizing this circumstance, a modified version of the barycentric mapping algorithm could be implemented. Before determining the closest tetrahedron for each vertex, the algorithm checks whether there exists a vertex in the tetrahedral mesh that is in the same position (algorithm 7, line 3). Algorithm 5 was used for this feature. When an identical vertex was found, it was stored and the barycentric mapping step (line 4-5) was skipped. If no equal tetrahedral vertex could be determined, the vertex is mapped barycentrically, as shown previously in algorithm 6. This accelerated the initialization speed by a factor of up to *2.5*. More detailed performance information are laid out in chapter 6.

Algorithm 7 Barycentric Mapping with identical map

```
1: procedure GenerateBarycentricMapping(renderMesh, tetrahedralMesh)
2:     for all vertices v_i in renderMesh do
3:         if not index of identical vertex v_t in tetrahedralMesh exists then
4:             determine closest tetrahedron t_i
5:             determine barycentric coordinate bc_i
6:         end if
7:     end for
8: end procedure
```

5.4.5 Serialization

For a given tetrahedral mesh and render mesh, the barycentric mapping and the generated constraints are constant. Each time, the program is started the initialization has to load and generate the data again. In early iterations, the initialization was not optimized for performance, which caused the start of the program to be rather slow. To solve this issue, the data was serialized after being generated, which would store the loaded and generated data into a file with the .tetmesh extension. The deformation module can be prompted to load the serialized data from the file rather than going through the costly generation process. Storing the data during the initialization increases the initialization time by 25%-30% for most meshes. This

however is well compensated for by the fact that loading the serialized data from a file reduces the initialization time by 70%-95%. The effect of the serialization on the performance is explored more thoroughly in chapter 6.

```
<3-character identifier> <count of data entries>
<entry 1>
...
<entry n>
```

Figure 5.4: The format of the serialized data in the .tetmesh file. The 3-character identifier uniquely identifies what data is to follow. The count specifies how many entries need to be reserved in the data structure, that the serialized data is to be stored in. The count field also allows for faster reading of the file, as it defines how many lines need to be read before looking for a new identifier.

```
bct 1244
1087
...
129
bcc 1244
0.969657 -0.00312827 0.0511999 -0.0177256
...
0.0676069 -0.00012562 1.01753 -0.0849863
```

Figure 5.5: Exemplarily serialized data. This snippet contains 1244 barycentric coordinates, identified by the identifier *bcc* and the 1244 respective related tetrahedron indices, identified by the *bct* identifier.

5.4.6 Code Architecture

In early iterations, the initialization step was done on the side of *Unity*. All data was loaded from files and generated in a C# script and was then passed to the deformation module in order to proceed with the simulation. This decision was made in order to verify the logical correctness of algorithms. As all data from within *Unity*s scripts is available at run time, debugging was very simple. Information could be simply accessed, printed to screen and analyzed in step-by-step debugging, which allowed for a rigorous testing of the algorithms of the initialization.
In later iterations, all initialization were ported from C# to C++ and moved to the deformation module. This decision was motivated by improved code design. Most of the data generated in the initialization step is not required by *Unity* and is exclusively used by the deformation module. Generating and storing data in a *Unity* script that is not required for the program is unnecessary. C# is generally slower than C++ and frequent passing of data between *Unity* and the deformation module additionally slows down the execution.
The deformation module was also designed to be a self contained module. This includes having a minimal interface to the program that is using it and generating all the required data on its own. This allows the deformation module to be freely changed and even used in different game engines, with minimal requirements on the game engine.

5.5 Deformation Loop

After all data necessary for the deformation simulation is generated in the initialization step, the deformation loop is started. *Unity* registers obstacles in the scene which collide with the car. The user can control the car using the keyboard. Controlled by *Unity*s internal game loop via the *Update* function, the deformation simulation gets invoked each frame. Collisions between the particles of the tetrahedral mesh and the obstacles are detected and resolved and the constraints are solved in order to compute the deformation for each frame.

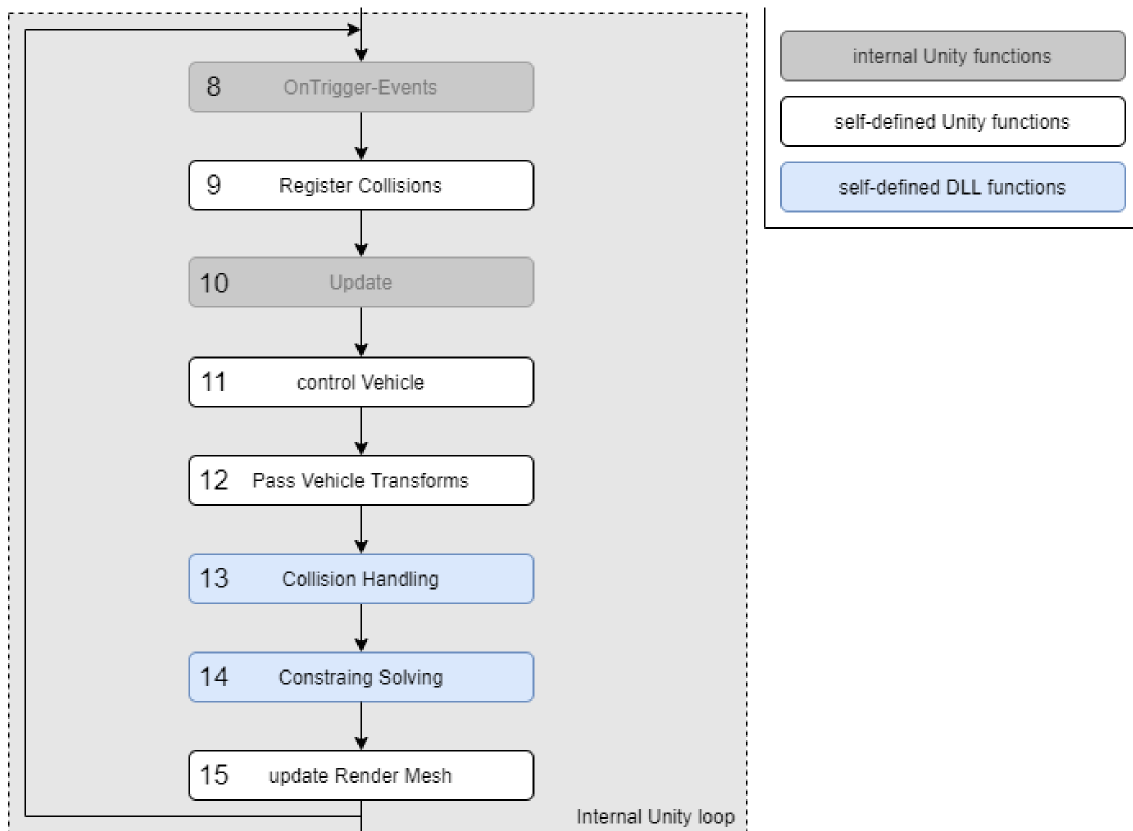

Figure 5.6: Excerpt of the flow chart focusing on the deformation loop. Functions that are executed each frame are shown.

5.5.1 Collision Handling

The collision handling step is the first of two steps in the deformation simulation. Collision handling involves detecting collisions between the vertices of the tetrahedral mesh and obstacles as well as responding to collisions. Collisions are typically handled by three-dimensional objects called colliders. In computer graphics, there is a large variety of collider types varying in complexity and applicability to different contexts. One of the simplest and most used collider types are axis aligned bounding boxes, commonly abbreviated as AABBs. AABBs are characterized by a position and the three dimensions of width, length and depth. Due to their simplicity, AABBs were implemented in this thesis as the only collider type. All colliders

are furthermore static. Static colliders are fixed in their position through the entire run time of the program. This allows *Unity* to pass the colliders positional data and size data to the deformation module in the initialization step, rather than continuously. This decision was made to focus on the deformation, rather than properties of the virtual environment. It was however made sure, to make the addition of new collider types as seamless as possible.
The collision detection is concerned with finding which vertices of the tetrahedral mesh collide with obstacles. The collision detection evolved throughout the course of this thesis to be increasingly sophisticated. The final state of the collision detection works on two layers, which are presented in the following sections.

5.5.1.1 Coarse Collision Detection

The first, coarse layer detects collisions between the entire car and the obstacles. This ensures that only relevant obstacles are checked for each particle, which greatly reduces the number of collision checks each frame. This step is handled by *Unity*s collision system. A box collider is attached to the car *GameObject* which serves as a convex hull to the car. Whenever this collider detects a collision with an obstacle in *Unity*s *OnTrigger* events, the index of that obstacle is registered by the *Obstacle Manager* object. The resulting list of relevant obstacle indices is then passed to the deformation module. Whenever the car and a certain obstacle no longer collide, that obstacles id is removed from the list.

5.5.1.2 Fine Collision Detection and Collision Response

The deformation module can now handle the second, fine layer of collision detection. The tetrahedral meshes space is local to the car it is attached to. This is done to limit the input, Unity has to give to the deformation module each frame. Instead of passing the world-space position of each vertex each frame, the vertices are local to the car, and the cars position and rotation is passed. This leads to the colliders and the tetrahedral mesh being in different spaces in the deformation module. To handle collisions, the vertices are transformed into global space (line 3). Then a collision check is performed. If the vertex collides with the collider, it is projected onto the surface of the collider and is transformed back into car-local space.

Algorithm 8 Collision Handling

```
 1: procedure CollisionHandling
 2:     for all tracked colliders c_i do
 3:         for all vertices v_j do
 4:             transform v_j into world space
 5:             if v_j collides with c_i then
 6:                 project v_j onto surface of c_i
 7:             end if
 8:             transform v_j into car-local space
 9:         end for
10:     end for
11: end procedure
```

5.5.2 Constraint Solving

The deformation module uses two types of constraints for the deformation computations: distance constraints and volume constraints. Distance constraints preserve the distance between two vertices, while volume constraints preserve the volume of a tetrahedron. Both types are generated on the basis of the tetrahedral mesh as explained in section 5.4.2.

5.5.3 Solver

The step of solving the constraints is the core of the simulation step as it is computationally the most expensive and handles the actual deformations. There are different strategies in solving the set of constraints. Both the Gauss-Seidel solver and the Jacobi-solver were implemented. Unfortunately, the computationally less expensive Jacobi solver proved as unstable. The solver used in this thesis therefore is the Gauss-Seidel solver. The implementation adds an iterative loop, that executes the solver multiple times to converge closer towards a perfect solution (see algorithm 9). Furthermore, different constraint types are solved consecutively, as their solving functions differ.

Algorithm 9 Iterative Gauss-Seidel Solver

```
1: loop iterationCount times
2:     solveDistanceConstraintsGaussSeidel()
3:     solveVolumeConstraintsGaussSeidel()
4: end loop
```

5.6 Parallelization

Enhancing performance, especially for the deformation loop had a big emphasis. The most impactful step was to transform serial implementations into parallel implementations. Most loops could be simply re-written into code that enabled the multi-core CPU to process tasks in parallel. This was accomplished via *Intel*s *Threading Building Blocks* (short: *TBB*) library. *TBB* offers a range of algorithms that emulate standard programming loops. Through *TBB* almost all loops could be parallelized which significantly increased the performance in both the initialization phase and the deformation loop. Details on the impact of parallelisation on the prototypes performance are explored in detail in chapter 6, section 6.4.
The algorithms that could not be parallelized were all hindered by one circumstance. They were dependent on the order of execution of their sub-tasks. When an algorithm is programmed parallely, there is little control over which sub-task gets computed first. The order of executed tasks is entirely determined by the scheduler of the underlying parallelization solution, i.e. *TBB*. This results in inconsistencies in the time each sub-task takes which in turn imposes on the algorithm that is can not depend on the order of execution. There are methods of synchronizing the sub-tasks but this did not suffice for this thesis. This issue existed in three algorithms.

The first is the deformation loop as a whole. The solver needs to be applied to the underlying, deformed mesh in multiple iterations. One iteration has to be fully executed before the subsequent one starts. A parallel implementation can therefore not accomplish this task, as the algorithm is logically serial. This also applied to the Gauss-Seidel solver. The algorithm is designed to compute each constraint solution individually in a way that later constraints depend on the results of previously processed ones. The third algorithm that was execution order dependent, was the constraint generation algorithm. Other then the prior algorithms, its serialism is not of logic nature. It is caused by the way constraint data is generated and stored into the data container. Consider a distance constraint. A constraint has data values for:

- An index to each of its vertices
- A resting value

This data can be stored in two ways. The first way is an array of structures (AoS). A structure in this context is a custom data type that composites of multiple variables. The opposite approach is a structure of arrays (SoA) which organizes each data type in a distinct array. One can retrieve the data of a specific constraint by getting the data from each array at the constraints index.

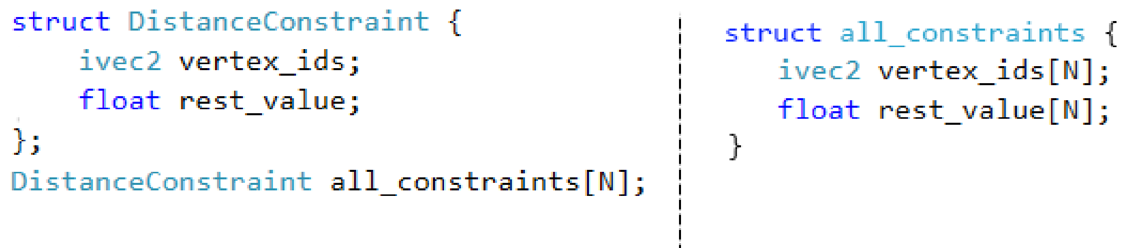

Figure 5.7: Comparison between an *Array of Structures* (left) and a *Structure of Arrays* (right) approach to storing data.

The prior approach is the more conventional as it is more intuitive for programmers. The latter is commonly used in real-time applications as the programmer has more direct access to the data, which enhances performance. It is however the origin of the order-dependency of the constraint generation algorithm. When thousands of constraint vertices and rest values are added to their respective data containers in parallel, it can not be guaranteed that both the vertex indices and the rest values are added to the container at the same time. This can bring the data entries indices out of order, which disturbs future manipulations on the constraints. As the algorithm is part of the initialization step, which has been greatly sped up through data serialization, there was no greater need to parallelize it.

5.7 Force Feedback

When a car collides with an obstacle and a deformation is computed, there are not only physical influences on the particles that make up the underlying meshes. The

car as a whole gets decelerated, rotated or diverted in its path. Modelling this behaviour with the current prototype is a challenge. In the prototype, there are two main systems working together. *Unity* handles the physics of the car as a whole, while the deformation module manipulates the particle positions of the cars underlying tetrahedral mesh. This setup has many advantages for a real-time deformation method, but it also comes with certain restrictions. Specifically, forces, accelerations and velocities are entirely unknown to the deformation module. It operates purely on particle positions. This restricts the way the deformation module can determine a collisions resulting force on the whole car. In this thesis' prototype, the force feedback on the car is handled by *Unity*s collider objects. The car object contains a set of colliders that can collide with obstacles in the environment and produce changes in the cars position and rotation. This computation happens as a part of *Unity*s internal game loop. This has the upside of being perfectly integrated in *Unity*s physics system but the downside of being entirely separate from the deformation module. Furthermore, the colliders are entirely rigid and impose this rigidity on areas in the tetrahedral mesh, which will hinder their deformation. In section 6.4 *Outlook and Future Work* of the following chapter, possibilities for a more accurate approach are investigated.

5.8 Summary

In chapter 5, the implementation of this thesis' prototype was outlined. A position-based deformation technique was the basis of the implementation. Details of the implementation were adapted to fit the context and requirement of this thesis. The separation of the deformation module and the game engine introduced some requirements on the design of the code architecture and the design of individual algorithms. The execution flow of the prototype can be separated into two main steps. The initialization step is responsible for generating all the data necessary for the deformation simulation. This includes generating all constraints and barycentrically mapping the tetrahedral mesh to the surface mesh. The deformation loop is the second step of the prototype which computes the deformation of the tetrahedral mesh and maps it to the surface mesh. The most notable algorithms in the deformation loop are the handling of the collision of particles with obstacles, the solving of the constraints in the tetrahedral mesh and the barycentric update, which maps the deformations in the tetrahedral mesh to deformations in the surface mesh. Additionally, to the execution flow and set up, methods to increase the performance were discussed. Specifically, the parallelization of the code, data serialization and a performance increase in the barycentric mapping algorithm were explained. The effect of these performance increases is laid out in detail in the upcoming chapter.

6
Results

In this chapter, the final prototype will be assessed regarding the research question and requirements stated in the first chapter of the thesis. The prototypes performance, stability, integratability and controllability as well as the visual quality are assessed based on data collected from the prototype. Subsequently, suggestions for future work are presented and related to currently existing features.

6.1 Critical Goal Assessment

In the first chapter of this thesis, the requirements for the development of the prototype were stated. The feasibility of the prototype with regards to each requirement will now be assessed. As performance and the quality of the visual results posed the most crucial requirements, an entire section is devoted to each the visual analysis and the performance analysis.
Scalability. The prototype allows the usage of an arbitrary surface mesh with an arbitrary tetrahedral mesh. One limitation comes with the use of *TetWild* to generate tetrahedral meshes from surface meshes. When a certain complexity of the meshes is reached *TetWild* will no longer be able to generate tetrahedral meshes. This however is no direct shortcoming of the prototype at hand. The deformation module is written in a way that existing features can be altered with relative ease and new ones can be added. This is a result of the modular design of the code. For instance, while only one type of collider was implemented for the prototype, the code was created with the addition of more collider types in mind. This also applies to other areas like the solver and different constraint types. The addition of more than one deformable vehicle is not currently supported, as the interface between *Unity* and the deformation module is not set up to allow so. This as a deliberate decision, to ease the prototype development, as more challenges (such collisions between vehicles) would arise.
Stability. The requirement for stability in chapter 1 stated that the deformation technique must be unconditionally stable. This could not be accomplished with absolute certainty. In certain collision scenarios with specific parameter set-ups the deformation module can become unstable. This is relatively rare however. The use of a Gauss-Seidel solver enables good stability for a majority of cases.
Integratability. The deformation module is well integrated into the *Unity* game loop. It utilizes functions that are part of *Unity*s standard flow control and is connected to *Unity* via a limited interface. The interface is used to send data back and forth, set up parameters and serialization paths and to query deformation com-

putations. The module is imported into *Unity* as a standard dynamic link library (DLL), which can be imported and used by most physics and game engines. The corresponding software thus needs to merely implement the interface and the module can be used. One shortcoming with regards to integratability is that the module works only based on obj-files. Other 3D formats are not supported.

Controllability. The deformation module offers different parameters that can be changed throughout the run-time of the prototype. The plasticity of the material as well as the stiffness of the two constraint types can be adjusted to alter the material behaviour of the car. The iterations of the solver can be changed in order to modulate the solver convergence and the computation time. However, many characteristics of metals are hard to approximate with the given parameters. A typical property of cars bodyworks is that a deformation in the material propagates throughout an entire bodywork component. This can not be adjusted in the current state of the prototype. Furthermore, the deformations are completely independent of the collision speed. This is in part due to the inner workings of the position-based approach to deformation. As the physical force layer is omitted, it is difficult to incorporate it into the system.

6.2 Visual Analysis

The visual quality of the deformations was developed with the performance constraints of the real-time environment in mind. The goal for the thesis therefore was to generate visually plausible deformations that could be generated within the given time constraints. Features that could potentially increase visual quality were implemented based on whether they were feasible with regards to their computational complexity. The visual results are analysed based on images taken from the prototype. Figure 6.1 shows the deformation of a vehicle as a result of an obstacle collision.

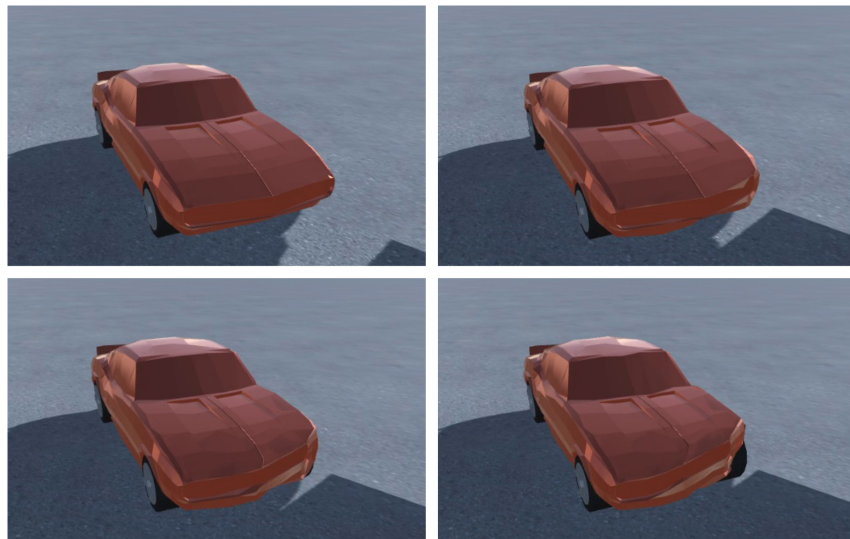

Figure 6.1: The process of a car being deformed by an obstacle collision. The obstacle has been made invisible for better visibility. The top left image shows the vehicle in its original state. The more it drives into the obstacle, the more it deforms. The deformation is especially visible on the cars front left corner.

Note that not only the vertices of the car that collide with the obstacle are displaced. The influence of the two constraints achieve a propagation effect of the deformation. This models the material characteristics of a part of a car bodywork. This deformation propagation can also result in visual artifacts. This is illustrated in figure 6.2.

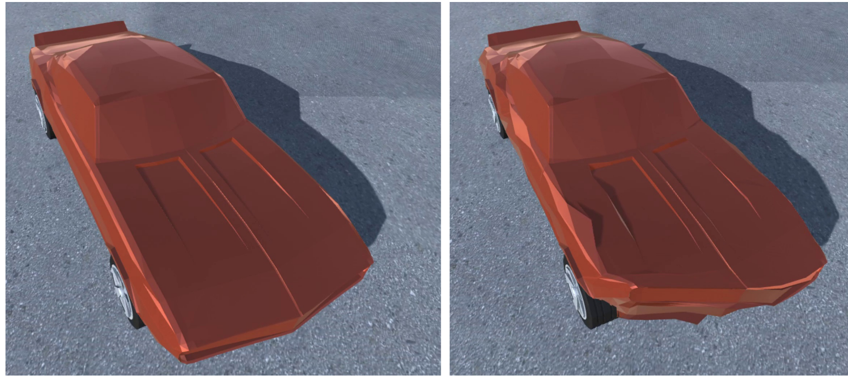

Figure 6.2: A car before and after a frontal collision. Artifacts are visible especially on the right side of the hood and the right side panel.

These artifacts are a result of the constraints trying to preserve the volume of the entire car. A decrease of the volume in the front area results in a propagation throughout the entire vehicle. This effect originates from the fact, that the entire vehicle is modeled by a single, continuous tetrahedral mesh. The bodyworks of real vehicles are made up of multiple individual panels, which typically limit the deformation propagation to more localized areas. A method of modeling this behaviour is proposed in section 6.4.5. With the current implementation of the prototype, such artifacts can not be avoided entirely. Tuning the parameters that control the strength of the constraints as well as the overall plasticity of deformations can however reduce their occurrence.

6.3 Performance Analysis

Performance was the most crucial aspect of the prototypes development. This section will therefore explain in detail which algorithms take most of the computation time of both the initialisation step and the deformation loop. Subsequently, it is evaluated how the measures that were taken to increase performance have actually contributed.

6.3.1 Methodology

To generate a performance assessment that is as general and broad as possible, different configurations for the surface and tetrahedral mesh have been evaluated. Two surface meshes were used to generate four tetrahedral meshes each. The underlying surface meshes are respectively comprised of 1259 and 3764 surface vertices. Each mesh was then tetrahedralized via *TetWild* into tetrahedral meshes comprised of ~1000, ~5000, ~10,000 and ~50,000 vertices. A tuple of a given surface mesh and a tetrahedral mesh generated from it will be referred to as a *mesh combination*. A total eight mesh combination were created as it was expected that different algorithms were differently dependent on the complexities of the tetrahedral mesh and the surface mesh. For instance, barycentric mapping is influenced by both meshes, while most of the deformation computations are dependent on only the tetrahedral

mesh. Therefore, having a range of variability in both mesh types is important. Figure 6.3 shows the two surface meshes used for the performance analysis and the surfaces of a selection of resulting tetrahedral meshes.

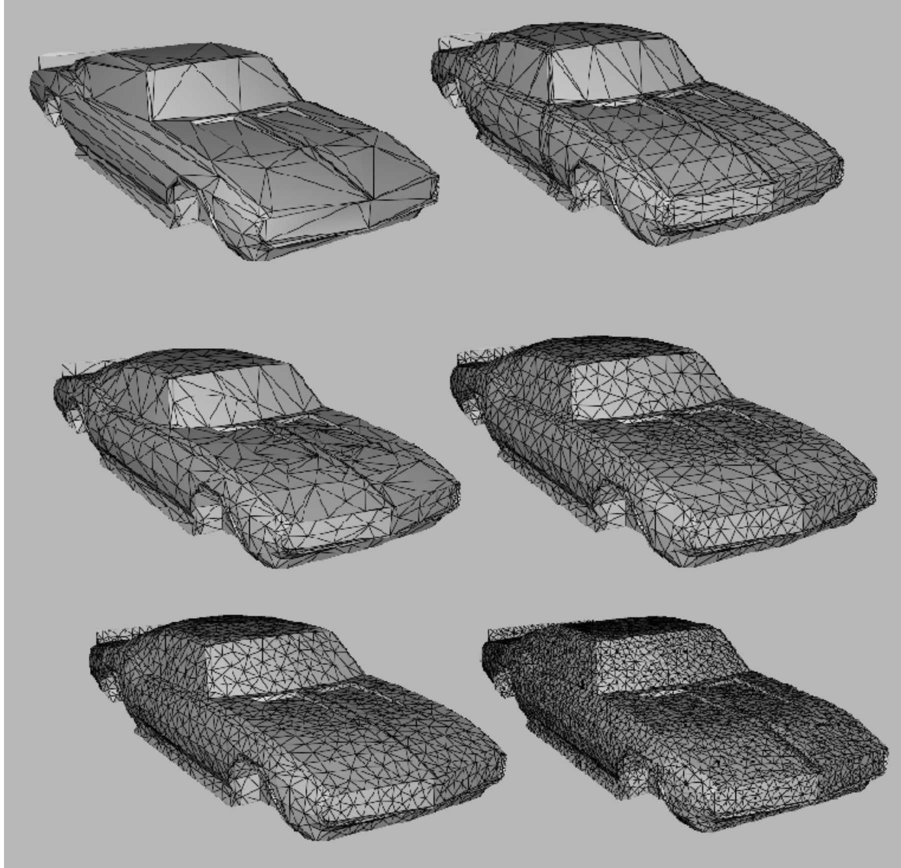

Figure 6.3: Top Row: The two surface meshes used for the generation of tetrahedral meshes, with 1259 (top left) and 3764 (top right) surface vertices. Center and Bottom Row: Tetrahedral meshes of 1000 (center left), 5000 (center right), 10,000 (bottom left) and 50,000 (bottom right) vertices.

When analyzing a specific optimization, each of the eight mesh combinations were used in a standardized virtual environment. The vehicles ran through a given scenario without the considered optimization, five times per mesh combination and the data was collected. The vehicles than ran through the scenario another five times per mesh combination, this time with the optimization enabled. The data could then be compared and evaluated. This procedure is executed on a computer with an Intel Core i5-8500 @3.00 GHz CPU and 8.00 GB RAM.

6.3.2 Initialization

In the initialization steps, performance increases were accomplished by three measures. First, all algorithms were parallelized, when possible. Second, a data serialization method was implemented to avoid the generation of previously generated data. Third, an algorithm was implemented that could reduce the computation time

of the barycentric mapping process. This measure was taken, as it became clear that the barycentric mapping of surface mesh to tetrahedral mesh took up a large amount of the processing time in the initialization. Each of the measures is now evaluated based on data collected from the performance tests. Throughout the analysis, a mesh combination of medium complexity is used to show which algorithms take up which portion of the total time. This mesh combination is comprised of a surface mesh with 3764 vertices and a tetrahedral mesh with 5000 vertices.

Table 6.1: The computation time (ms) of a serially implemented initialization step of a mesh combination of medium complexity.

file reading	**constraint generation**	**barycentric mapping**	**total time**
40	14	2571	2639

By far the most influential algorithm on initialization performance is the barycentric mapping algorithm (table 6.1). This can be attributed to the time complexity of $\mathcal{O}(n_s * n_t)$, where n_s is the number of surface mesh vertices and n_t is the number of tetrahedral mesh vertices. The complexity results from the algorithm going through all tetrahedral mesh vertices vertex to find the closest one to each surface vertex. While this complexity is inherent to the algorithm, it is clear that the barycentric algorithm is to be optimized first to increase performance of the initialization step. There are two measures that were taken.

6.3.2.1 Parallelization

The first measure is the parallelization of the initialization code. Table 6.2 shows the results of the comparison between a serial and a parallel implementation of the initialization step. The more the complexity of the underlying meshes increases, the more does the performance increase. For simple meshes the parallel implementation outperforms the serial one by a factor of *2.49*. For mesh combinations of higher complexity this factor rises to op to *5.20*.

Table 6.2: Comparison between the initialization times (ms) of a serial implementation (*se*) and a parallel implementation (*pa*).

		Surface Vertices					
		1259			**3674**		
		se	*pa*	*res*	*se*	*pa*	*res*
Tet Verts	**1000**	164	65	2.49	411	110	3.73
	5000	964	229	4.21	2600	533	4.95
	10000	1800	418	4.37	5200	1000	5.15
	50000	9700	2100	4.56	2910	5600	5.20

Most of this performance increase results from a reduction in the barycentric mapping algorithm. However it still consumes most of the computation time.

Table 6.3: The computation time (ms) of a parallelly implemented initialization step of a mesh combination of medium complexity.

file reading	constraint generation	barycentric mapping	total time
38	14	467	533

6.3.2.2 Barycentric Mapping

The internal functionality of *TetWild* results in a tetrahedral mesh, that has many vertices in the exact same position as vertices of the underlying surface mesh. Table 6.4 shows that 50%-82% of surface vertices can be mapped directly to a vertex in the tetrahedral mesh. These surface vertices do not require the computationally expensive search for the closest tetrahedron and can be simply mapped to a singular tetrahedral vertex. This approach is explained in detail in section 5.4.4.

Table 6.4: The quota of surface vertices that can be mapped directly to vertices of the tetrahedral mesh.

		Surface Vertices	
		1259	**3767**
Tet Verts	**1000**	0.82	0.50
	5000	0.82	0.53
	10000	0.82	0.53
	50000	0.82	0.51

The following table gives an overview of how much this increases performance compared to the parallel implementation without a direct mapping.

Table 6.5: Comparison of the initialization times (ms) of a parallel implementation (*pa*) and a parallel implementation with a direct mapping (*pd*) to improve the computation speed of the barycentric mapping step.

		Surface Vertices					
		1259			**3674**		
		pa	*pd*	*res*	*pa*	*pd*	*res*
Tet Verts	**1000**	65	56	1.17	110	88	1.25
	5000	229	111	2.01	533	310	1.72
	10000	418	180	2.32	1000	594	1.71
	50000	2100	851	2.5	5600	3300	1.68

The contribution of the barycentric mapping algorithm to the total computation time is still prominent but has been much decreased.

Table 6.6: The computation time of a parallelly implemented initialization step with a direct mapping algorithm for a mesh combination of medium complexity (combination 3674/5000).

file reading	constraint generation	barycentric mapping	total time
38	14	248	310

6.3.2.3 Data Serialization

With a given mesh combination, the generated data for constraints and barycentric mapping are constant. This circumstance has been exploited with a data serialization implementation that lets the user save and load currently generated data. The data that resulted from this last improvement is shown in table 6.7.

Table 6.7: Comparison of the initialization times (ms) of a parallel implementation with a direct barycentric mapping (*pd*) and an initialization that is replaced by data deserialization (*dd*).

		Surface Vertices					
		1259			**3674**		
		pd	*dd*	*res*	*pd*	*dd*	*res*
Tet Verts	**1000**	56	12	4.68	88	18	4.79
	5000	111	66	1.69	310	75	4.13
	10000	180	136	1.31	594	148	4.00
	50000	851	774	1.10	3300	782	4.25

As this optimization decreased the loading time to under 1 second, the result was deemed sufficient for this thesis' purpose.

6.3.3 Deformation Loop

Similarly to the analysis of the initialization step, the deformation loop is analyzed in a standardized environment. The same collisions are executed multiple times and the performance data is collected for each mesh combination. The resulting data is averaged per frame. For computer applications, 30 frames per seconds (fps) is often used as the standard for an interactive speed. 60 fps is regarded as real-time speed. This time frame is comprised of all functionality that is executed each frame of the game loop, including rendering, physics calculations and artificial intelligence. Therefore, the deformation simulation should take only a fraction of the above time frames.
The deformation loop can be separated into three steps. First, the vertices are displaced by collisions with obstacles (section 5.5.1). Second, the solver iteratively solves the constraints according to the prior deformation and the provided parameters (section 5.5.3). Finally, the surface mesh is updated via the barycentric update (section 5.4.4). The first iteration of this process was implemented serially, without the use of any parallel hardware capabilities. The CPU used, is an Intel i5-8500 @ 3.00 GHz. Table 6.8 shows the collected data for the eight mesh combinations. It

is important to note that the times noted for the constraint solving step are taken for a single iteration, whereas typically Gauss-Seidel solvers are run multiple times. It becomes clear that the focus lies on the collision projection and the constraint solving step, as they take up most of the frames time.

Table 6.8: Averaged execution times (ms) of a serial implementation of the deformation loop for different mesh combinations.

		Surface Vertices							
		1259				**3674**			
		cp	*cs*	*bc*	*total*	*cp*	*cs*	*bc*	*total*
Tet Verts	**1000**	0.18	0.18	0.02	0.38	0.19	0.19	0.05	0.43
	5000	0.94	1.21	0.03	2.17	0.85	1.15	0.07	2.08
	10000	1.80	2.32	0.03	4.16	1.75	2.42	0.08	4.25
	50000	10.73	15.52	0.04	25.29	13.44	14.31	0.09	27.85

where
cp .. collision projection (ms)
cs .. constraint solving (ms)
bc .. barycentric update (ms)
total .. total deformation loop (ms)

To optimize the serial implementation, the collision projection and barycentric update algorithms were parallelized. As a Gauss-Seidel approach was used for the constraint solver, this step is inherently serial. Still, the performance could be increased significantly, as shown in table 6.9.

Table 6.9: Averaged execution times (ms) of a parallel implementation of the deformation loop for different mesh combinations.

		Surface Vertices							
		1259				**3674**			
		cp	*cs*	*bc*	*total*	*cp*	*cs*	*bc*	*total*
Tet Verts	**1000**	0.09	0.17	0.03	0.29	0.10	0.20	0.06	0.36
	5000	0.23	1.15	0.04	1.42	0.23	1.25	0.07	1.55
	10000	0.38	2.36	0.05	2.79	0.39	2.50	0.07	2.96
	50000	1.51	14.31	0.05	15.87	1.53	14.25	0.08	15.86

where
cp .. collision projection (ms)
cs .. constraint solving (ms)
bc .. barycentric update (ms)
total .. total deformation loop (ms)

When compared to the serial implementation (see table 6.10), a number of things can be noticed. The performance of the collision projection algorithm can be boosted

greatly from a factor of *1.93* up to a factor of *8.19*. The difference in the constraint solving step is fluctuating inconsistently. This can be attributed to external factors, as the algorithm of the solver has not been adjusted. Interestingly, the barycentric update function loses performance on parallelization for smaller surface vertices. As the barycentric update encompasses merely ~1000 tasks, the overhead for setting up a parallel loop outweighs the faster computation speed. From the data it can also be deduced that the biggest factor still is the constraint solution, especially with a solver that runs through multiple iterations. A big change in the performance of the collision projection algorithm has only little impact on the performance of the entire deformation loop. In the following chapter it is explored how the constraint solving step can be accelerated.

Table 6.10: A comparison between the serial and the parallel implementation of the deformation loop. The factors with which the parallel implementation outperforms the serial one are displayed. For instance, the parallel implementation of a collision projection step in the mesh combination 1259/1000 outperforms the serial version by *2.09*

		Surface Vertices							
		1259				**3674**			
		cp	*cs*	*bc*	*total*	*cp*	*cs*	*bc*	*total*
Tet Verts	**1000**	2.09	1.03	0.60	1.29	1.93	0.96	0.81	1.20
	5000	4.04	1.05	0.69	1.53	3.73	0.92	1.08	1.34
	10000	4.73	0.98	0.75	1.49	4.52	0.97	1.20	1.44
	50000	7.10	1.01	0.78	1.59	8.79	1.00	1.12	1.76

where
cp .. collision projection (ms)
cs .. constraint solving (ms)
bc .. barycentric update (ms)
total .. total deformation loop time (ms)

6.4 Outlook and Future Work

In this final chapter, possible extensions to the deformation system are presented. Existing research that could possibly assists such extensions is outlined. Furthermore, existing features that could be re-used or expanded on are outlined. For complex extensions, requirements for a desired algorithm are laid out.

6.4.1 Utilizing the GPU

The deformation simulation in this thesis' prototype is entirely done on the CPU. However, almost all calculations of computer graphics can achieve better performance when designed for and run on a GPU. GPUs are specifically designed to

process computations that are common for computer graphics. Especially the constraint solving step can undoubtedly benefit greatly from being implemented on the GPU.

6.4.2 A Parallel Gauss-Seidel Solver

The core of the deformation simulation is the constraint solver. Optimizations are therefore most critical in this algorithm. In the final prototype a Gauss-Seidel method was implemented, which proved as unconditionally stable but computationally slow. The relatively slow execution time is caused by the Gauss-Seidel algorithm being inherently serial. A parallelization is therefore impossible without complex synchronization techniques which is unsuitable for a real-time context. The implementation of a Jacobi solver could eradicate that issue as it is inherently can be parallel. The Jacobi implementation proved as unstable however. As this characteristic of the Jacobi algorithm is well known in the field of computer graphics, there is current research done on the issue. Fratarcangeli et. al provide a semi-parallel solver algorithm that combines the stability of a Gauss-Seidel method with the fast computation speed of Jacobi [29].
This method seems very applicable to this thesis' context. In the initialization, the set of constraints is partitioned into subsets, via a randomized graph coloring algorithm. The subsets are generated in a way that all constraints within a given subset are independent of one another. This allows all constraints within a given subset to be solved in parallel while the subsets themselves are solved iteratively. This method is capable of solving hundred-thousands of constraints in a few milliseconds using the Gauss-Seidel approach at its core.

6.4.3 Simulated Force Feedback from Obstacle Collisions

The current method of force feedback on the car is highly performant as it makes use of *Unity*s physics and collision system. However, it lacks in visual detail and configurability. It works entirely independent of the deformation module and therefore can not be modified. For future work, it is therefore worthwhile to investigate a method of generating a force feedback on the car as a function of how many particles are hit and how the respective parameters are set up. Implementing such a method requires extracting the velocity data from *Unity* which was previously not needed for any computations. The *Rigidbody* component is used to contain and manipulate all physical properties of an object in *Unity*. The physical data can simply be extracted from the cars *Rigidbody* component and sent to the deformation module each frame. The more challenging task is the inclusion of external physical data into the position-based approach.

6.4.4 Saving and Loading Deformation States

In the early state of the prototype development the generation of constraints and other initialization data executed rather slowly, which resulted in slow loading times of the program. To speed up the initialization, a feature was added that could store

all data generated to a file (see section 5.4.5). Now, the previously generated data could be loaded from a file, which reduced the initialization time to a few seconds. For a future extension, this data serialization feature could be used. The feature would enable the user to store and load the current deformation state of the car. To implement this, the file format that is currently used has to be extended to contain the current states. To enhance the saving and loading speed of the files as well as the space efficiency of the stored data, the data should be stored in binary form as opposed to the current text form.

6.4.5 Weight-Painting for Deformation Parameters

Vehicle exteriors do not deform uniformly across the entire surface. Some exterior parts are softer and more easily deformable than others. Typically, cars have load paths that distribute and dissipate the energy of a crash (see figure 6.4). Modelling this behaviour could significantly improve the realism of a deformation. To account for different vehicle types and to allow creative choices, a general solution as to be added.

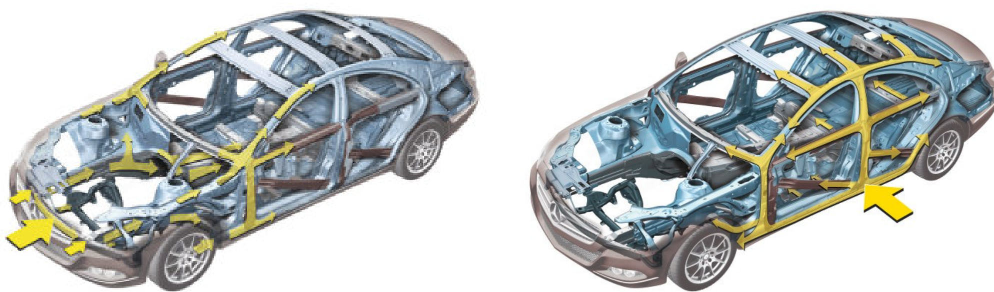

Figure 6.4: Load distribution for frontal impacts (left) and side impacts (right) on a Mercedes-Benz CLS-Class. [37]

One option for adding such functionality is to create a tool that allows artists to freely assign different material parameters to different areas of a vehicle model (see figure 6.5). This approach is inspired by *Blender*s weight painting tool [39]. Weight painting is a method frequently used in the animation of 3D objects.

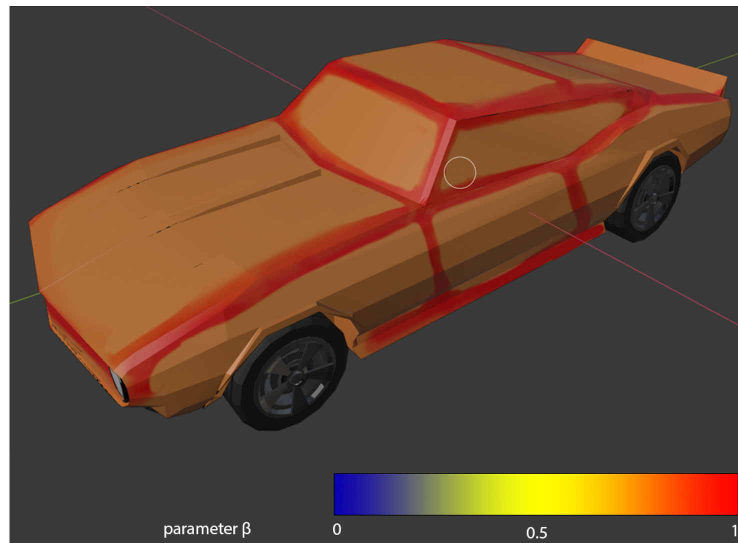

Figure 6.5: Mock-Up of a tool for painting parameter values onto the mesh of a vehicle. The red areas are assigned greater values. This can be used to assign areas of a mesh different physical properties.

7
Conclusion

The aim of this thesis was to investigate how one can implement a deformation system for a real-time application, such as a video game. The first part of the thesis was oriented around finding out the appropriate technology, software development methodology and simulation algorithms. The approach chosen was to develop a deformation library that could be used as a plug-in to a game engine. For this project, the game engine *Unity3D* was chosen as it can provide features that let me develop the prototype with a relatively low start-up time. The deformation library was developed to use a position-based approach to simulate deformations. The approach was chosen based on its low computational cost and based on the fact that it fit together quite well with the way game engines represent 3D objects and physics. The position-based approach was adjusted to fit the specific challenges of this thesis' set-up. Specifically, the particle movement was not computed as a part of the deformation simulation, but was inferred from *Unity*s internal physics simulation.
The second part of the thesis was focused around the details and challenges of the implementation. Algorithms were discussed in detail and opportunities for optimization were explained. Finally, the performance of the algorithms was tested in detail and the effect of optimization steps was evaluated. The performance of both steps that the deformation library executes could be increased. The deformation-loop could be sped up by a factor of *1.20* to *1.7*, while the initialization step could be sped up by a factor of up to *8*.
Finally, possibilities for future work are outlined. The two most important opportunities are the utilization of the GPU and the inclusion of force-feedback in the deformation simulation. Using the GPU instead of the CPU for the deformation simulation has the potential to significantly increase the performance of the deformation loop In conjunction with the use of the GPU, the solver algorithm has to be adjusted to allow for an efficient parallelization. A real-time force feedback is necessary to seamlessly use the deformation library as a plug-in.

Bibliography

[1] Demetri Terzopoulos, John Platt, and Kurt Fleischert. “Elastically Deformable Models”. In: *ACM SIGGRAPH Computer Graphics* 21.4 (1987), pp. 205–214. DOI: 10.1145/37401.37427.

[2] Demetri Terzopoulos and Kurt Fleischer. “Deformable models”. In: *Computer* (1988), pp. 306–331.

[3] Demetri Terzopoulos and Kurt Fleischer. “Modeling inelastic deformation : Viscoelasticity, Plasticity, Fracture”. In: *ACM SIGGRAPH Computer Graphics* 22.4 (1988), pp. 269–278. DOI: 10.1145/378456.378522.

[4] Psygnosis. *Destruction Derby*. 1995.

[5] David Baraff and Andrew Witkin. “Large steps in cloth simulation”. In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques - SIGGRAPH ’98*. New York, New York, USA: ACM Press, 1998, pp. 43–54. ISBN: 0897919998. DOI: 10.1145/280814.280821.

[6] Blender Foundation. *Blender*. Amsterdam, 1998.

[7] Mathieu Desbrun, Peter Schröder, and Alan Barr. “Interactive animation of structured deformable objects”. In: *Proceedings of the 1999 conference on Graphics interface ’99*. Kingston, Ontario, Canada: Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1999, pp. 1–8.

[8] James F. O’Brien and Jessica K Hodgins. “Graphical Modeling and Animation of Brittle Fracture”. In: *Proceedings of ACM SIGGRAPH 1999* 21.3 (1999), pp. 137–146.

[9] Gilles Debunne et al. “Dynamic real-time deformations using space & time adaptive sampling”. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH ’01*. New York, New York, USA: ACM Press, 2001, pp. 31–36. DOI: 10.1145/383259.383262. URL: http://portal.acm.org/citation.cfm?doid=383259.383262.

[10] Codemasters Interactive. *TOCA Race Driver*. 2002.

[11] Matthias Müller et al. “Stable real-time deformations”. In: *SCA ’02 Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*. San Antonio, Texas, 2002, pp. 49–54. DOI: 10.1145/545261.545269.

[12] James F. O’Brien, Adam W. Bargteil, and Jessica K. Hodgins. “Graphical modeling and animation of ductile fracture”. In: *ACM Transactions on Graphics* 21.3 (2002), pp. 291–294. ISSN: 07300301. DOI: 10.1145/566654.566579.

[13] Activision Interactive. *Street Legal Racing: Redline*. 2003.

[14] Thomas Jakobsen. “Advanced Character Physics”. In: *Game Developer Conference* (2003). DOI: 10.1063/1.1595059.

[15] J Teran et al. “Finite Volume Methods for the Simulation of Skeletal Muscle”. In: *SCA ’03 Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. San Diego, California: Eurographics Association Aire-la-Ville, Switzerland, Switzerland ©2003, 2003, pp. 68–74. ISBN: 1-58113-659-5. URL: https://dl.acm.org/citation.cfm?id=846285.

[16] Empire Interactive. *FlatOut*. 2004.

[17] Markus Gross and Matthias Müller. “Interactive virtual materials”. In: *GI ’04: Proceedings of Graphics Interface 2004*. London, Ontario, Canada, 2004, pp. 239–246. ISBN: 1-56881-227-2. URL: http://portal.acm.org/citation.cfm?id=1006087.

[18] Rigs Of Rods Contributors. *Rigs Of Rods*. 2005. URL: https://www.rigsofrods.org/.

[19] Unity Technologies. *Unity*. 2005. URL: https://unity.com/.

[20] Matthias Müller et al. “Position based dynamics”. In: *3rd Workshop in Virtual Reality Interactions and Physical Simulations, VRIPHYS 2006* (2006), pp. 71–80. DOI: 10.1007/978-3-319-08234-9{_}92-1.

[21] Matthias Müller et al. “Position based dynamics”. In: *Journal of Visual Communication and Image Representation* 18.2 (2007), pp. 109–118. ISSN: 10473203. DOI: 10.1016/j.jvcir.2007.01.005.

[22] Tom Preston-Werner, Chris Wanstrath, and P.J. Hyett. *GitHub*. San Francisco, CA, 2008.

[23] Eric G. Parker and James F. O’Brien. “Real-time deformation and fracture in a game environment”. In: *SCA ’09 Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* January 2009 (2009), pp. 165–175. DOI: 10.1145/1599470.1599492.

[24] *Trello*. London, 2011. URL: https://trello.com/.

[25] Jan Bender et al. “Position-based simulation of continuous materials”. In: *Computers and Graphics* 44.1 (Nov. 2014), pp. 1–10. ISSN: 00978493. DOI: 10.1016/j.cag.2014.07.004.

[26] BeamNG GmbH. *BeamNG.drive*. 2015.

[27] Russel Smith. *Open Dynamics Engine*. 2015. URL: http://www.ode.org/.

[28] Huda Basloom. “A Survey On Physical Methods For Deformation Modeling”. In: *Computer graphics forum* 5.10 (2016), pp. 59–64.

[29] Marco Fratarcangeli, Valentina Tibaldo, and Fabio Pellacini. “Vivace: A practical gauss-seidel method for stable soft body dynamics”. In: *ACM Transactions on Graphics* 35.6 (2016), pp. 1–9. ISSN: 15577368. DOI: 10.1145/2980179.2982437.

[30] Andrew Hickey and Shaoping Xiao. “Finite Element Modeling and Simulation of Car Crash”. In: *International Journal of Modern Studies in Mechanical Engineering* 3.1 (2017), pp. 1–5. DOI: 10.20431/2454-9711.0301001.

[31] Ageia and NVIDIA. *PhysX*. 2018. URL: https://www.geforce.com/hardware/technology/physx.

[32] Erwin Coumans. *Bullet*. 2018. URL: https://pybullet.org/wordpress/.

[33] Marco Fratarcangeli, Huamin Wang, and Yin Yang. *Parallel iterative solvers for real-time elastic deformations*. ACM, 2018. DOI: 10.1145/3277644.3277779.

[34] Yixin Hu et al. “Tetrahedral Meshing in the Wild”. In: *ACM Trans. Graph.* 37.4 (2018), 60:1–60:14. ISSN: 0730-0301. DOI: 10.1145/3197517.3201353.

[35] Yixin Hu et al. *TetWild*. New York, NY, 2018. URL: https://github.com/Yixin-Hu/TetWild.

[36] THQ Nordic. *Wreckfest*. 2018.

[37] Daimler AG. *The bodywork: A first: frameless, all-aluminium doors*. 2019. URL: https://media.daimler.com/marsMediaSite/ko/en/9361919.

[38] *OpenGL Mathematics*. 2019. URL: https://glm.g-truc.net/0.9.9/index.html.

[39] Blender Foundation. *Weight Paint - Introduction*. 2020. URL: https://docs.blender.org/manual/en/latest/sculpt_paint/weight_paint/introduction.html#the-weighting-color-code.

[40] Unity Technologies. *Standard Assets (for Unity 2017.3)*. URL: https://assetstore.unity.com/packages/essentials/asset-packs/standard-assets-for-unity-2017-3-32351.

[41] Unity Technologies. *Unity Manual*. URL: https://docs.unity3d.com/Manual/index.html.

[42] Unity Technologies. *Unity Manual: Order of Execution for Event Functions*. URL: https://docs.unity3d.com/Manual/ExecutionOrder.html.