INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

isep

Parametrização por vértice de física de corpo moles usando texturas

RICARDO MANUEL CRUZ ROQUE

Outubro de 2017

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Per-vertex soft body dynamics parameterization using textures

Ricardo Manuel Cruz Roque

Dissertation for obtainment of the Master Degree in Informatics Engineering, Specialization Track in Graphic Systems and Multimedia

Advisor: Filipe de Faria Pacheco

Porto, October 2017

Resumo

A simulação de física de corpos moles é usada em software de rasterização e em jogos de vídeo de forma a simular corpos deformáveis. As implementações atuais desta tecnologia só permitem um conjunto de variáveis de física por corpo ou, no caso de uma malha esqueletal, um conjunto por osso, o que não permite ao artistas controlar totalmente como a simulação vai afetar o corpo. À medida que o poder computacional do *hardware* melhora, o uso desta tecnologia torna-se mais comum. Devido a isto, devem-se desenvolver novas abordagens que dão aos artistas que criam corpos deformáveis um maior controlo sobre a simulação e que tornam o seu uso mais simples.

O objetivo desta dissertação é desenhar e desenvolver um sistema simples de física de corpos moles que permita aos artistas usarem texturas para atribuir a cada vértice do corpo o seu próprio conjunto de variáveis de física. Esta abordagem deve dar aos artistas maior controlo sobre a forma de como o corpo é deformado, exigindo apenas que eles usem *software* de edição de imagem, algo que a maioria dos artistas de 3D já conhece.

O sistema de física de corpos moles desenvolvido foi dividido em dois módulos independentes, um módulo de simulação que obtém as variáveis físicas de um corpo de uma fonte externa e um módulo de dados que lê as variáveis físicas de um corpo através de uma textura. Esta divisão facilita a implementação em, e a portabilidade para, software existente, como software de rasterização e motores de jogo.

Devido a apenas ter sido possível encontrar um pequeno grupo de testes composto por utilizadores inexperientes, os resultados desta dissertação não são conclusivos e não foi respondida a pergunta de o sistema desenvolvido dar maior controlo aos seus utilizadores enquanto é mais fácil de utilizar em comparação com as abordagens existentes.

Palavras-chave: Física de corpos moles, Deformações de malhas, Texturas

Abstract

Soft body dynamics simulation is used in rendering software and video games to simulate deformable bodies. Current implementations of this technology only allow one set of physics variables per body or, in the case of a skinned mesh, one set per bone, not giving the artists full control of how the simulation affects the body. As hardware's computational power improves, the use of this technology is becoming more common. Due to this, new approaches should be developed to give artists more control over the simulation while making its use simpler.

The objective of this dissertation is to design and develop a simple soft body dynamics system that allows artists to use textures to assign each vertex of the body mesh its own set of physics variables. This approach should give artists more control over the way the body is deformed while only requiring them to use image editing software.

The developed soft body dynamics system was split into two self-contained modules, a simulation module that gets a body's physics variables from an external source, and a data module that reads a body's physics variables from a texture. This split facilitates implementation in, and porting to, existing software such as rendering software and game engines.

Due to only being able to find a small test group composed of inexperienced users, this dissertation's results are not conclusive and the question of whether the developed system gives more control to its users while being easier to use in comparison to existing approaches, or not, has not been answered.

Keywords: Soft body physics, Mesh deformations, Textures

Contents

1 Introduction 1
1.1 Problem 1
1.2 Objectives 1
1.3 Approach 1
1.4 Expected results 2
1.5 Dissertation outline 2

2 Context and state of the art 5
2.1 Context 5
2.2 Soft body dynamics simulation 5
2.2.1 What is soft body dynamics simulation? 5
2.2.2 Uses for soft body dynamics simulation 6
2.3 Polygon meshes 6
2.3.1 What is a polygon mesh? 6
2.3.2 Uses for polygon meshes 6
2.4 Texture mapping 6
2.4.1 What are textures? 6
2.4.2 What is texture mapping? 7
2.4.3 Uses for texture mapping 7
2.5 Tessellation 7
2.5.1 What is tessellation? 7
2.5.2 Uses for tessellation 7
2.6 Physics 8
2.6.1 Hooke's law 8
2.6.2 Newton's second law of motion 8
2.6.3 Damping 8
2.7 Soft body dynamics models 8
2.7.1 Finite element simulation 8
2.7.2 Mass spring model 9
2.7.3 Shape matching 9

3 Existing soft body dynamics parametrization approaches 11
3.1 Blender 11
3.2 Carbon 12
3.3 BeamNG.drive 13
3.4 PhysX Clothing 15

3.5 Source ... 17

4 Design ... 19

4.1 Soft body dynamics model for the simulation module ... 19

4.2 Parametrization ... 20
4.2.1 Physics variables ... 20
4.2.2 Variable maximums ... 21

4.3 System modules ... 21
4.3.1 Simulation module ... 21
4.3.2 Data module ... 22

4.4 Application architecture ... 23

5 Implementation ... 25

5.1 Simulation module ... 25
5.1.1 Object data structure ... 25
5.1.2 Adding an object to the simulation ... 26
5.1.3 Modifying an object's physics variables during runtime ... 27
5.1.4 Removing an object from the simulation ... 28
5.1.5 Processing the simulation ... 29

5.2 Data module ... 31
5.2.1 PhysicsObject script ... 32
5.2.2 PhysicsSimulation script ... 33

6 Evaluation ... 37

6.1 Testing procedure ... 37
6.1.1 Test project ... 37
6.1.2 Survey questions ... 38

6.2 Result analysis ... 39

7 Conclusion ... 43

7.1 Limitations and future work ... 44

List of figures

Figure 1 – Soft Body parameters in Blender 2.75 12
Figure 2 – Non-uniform mesh behavior in Carbon Tetrahedron (Numerion Software, 2017) .. 13
Figure 3 – A vehicle’s nodes in JBeam (BEAM NG INC, 2017) 14
Figure 4 – A vehicle’s beams in JBeam (BEAM NG INC, 2017) 15
Figure 5 – Object variables in PhysX 3.4.0 Clothing plug-in for 3D Studio Max 16
Figure 6 – Vertex variables in PhysX 3.4.0 Clothing plug-in for 3D Studio Max 16
Figure 7 – File types for mesh information in PhysX 3.4.0 Clothing plug-in for 3D Studio Max 2015 17
Figure 8 – Design of the simulation module 22
Figure 9 – Design of the data module 22
Figure 10 – Architecture of the soft body dynamics system 23
Figure 11 – The object script as it appears in the Unity editor 32
Figure 12 – Unity test project in motion 38
Figure 13 – Survey results for the questions “For how long have you been using soft body dynamics systems?” and “How many different soft body dynamics systems have you used?” 40
Figure 14 – Survey’s ease of use questions 41
Figure 15 – Survey’s answer about the amount of control over the simulation the system provides 42
Figure 16 – Survey's answers to most wanted improvement 42

Glossary, Acronyms and Symbols

Glossary

3D object	A set of vertices, edges and faces
Edge	A connection between two vertices
Face	A set of edges that make a polygon
Material	A set of variables for a shader to use
Mesh group	A list of 3D objects
Shader	A program used to draw an image
Smoothing group	A set of faces whose shared edges will be smoothed during rendering
UV coordinates	Coordinates assigned to a vertex and used to define its position in a 2D image
Vertex	Set of data that represents a position in 3D space
Vertex normal	A directional vector that represents a face’s normal on a vertex’s position

List of Acronyms

2D	Two-dimensional
3D	Three-dimensional
CGI	Computer-generated imagery
QC	Quake C
SMD	Studiomdl Data

List of Symbols

$\boldsymbol{d}$	derivative

1 Introduction

This section is dedicated to the presentation of the dissertation and its theme, per-vertex soft body dynamics parameterization using textures, as well as its objectives and motivations. To do this, it starts by presenting the problem that inspired this project, as well as its objectives, approach and expected results, ending with a short description of this dissertation's structure.

1.1 Problem

Soft body dynamics simulation is used in rendering software and video games in order to simulate deformable bodies. Current implementations of this technology only allow one set of physics variables per body or, in the case of a skinned mesh, one set per bone, not giving the artists full control of how the simulation affects the body. As hardware's computational power improves, the use of this technology is becoming more common. Due to this, new approaches should be developed in order to give artists creating deformable bodies more control over the simulation while making its use simpler.

1.2 Objectives

The objective of this dissertation is to design and develop a simple soft body dynamics system that allows artists to use textures in order to assign each vertex of the body mesh its own set of physics variables. This approach should give artists more control over the way the body is deformed while only requiring them to use image editing software, something most 3D artists are already familiar with.

1.3 Approach

The developed system should have two modules:

- A simulation module that gets a body's physics variables from an external source;
- A data module that reads a body's physics variables from a texture.

These modules should be self-contained in order to facilitate implementation in, and porting to, existing software such as rendering software and game engines.

Before developing this system, a bibliographical research about the state of the art of soft body dynamics simulation's algorithms, parameterization, and optimization, will be done, followed by an analysis of how these techniques give control to their users. The system will be iteratively developed and, in order to test it, integrated into modern game engines, such as Unity 4 and Unreal Engine 5.

Each development iteration will use artist feedback to improve the system being developed. This feedback will be collected by asking artists who are experienced with current systems to try the developed system and then fill out a survey, followed by an analysis of their answers.

In order to confirm if this dissertation succeeds in its objective, a comparison between the developed system and current systems will have to be done. This comparison will use the feedback collected from a last testing and survey cycle.

1.4 Expected results

In this dissertation it's expected to develop and test a new, texture based, approach to soft body dynamics parameterization.

This approach should be easier to use and give the artist more control over the simulation than currently used approaches.

1.5 Dissertation outline

This dissertation is organized in four main sections, the Introduction, the Context and state of the art, the Design, and the Evaluation.

The Introduction has the objective of presenting this project and its objectives.

The Context and state of the art presents and describes the context in which this project is framed, as well as current methodologies and technologies.

The Design presents the design of the textures and modules.

The Implementation details how the design was implemented and explains any decisions made during its implementation.

The Evaluation details how the developed solution will be evaluated.

The Conclusion presents the conclusions taken from the work done for this dissertation as well as its limitations and possible improvements.

2 Context and state of the art

This section describes the context in which the problem is inserted as well as concepts, and technologies, useful for the understanding of this dissertation.

2.1 Context

There are various technologies that are traditionally considered too computationally expensive to use in real-time applications, but, as hardware's computational power keeps improving, their use in real-time applications is becoming increasingly common.

Soft body dynamics simulation is one of such technologies, a technology previously restricted to non-real-time uses, such as rendering computer-generated imagery (CGI) in movies, which is becoming increasingly common in real-time applications, such as video games.

This technology is currently implemented in ways that either don't give its users full control of how the simulation will affect the bodies or that are hard to use. Common ways consist of only allowing one set of physics variables per body or vertex group, or, in the case of a skinned mesh, one set per bone. In most cases, these variables have to be assigned manually by either editing a text file or numeric text boxes inside an application.

Due to the technology's increased use and lack of easy-to-use implementations, new approaches focused on fixing these issues should be developed.

2.2 Soft body dynamics simulation

2.2.1 What is soft body dynamics simulation?

Soft body dynamics simulation is a technology that simulates the physical deformation that happens to a body when a force is applied to it.

2.2.2 Uses for soft body dynamics simulation

The main use for soft body dynamics simulation is to increase the realism of a rendered scene. This technology is mainly used in movies and video games in order to accurately animate clothing and easily deformed bodies.

2.3 Polygon meshes

2.3.1 What is a polygon mesh?

A polygon mesh, also called mesh, is a collection of several sets of data that, together, define one or more 3D objects.

While the main data sets that define a polygon mesh are a vertex list, an edge list and a face list, it can contain additional data sets.

2.3.2 Uses for polygon meshes

Polygon meshes have multiple uses:

- Polygon meshes can be used to define areas or volumes for use in collision detection;
- Some rendering processes use polygon meshes to define the objects that will be rendered. In this case, the mesh may also contain:
    - An UV coordinates list – A list of 2D coordinates that each assign a vertex to a texture's coordinates;
    - A vertex normal list – A list with the normal direction of each vertex;
    - A smoothing groups list – A list of polygons that, if adjacent, will be shaded smoothly;
    - A materials list – A list with material identifications and with faces use them.

2.4 Texture mapping

2.4.1 What are textures?

Textures are image files used in graphical rendering. These images commonly have either three, red, green, and blue, or four, red, green, blue, and alpha, channels. Each channel stores its type

of color data in one byte for each pixel. Since a channel only uses a byte, an eight bit value, for a pixel's data, this means that the data varies between 0 and 255.

2.4.2 What is texture mapping?

Texture mapping is the process that connects the 2D positions of a texture to the 3D positions of a mesh.

2.4.3 Uses for texture mapping

Texture mapping is used a lot in 3D rendering, as current rendering technologies use various different textures for each mesh in order to create the wanted look.

2.5 Tessellation

2.5.1 What is tessellation?

Tessellating a polygon mesh is the act of reading a polygon mesh and creating new meshes from it. The new meshes are created by subdividing a single face, a triangle or quad, according to defined parameters and moving the newly created vertices. (Segal & Akeley, 2016)

2.5.2 Uses for tessellation

Tessellation is used to increase software real-time performance in a variety of ways (Microsoft Developer Network, 2017):

- By only storing a low-resolution mesh and using tessellation to create a mesh with higher detail, an application is able to render a highly detailed mesh while using less memory and bandwidth;
- Tessellation can be used to create meshes according to the distance to the camera, meaning that the closer meshes will be more detailed while the meshes that are farther away will be less detailed;
- Any calculations that are done on a polygon mesh can be done on a low-resolution mesh while what is rendered is a high-resolution mesh, saving performance.

2.6 Physics

2.6.1 Hooke's law

$$F = kX \tag{1}$$

Hooke's law, (1), describes how the force, F, needed to extend or compress a spring a certain distance, X, is calculated by multiplying the spring's stiffness, k, by X.

2.6.2 Newton's second law of motion

$$F = m\frac{d^2x}{dt^2} = ma \tag{2}$$

Newton's second law of motion, (2), describes that the inertial force, F, of an object is equal to its mass, m, times its acceleration, a, the second derivative of displacement, x, over time, t.

2.6.3 Damping

$$F = -cv \tag{3}$$

The damping force, F, generated during motion, is calculated by multiplying an object's damping coefficient, c, by its velocity, v. The resulting force will always point in the opposite direction of the object's motion.

2.7 Soft body dynamics models

Various methods to calculate soft body dynamics exist, each with its own advantages and disadvantages.

2.7.1 Finite element simulation

Finite element simulation uses the finite element method to calculate soft body dynamics. This is done by breaking the body into small elastic elements with several nodes, usually tetrahedrons, calculating the strain and stress each node is under, and then calculating new positions for each node so that overall stress is reduced as much possible. For a full description of how the finite element method may be used in soft body dynamics read 3D Model Deformation using Finite Elements by Michael Cairns. (Cairns, 2012)

2.7.2 Mass spring model

In the mass spring model, the physical body consists of a group of nodes connected by springs. In a simple implementation of soft body dynamics, using this model, each of the mesh's vertices may be considered a node and each of the edges considered a spring.

This model applies Hooke's law to the springs, calculating the forces, then uses the calculated forces to move the nodes with Newton's second law of motion. Other forces, e.g. Damping, may be added, in order to increase realism.

2.7.3 Shape matching

Shape matching is a method for calculating vertex movement in soft body dynamics that tries to restore the body's original shape by applying forces to moved vertices in the direction towards their original position. This method is very computationally inexpensive but is not nearly as physically accurate as other methods. (Muller, et al., 2005)

3 Existing soft body dynamics parametrization approaches

This section presents some soft body dynamics engines and details their approaches to parametrization.

3.1 Blender

Blender is a 3D content creation tool with various features, including modeling, rendering, animation, video editing, visual effects, compositing, texturing, rigging, simulations, and game creation. (Blender Documentation Team, 2017)

Parameterization in Blender's soft body dynamics system is done per object while allowing the use of vertex groups for added control. (Blender Documentation Team, 2017)

Simulation parameters are divided into three groups and are defined with values ranging from 0 to 1, assigned via a numeric text box.

As shown in Figure 1, the Soft Body parameters group consists of friction and mass, the Soft Body Goal parameters group consists of strength default, strength minimum, strength maximum, stiffness, and damping, and the Soft Body Edges parameters group consists of pull, push, damp, plastic, bending, and length.

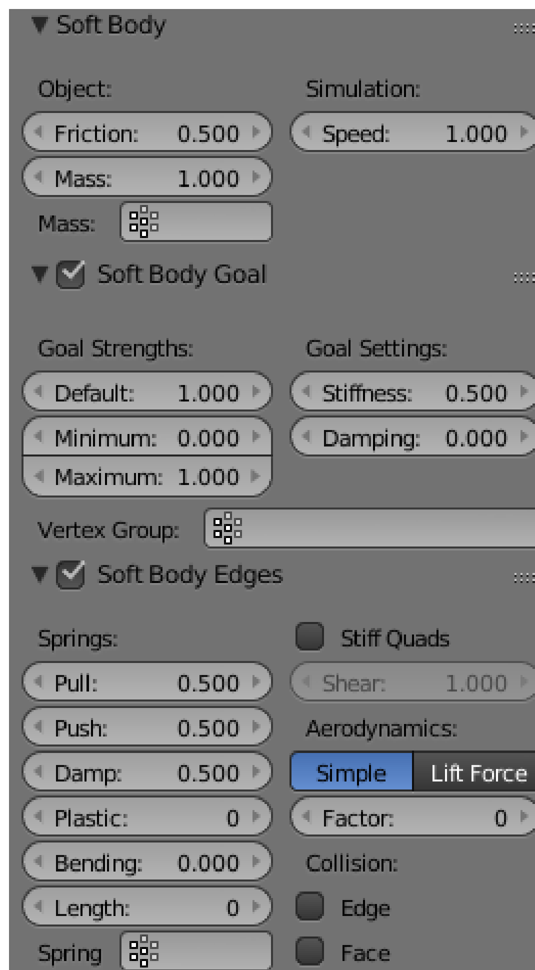


Figure 1 – Soft Body parameters in Blender 2.75

Blender's soft body dynamics system also allows for plastic deformation. This kind of deformation allows users to set a plasticity value which is used to calculate if a deformation will become permanent.

3.2 Carbon

Carbon is a high-performance physics framework that supports the simulation of both soft and rigid bodies. This framework is available as a C++ library and as a set of plug-ins for Houdini FX. (Numerion Software, 2017).

As exemplified in Figure 2, Carbon Tetrahedron allows for non-uniform parameterization of meshes with the use of paint maps (Numerion Software, 2017). However, there is no information on how this parametrization is done, nor on how it is stored, as there is no publicly available documentation.

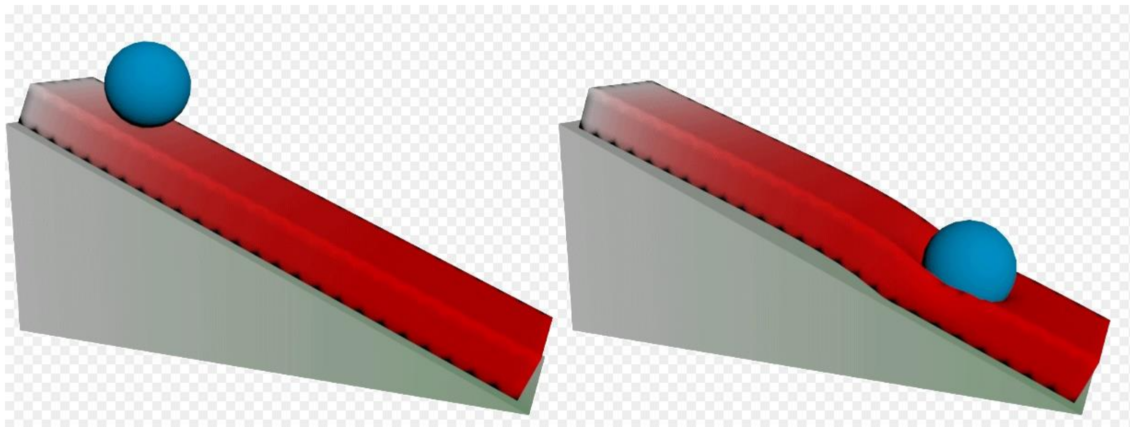


Figure 2 – Non-uniform mesh behavior in Carbon Tetrahedron (Numerion Software, 2017)

3.3 BeamNG.drive

BeamNG.drive is an open world vehicle simulator (BEAM NG INC, 2016) that uses a proprietary physics engine (Support.BeamNG, 2016).

Vehicle physics in BeamNG.drive are controlled with a skeleton constructed from nodes and beams, and is defined with a JBeam file. A JBeam file is a text file based on JSON (BEAM NG INC, 2017).

A node is a mesh's vertex, or group of vertices, and can have several different physics values, as shown in Code 1. These values are id, posX, posY, posZ, nodeWeight, collision, selfCollision, frictionCoef, nodeMaterial, fixed, surfaceCoef, volumeCoef, and pairedNode (BEAM NG INC, 2017).

Figure 3 shows an example of a group of nodes belonging to a vehicle.

```
"nodes" :[
    ["id", "posX", "posY", "posZ"],
    {"collision":true},
    {"selfCollision":true},
    {"nodeWeight":35},
    {"nodeMaterial":"|NM_PLASTIC"},
    {"group":"body"},
    ["n1rr", -0.90, -0.93, 0.23],
    ["n1ll", 0.90, -0.93, 0.23],
    {"group":""},
]
```

Code 1 – example of a nodes section with some properties in JBeam (BEAM NG INC, 2017)

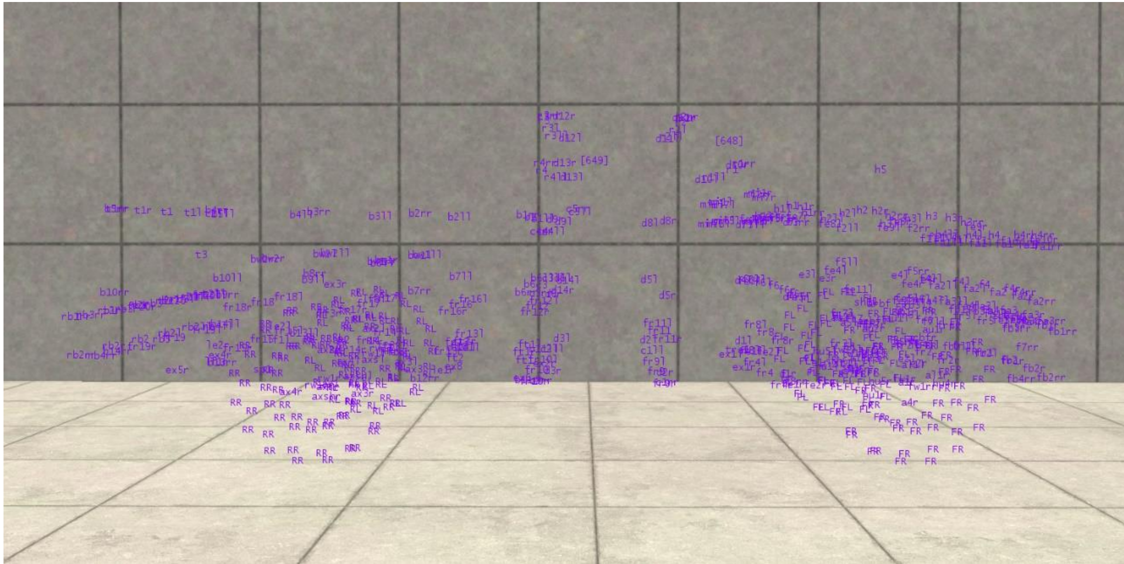


Figure 3 – A vehicle's nodes in JBeam (BEAM NG INC, 2017)

A beam acts as a connection between nodes, and can have several different physics values, as shown in Code 2. These values are id1, id2, beamType, beamStrength, beamSpring, beamDamp, beamDeform, beamPrecompression, breakGroup, and breakGroupType (BEAM NG INC, 2017).

```
"beams": [
      ["id1:", "id2:"],
      {"breakGroupType":1},
      {"breakGroup":"frame"},
      {"beamSpring":1251000,"beamDamp":250},
      {"beamDeform":16000,"beamStrength":24000},
      ["f1rr","f1r"],
      ["f1r","f1l"],
      ["f1l","f1ll"],
      {"breakGroupType":0},
      {"breakGroup":""},
]
```

Code 2 – example of a nodes section with some properties in JBeam (BEAM NG INC, 2017)

Depending on the type of beam, it can have added values. If the beam is anisotropic, a type of beam with variable behavior depending on whether it's being compressed or expanded, then those values are springExpansion, dampExpansion, beamLongBound, and transitionZone (BEAM NG INC, 2017).

If the beam is bounded, the added values are beamLongBound, beamShortBound, beamLimitSpring, beamLimitDamp, beamLimitDampRebound, beamDampRebound, beamDampFast, beamDampReboundFast, and beamDampVelocitySplit (BEAM NG INC, 2017).

If the beam is support, the only added value is beamLongBound (BEAM NG INC, 2017).

If the beam is pressured, the added values are pressure, surface, volumeCoef, and maxPressure (BEAM NG INC, 2017).

If the beam is l-beam, the only added value is id3 (BEAM NG INC, 2017).

Figure 4 shows an example of a group of nodes belonging to a vehicle.

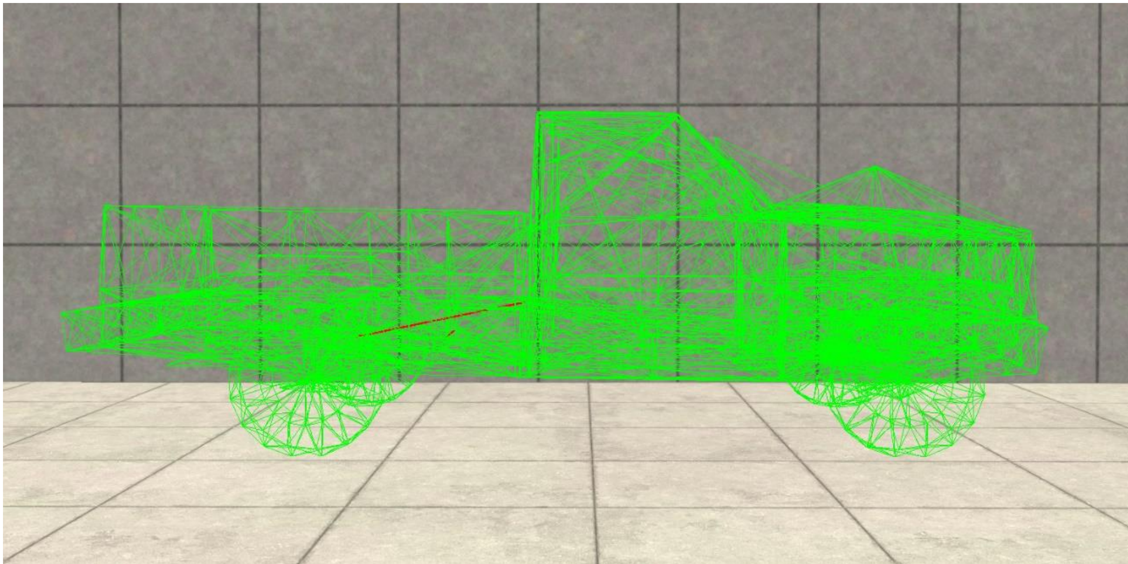


Figure 4 – A vehicle's beams in JBeam (BEAM NG INC, 2017)

3.4 PhysX Clothing

PhysX Clothing is a dynamic clothing simulation module for games that has authoring tool plug-ins for 3D Studio Max and Maya (Nvidia Corporation, 2017).

Parameterization in PhysX Clothing is done in different ways for different parameters. Some parameters are defined per object while others are defined per vertex.

The object variables are defined with values ranging from 0 to 1 and are assigned via a numeric text box. As shown by Figure 5, these variables consist of gravity scale, friction, bend resistance, shear resistance, stretch limit, relax, damping, drag, inertia blend, fiber compression, fiber expansion, and fiber resistance.

Figure 5 – Object variables in PhysX 3.4.0 Clothing plug-in for 3D Studio Max

The vertex variables are also defined with values ranging from 0 to 1 but are assigned with a paint tool where the user uses a numeric text box to choose a value and then uses the mouse to click on or near the vertices he wants to assign that value to. This operation also changes the vertex color of the affected vertices, effectively painting the mesh. These variables consist of maximum distance, backstop offset, and latch to nearest, as seen in Figure 6.

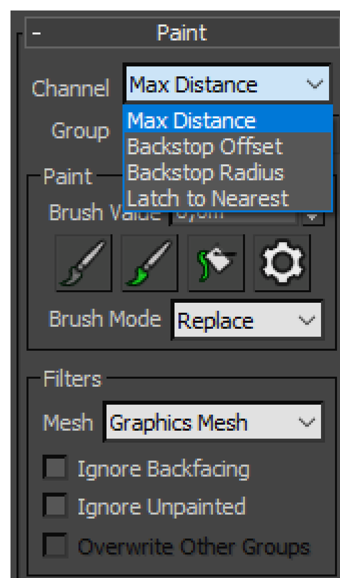


Figure 6 – Vertex variables in PhysX 3.4.0 Clothing plug-in for 3D Studio Max

Figure 7 shows that the mesh information and physical simulation parameters can be exported in two proprietary file types, either .apb or .apx. The main difference between these two file types is that .apb is binary while .apx is written in XML.

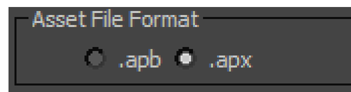


Figure 7 – File types for mesh information in PhysX 3.4.0 Clothing plug-in for 3D Studio Max 2015

3.5 Source

Source is a game engine with soft body dynamics capabilities (Valve Developer Community, 2012).

In the Source engine's soft body dynamics system, bones are used to add dynamic movement to a mesh. This type of bone is called a jigglebone (Valve Developer Community, 2013).

The Source engine stores its skeleton data in a binary .mdl file. This file is compiled from a text file in the Studiomdl Data format, from now on referred to as SMD (Valve Developer Community, 2016).

The compilation process requires a Quake C file, from now on referred to as QC. The QC file is a text file that contains a script that controls the compilation process, as is exemplified by Code 3 (Valve Developer Community, 2016) .

```
$modelname      "props_sdk\myfirstmodel.mdl"
$body mybody    "myfirstmodel-ref.smd"
$staticprop
$surfaceprop    combine_metal
$cdmaterials    "models\props_sdk"

$sequence idle  "myfirstmodel-ref.smd" // no animation wanted, so re-using the
reference mesh

$collisionmodel "myfirstmodel-phys.smd" { $concave }
```

Code 3 – A simple QC script (Valve Developer Community, 2016)

A skeleton's jigglebones are parametrized using the QC script used during its compilation and the parametrization syntax is shown in Code 4, where a physics variable is named property and are grouped in property groups.

```
$jigglebone <name> {
  <property group> {
         <property> <value> [<value>]
         …
  }
}
```

Code 4 – Jigglebone parametrization syntax in a QC script (Valve Developer Community, 2013)

The physics variables found in the is_flexible property group are, yaw_stiffness, yaw_damping, pitch_stiffness, pitch_damping, along_stiffness, along_damping, allow_length_flex, length, and tip_mass (Valve Developer Community, 2013).

The is_rigid property group consists of the length and the tip_mass physics variables (Valve Developer Community, 2013).

Both the is_flexible and the is_rigid groups may have the angle_constraint, yaw_constraint, yaw_friction, pitch_constraint, and the pitch_friction physics variables. (Valve Developer Community, 2013)

The has_base_spring property group can be used with either previous group and contains the stiffness, damping, left_constraint, left_friction, up_constraint, up_friction, forward_constraint, and the forward_friction physics variables (Valve Developer Community, 2013).

4 Design

This section shows the expected design of the soft body dynamics system that will be developed as part of this dissertation and explains any decisions made during the creation of this design.

4.1 Soft body dynamics model for the simulation module

Considering that the simulation module is being developed only for testing purposes, a simple implementation is the best choice for this dissertation.

Due to its simplicity, Shape matching, described in 2.7.3, was the chosen model. If a need for a level of realism not provided by this model is identified then it can be combined with the Mass spring model.

Using this model, when a vertex's world position is changed, the system will save the amount of movement done but keep the vertex's world position as it previously was and simulate its movement to its new world position. To add some realism to this motion, Hooke's law will be used in conjunction with Newton's second law of motion, and with added Damping.

If Newton's second law of motion states that $F = ma$ and Hooke's law states that $F = kX$, then their combination will be $ma = kX$. Adding Damping, $F = -cv$, will result in $ma = kX - cv$. Since an acceleration will be applied to the vertices in order to move them, the final equation is the following:

$$a = (kX - cv) \div m \tag{4}$$

In the final equation, m, k, and c, remain constant throughout the simulation and can be defined prior to it.

After the acceleration is calculated, the movement of the vertex needs to be calculated. This will be done by first calculating the vertex displacement using $d = v_0 t + \frac{1}{2} a t^2$, adding it to the

vertices position, and then calculating the new velocity using $v = v_0 + at$. This new velocity will be used as v_0 during the next simulation frame.

This process will be done for each of the three coordinate axes, x, y, and z.

4.2 Parametrization

4.2.1 Physics variables

For the data module to know what to expect when reading physics variables from a texture file we need to define what variable each texture channel will contain. Since both OpenGL and DirectX are designed to work with a maximum of four channels, red, green, blue, and alpha, per texture, each texture will contain, at most, four channels.

Three physics variables that remain constant throughout the simulation, mass, stiffness, and damping were identified. These variables will be saved in the texture.

Since only three variables have been identified, we can add an additional one, simulation weight. This variable will influence how much the vertex's world position changes will affect it, where the minimum value, 0, will make the system ignore the changes and the maximum value, 255, will make the system save the entirety of the movement but keep the vertex's world position unchanged. Any values in between will be used as a ratio for this movement, i.e. if the simulation weight is at a fifth of the maximum value and the vertex was moved ten units to the left then the system will instantly move the vertex to two units to the right of the new position and save a movement of two units.

Now that we have four variables we can distribute them through a texture's four channels. The alpha channel poses a small problem, as textures may not have one or have one that only accepts two values, 0 and 1. This means that it needs to be assigned an optional, or easily assumed, value.

Out of the four identified variables, the one that best fits these needs is the simulation weight. This is because, if the alpha channel doesn't exist, the maximum value can be assumed, and, if it doesn't accept in between values, it can work as an on or off switch.

Since the rest of the channels have no special needs, the remaining variables can be randomly assigned, as such, the red channel will store mass, the green channel will store stiffness, the blue channel will store damping and the alpha channel will store simulation weight.

4.2.2 Variable maximums

Since a texture channel can only store integer values between 0 and 255 a user may not be able to define big enough values, or the difference between a value and the next one, e.g. 15 and 16, may be too big and cause the need for a fraction. There's also the possibility that the minimum value is too low and the user needs to define a higher one.

All of these problems are solved by adding a way to define new minimum and maximum values, which will still be represented as 0 and 255 in the texture but be used in the simulation via the following equation:

$$value_{used} = (255 - value) * minimum + value * maximum \quad (5)$$

4.3 System modules

Due to the intent of using various game engines for testing, and due to the fact that each game engine has its own way of being told what system files to use, the developed system will need to be ported to each engine. This makes it much more difficult to test the system in a variety of engines.

A way to solve this issue is to separate the data acquisition from the simulation itself, creating two modules, one that does the soft body dynamics' work, a simulation module, and a data module that acts as a communication layer between the game engine and the simulation module.

Another advantage this solution has is that either the data module or the simulation module can be easily changed without affecting the other, allowing for testing with different types of simulation and data acquisition methods.

4.3.1 Simulation module

While there will only be one implementation of the simulation module, its design, shown in Figure 8, also has to follow a set of rules.

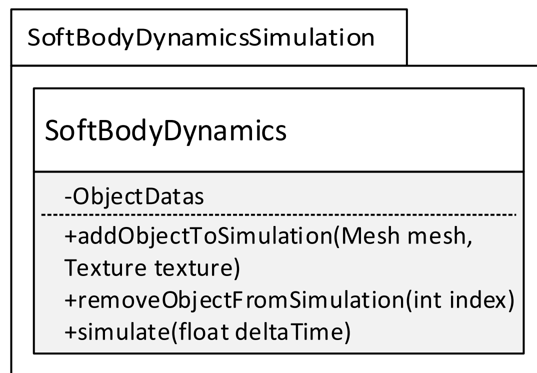


Figure 8 – Design of the simulation module

This module will:

- Contain a list of object data's which will include all of the data needed to run the simulation;
- Be notified when a new mesh and a new texture are to be added to the simulation, returning their array index;
- Be notified when a mesh and a texture are to be deleted;
- Be signaled to start a new frame of simulation;
- Modify the position an objects' vertices, according to the simulation.

4.3.2 Data module

The implementation of this module will vary depending on the engine but its design will always follow a set of rules. The design is shown in Figure 9.

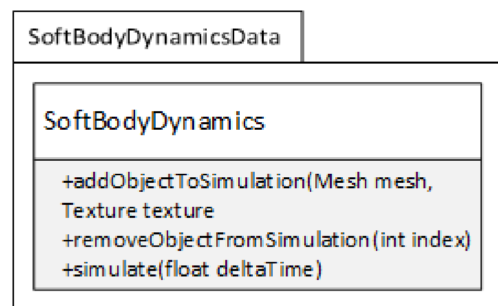


Figure 9 – Design of the data module

This module will:

- Notify the simulation module every time a new mesh and a new texture is received, sending it their data;
- Notify the simulation module every time a mesh and a texture is deleted, sending it their array index;
- Signal the simulation module to start a new frame of simulation, sending it the elapsed time and wait for the simulation to end;
- Receive the vertex changes from the simulation module and apply them to the mesh.

4.4 Application architecture

As shown in Figure 10, just like the design of each module, the system's design is very simple. The game engine and the simulation module are independent, with the data module depending on both of them.

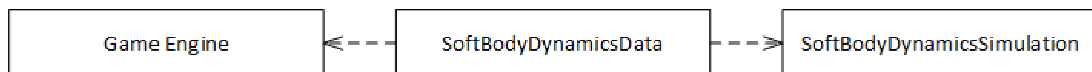


Figure 10 – Architecture of the soft body dynamics system

5 Implementation

This section presents the implementation of the developed soft body dynamics system and explains any decisions made during it.

5.1 Simulation module

This module was implemented in C++, for its performance and portability, and compiled into a DLL so it can easily be integrated into existing software.

The DLL's interface exposes the AddObjectToSimulation, ChangeObjectsPhysicsTexture, ChangeObjectsPhysicsVariableLimits, RemoveObjectFromSimulation, Simulate, and CleanSimulationData functions.

While this module only supports the simulation of static meshes, the implementation of both the simulation of skeletal meshes and collision detection was also planned. These two features weren't implemented due to development issues that weren't overcome within the deadline.

5.1.1 Object data structure

All the data needed to run the simulation on an object is contained within the ObjectData struct. This data is composed of:

- minMass, maxMass, minStiffness, maxStiffness, minDamping, maxDamping – Six floats representing the minimum and maximum limits for the simulation variables, mass, stiffness, and damping;

- uv, vertices, texturePixels – Pointers to three external arrays containing, respectively, the object's UV data, the object's vertex data, and the physics texture's color data. This module receives these pointers directly from the data module instead of copying them. In the case of the UV and color data this is done in order to improve the performance

related to adding and removing an object from the simulation. In the case of the vertex data, it's done so the simulation can directly alter a vertex's position without having to pass that data to the data module;

- velocity, originalVertices – Pointers to two internal arrays containing, namely, the velocity corresponding to each vertex, and a copy of the object's vertex data from when it was added to the simulation;
- vertexAmount, textureHeight, textureWidth – Three integers containing the size of several arrays. vertexAmount represents the size of the UV, vertex, and velocity arrays, while textureHeight and textureWidth represent the physics texture's two sizes, which, in the texture sampling function, are used in converting two-dimensional coordinates into a one-dimensional array index;
- position, rotation, scale – Three pointers to external data representing the object's world space transformation. This module receives these pointers directly from the data module instead of copying them so the data module doesn't need to pass a copy every time the simulation is run;
- previousPosition, previousRotation, previousScale – Three pointers to internal data representing the object's world space transformation during the last time the simulation was run;

5.1.2 Adding an object to the simulation

In order for the data module to add a new object to the simulation, it has to send the object's information to the simulation module using the AddObjectToSimulation function shown in Code 5. This function receives pointers to the object's position, rotation, scale, vertex array, UV array, and to the physics texture's color data. It also receives the number of vertices the object's mesh has, the physics texture's height and width, as well as the physics variables minimum and maximum limits.

```
int AddObjectToSimulation(Vector3* objectPosition, Quaternion* objectRotation,
Vector3* objectScale, Vector3* meshVertices, Vector2* meshUV, int vertexAmount,
Pixel* pixels, int textureHeight, int textureWidth, float minMass, float
maxMass, float minStiffness, float maxStiffness, float minDamping, float
maxDamping)
{
	ObjectData* od = new ObjectData();
	size_t vec3Size = sizeof(Vector3);
	int vertexArraySize = vertexAmount * vec3Size;
	SetupObjectData(od, objectPosition, objectRotation, objectScale,
meshVertices, meshUV, vertexAmount, pixels, textureHeight, textureWidth,
minMass, maxMass, minStiffness, maxStiffness, minDamping, maxDamping);
	od->originalVertices = (Vector3*)malloc(vertexArraySize);
	memcpy_s(od->originalVertices, vertexArraySize, meshVertices,
vertexArraySize);

	return _objects.Add(od);
}
```

Code 5 – The AddObjectToSimulation function

The AddObjectToSimulation function instantiates a new ObjectData object and uses the received data to initialize it, then adds the new object into the module's ObjectData list, returning its index. The data module can use this index with some of the simulation module's functions, in order to perform object related actions.

All the SetupObjectData function does is copy the received variables into the new ObjectData object.

5.1.3 Modifying an object's physics variables during runtime

In order for an object's physics texture, or the physics variable limits, to be modified, the data module has to use the respective functions, ChangeObjectsPhysicsTexture and ChangeObjectsPhysicsVariableLimits. These functions receive the object's index and its related data, check if an object with that index exist and, if it does, replace its data with the new one, as shown in Code 6.

```
void ChangeObjectsPhysicsTexture(int index, Pixel* pixels, int textureHeight,
int textureWidth)
{
	if (_objects[index] != nullptr)
	{
		_objects[index]->texturePixels = pixels;
		_objects[index]->textureHeight = textureHeight;
		_objects[index]->textureWidth = textureWidth;
	}
}

void ChangeObjectsPhysicsVariableLimits(int index, float minMass, float
maxMass, float minStiffness, float maxStiffness, float minDamping, float
maxDamping)
{
	if (_objects[index] != nullptr)
	{
		_objects[index]->minMass = minMass;
		_objects[index]->maxMass = maxMass;
		_objects[index]->minStiffness = minStiffness;
		_objects[index]->maxStiffness = maxStiffness;
		_objects[index]->minDamping = minDamping;
		_objects[index]->maxDamping = maxDamping;
	}
}
```

Code 6 – Functions used in order to modify an object's physics variables during runtime

Since all the variables modified by either of these two functions are used as is during the simulation, the changes take effect immediately, enabling the modification of an object's physics variables during runtime.

5.1.4 Removing an object from the simulation

In order for an object to be removed from the simulation, the data module has to use the RemoveObjectFromSimulation function, passing to it the object's index. This function first deletes the object then removes its pointer from the objects list, as shown in Code 7.

```
void RemoveObjectFromSimulation(int index)
{
	delete _objects[index];
	_objects.Remove(index);
}
```

Code 7 – Function used in order to remove an object from the simulation

While debugging the simulation module using the Unity Editor, it was noticed that the DLL wasn't unloaded between executions, making a second execution include pointers to, now invalid, data. In order to fix this, a CleanSimulationData function, intended to be called when an execution is being stopped, was implemented. This function iterates through the objects list, deleting each one and then removes all the pointers the list, as shown in Code 8.

```
void CleanSimulationData()
{
        for (int i = 0; i < _objects.GetLength(); ++i)
        {
                delete _objects[i];
        }

        _objects.Clear();
}
```

Code 8 – Function used in order to remove all objects from the simulation

5.1.5 Processing the simulation

In order for the simulation to advance, the data module has to use the Simulate function, passing it the size of the time step to be simulated, giving the software more flexibility than if it used a fixed time step.

The simulation algorithm starts by calculating the difference between the object's transformations since the last time the simulation was run, as shown in Code 9.

```
for (int i = 0; i < _objects.GetLength(); ++i)
{
        od = _objects[i];
        pDelta = *od->position - *od->previousPosition;
        pDelta /= *od->scale;
        RotateVector3(&pDelta, &od->rotation->Inversed());
        rDelta = od->rotation->Inversed() * *od->previousRotation;
        sDelta = *od->scale - *od->previousScale;

        for (int j = 0; j < od->vertexAmount; ++j)
        {
                //Simulation code
        }

        od->previousPosition->x = od->position->x;
        od->previousPosition->y = od->position->y;
        od->previousPosition->z = od->position->z;

        od->previousRotation->x = od->rotation->x;
        od->previousRotation->y = od->rotation->y;
        od->previousRotation->z = od->rotation->z;
        od->previousRotation->w = od->rotation->w;

        if(od->scale->x != 0) od->previousScale->x = od->scale->x;
        if(od->scale->y != 0) od->previousScale->y = od->scale->y;
        if(od->scale->z != 0) od->previousScale->z = od->scale->z;
}
```

Code 9 – Code that calculates an object's transformation differences

In this code excerpt, pDelta is a three-dimensional vector that represents the difference between the objects' current and previous positions and is calculated by a simple subtraction. It is then divided by the objects current scale and rotated by the inverse of the current rotation converting it to world space.

In the same code excerpt, rDelta is a quaternion that represents the difference between the objects' current and previous rotations and is calculated by multiplying the inverse of the current rotation with the previous rotation.

Also in the same code excerpt, sDelta is a three-dimensional vector that represents the difference between the objects' current and previous scales, however it isn't accurate as scale is multiplicative. The calculation is done this way so we can apply the simulation weight to this difference and then use it to calculate the correct difference, as seen in Code 10.

After the simulation is done, the current transformations are saved. In the case that the current scale is zero, the value isn't saved in order to avoid division by zero, trading some accuracy for stability.

The reverse of each transformation is then applied to each individual vertex, causing them to be moved to their previous world space position, as shown in Code 10. However, before each vertex is moved its UV coordinates are used to sample the physics texture and the returned alpha value, the simulation weight, is used to determine how much the transformations will affect the vertex.

```
RotateVector3(&od->vertices[j], &rDelta);
od->vertices[j] /= ((sDelta * sample.a) + *od->previousScale) / *od-
>previousScale;
od->vertices[j] -= pDelta * sample.a;
```

Code 10 – Code that moves a vertex to its previous world space position

In this code excerpt sample represents the color information relevant to this vertex and sample.a is its simulation weight. This code also contains the correct calculation of the scale difference.

After this, the equations described in section 4.1 are used in order to accelerate and move the vertices into their new positions, as shown in Code 11.

```
	for (int j = 0; j < od->vertexAmount; ++j)
	{
		sample = SampleTexture(od->texturePixels, &od->uv[j], od->textureWidth,
od->textureHeight);
		sample.r = ((1.0f - sample.r) * od->minMass) + (sample.r * od-
		>maxMass);
		sample.g = ((1.0f - sample.g) * od->minStiffness) + (sample.g * od-
	>maxStiffness);
		sample.b = ((1.0f - sample.b) * od->minDamping) + (sample.b * od-
		>maxDamping);

		if (sample.r == 0 || sample.a == 0) continue;

		RotateVector3(&od->vertices[j], &rDelta);
		od->vertices[j] /= ((sDelta * sample.a) + *od->previousScale) / *od-
	>previousScale;
		od->vertices[j] -= pDelta * sample.a;

		if (sample.g == 0) continue;

		od->vertices[j] += od->velocity[j] * deltaTime;

		vDelta = od->originalVertices[j] - od->vertices[j];

		accel = (vDelta * sample.g - od->velocity[j] * sample.b) / sample.r;
		od->vertices[j] += accel * 0.5f * sqrDeltaTime;
		od->velocity[j] += accel * deltaTime;
	}
```

Code 11 – Simulation code

In this code excerpt, SampleTexture is used in order to get a copy of the color information related to this vertex from the physics texture. This sample is then modified according to equation (5), with 255 replaced by 1.0f as, in this case, the color data is represented by floats that vary between 0 and 1. If either the mass or the simulation weight are zero, the simulation won't actually affect the vertex.

Afterwards, the vertex transformations are reversed, and if the stiffness is zero, the simulation skips the rest of the calculations, as the vertex will never return to its original position.

Lastly, the equations described in section 4.1 are applied. First, the velocity calculated in the previous execution is applied to the vertex. Then a new acceleration is calculated and added to this velocity.

5.2 Data module

Due to familiarity, the data module was implemented in Unity. This module was divided into two scripts.

An Unreal Engine 4 implementation was also planned but was eventually put aside due to time constraints.

One of the scripts, the object script, is attached to the objects that are to be simulated and allows for their physics texture, as well as their minimum and maximum variables, to be set.

The other script, the simulation script, is the one who interfaces with the simulation module.

5.2.1 PhysicsObject script

This script contains all the object specific variables, the physicsTexture, minMass, maxMass, minStiffness, maxStiffness, minDamping, maxDamping, and whatever indexes this object may have in the simulation.

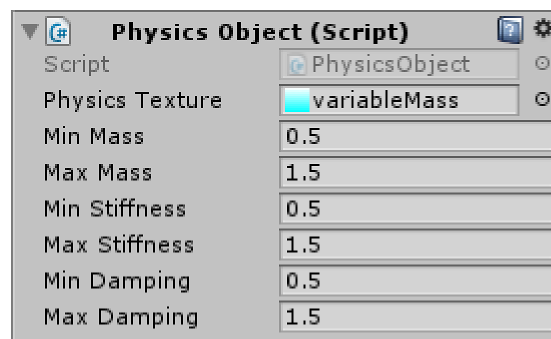


Figure 11 – The object script as it appears in the Unity editor

As shown in Figure 11, all the variables are editable. To make sure that all the variables are positive and that the minimums are lower than the maximums, the script makes use of Unity's OnValidate function, which is invoked every time a variable is changed in the editor. If the editor is playing, then it also notifies the simulation script, which will then notify the simulation module. This validation is shown in Code 12.

```
void OnValidate()
{
        if (minMass      < 0) minMass      = 0;
        if (maxMass      < 0) maxMass      = minMass;
        if (minStiffness < 0) minStiffness = 0;
        if (maxStiffness < 0) maxStiffness = minStiffness;
        if (minDamping   < 0) minDamping   = 0;
        if (maxDamping   < 0) maxDamping   = minDamping;

        if (minMass      > maxMass)      minMass      = maxMass;
        if (minStiffness > maxStiffness) minStiffness = maxStiffness;
        if (minDamping   > maxDamping)   minDamping   = maxDamping;

        if (!Application.isPlaying) return;
        if (!initialValidationDone)
        {
                initialValidationDone = true;
                return;
        }

        if(physicsTexture != null)
        PhysicsSimulation.ChangePhysicsTexture(physicsTexture, indexes);
        PhysicsSimulation.ChangeObjectsPhysicsVariableLimits(minMass, maxMass,
 minStiffness, maxStiffness, minDamping, maxDamping, indexes);
}
```

Code 12 – PhysicsObject script's validations

5.2.2 PhysicsSimulation script

This script uses the DllImport attribute to import the exposed functions in the simulation module, as shown in Code 13. The declared function has to match the declaration in the DLL.

```
[DllImport("PhysicsCore", EntryPoint = "AddObjectToSimulation")]
private static extern int AddObjectToSimulation(ref Vector3 position, ref
Quaternion rotation, ref Vector3 scale, Vector3[] meshVertices, Vector2[]
meshUV, int vertexAmount, Color[] pixels, int textureHeight, int textureWidth,
float minMass, float maxMass, float minStiffness, float maxStiffness, float
minDamping, float maxDamping);
[DllImport("PhysicsCore", EntryPoint = "ChangeObjectsPhysicsTexture")]
private static extern void ChangeObjectsPhysicsTexture(int index, Color[]
pixels, int textureHeight, int textureWidth);
[DllImport("PhysicsCore", EntryPoint = "ChangeObjectsPhysicsVariableLimits")]
private static extern void ChangeObjectsPhysicsVariableLimits(int index, float
minMass, float maxMass, float minStiffness, float maxStiffness, float
minDamping, float maxDamping);
[DllImport("PhysicsCore", EntryPoint = "RemoveObjectFromSimulation")]
private static extern void RemoveObjectFromSimulation(int index);
[DllImport("PhysicsCore", EntryPoint = "Simulate")]
private static extern void Simulate(float deltaTime);
[DllImport("PhysicsCore", EntryPoint = "CleanSimulationData")]
private static extern void CleanSimulationData();
```

Code 13 – Importing the simulation module functions in the data module

It also contains a list with pointers to all the data shared with the simulation module and is the one responsible for communicating with it, making sure that all the data is sent correctly, also making sure that it's used correctly within Unity.

To do this, the scripts starts by gathering all objects that have an object script attached to them, as shown in Code 14, and then uses the AddObject function to add them to the simulation.

```
PhysicsObject[] objects = FindObjectsOfType(typeof(PhysicsObject)) as
PhysicsObject[];
for (int i = 0; i < objects.Length; i++)
{
       if (objects[i].physicsTexture != null)
       {
              objects[i].ObjectIndexes = AddObject(objects[i].gameObject,
 objects[i].physicsTexture, objects[i].minMass, objects[i].maxMass,
 objects[i].minStiffness, objects[i].maxStiffness, objects[i].minDamping,
 objects[i].maxDamping);
       }
}
```

Code 14 – Code that gathers every object to be added to the simulation

In order prepare for the object to be added, the AddObject function starts by gathering all the meshes attached to the object, copying and saving their data, and then sending pointers to it to the simulation module, as shown in Code 15.

```
public static int[] AddObject(GameObject, Texture2D texture, float minMass,
float maxMass, float minStiffness, float maxStiffness, float minDamping, float
maxDamping)
{
        List<int> indexes = new List<int>();
        var filters = gameObject.GetComponentsInChildren<MeshFilter>();
        if (filters != null)
        {
                foreach (var filter in filters)
                {
                        filter.mesh.MarkDynamic();
                        ObjectData meshData = new ObjectData();
                        meshData.mesh = filter.mesh;
                        meshData.vertices = meshData.mesh.vertices;
                        meshData.uv = meshData.mesh.uv;
                        meshData.pixels = texture.GetPixels();
                        meshData.transform = filter.gameObject.transform;
                        meshData.position =
filter.gameObject.transform.position;
                        meshData.rotation =
filter.gameObject.transform.rotation;
                        meshData.scale = filter.gameObject.transform.lossyScale;

                        int index = PhysicsSimulation.AddObjectToSimulation(ref
meshData.position, ref meshData.rotation, ref meshData.scale,
meshData.vertices, meshData.uv, meshData.mesh.vertexCount, meshData.pixels,
texture.height, texture.width, minMass, maxMass, minStiffness, maxStiffness,
minDamping, maxDamping);
                        meshData.index = index;
                        datas.Add(meshData);
                        indexes.Add(index);
                }
        }
        return indexes.ToArray();
}
```

Code 15 – PhysicsSimulation script's AddObject function

Something of note is that the function MarkDynamic is called on the mesh, this flags it as a continually updated mesh, increasing the performance related to modifying it. There's also two variables that aren't sent to the simulation module, the transform, which is needed in order to update the transformation related pointers, and the index, which is returned by the simulation module and will be saved in the object script.

In order to update the simulation, Unity's Update, a function invoked every execution frame, is used. In this function, shown in Code 16, the simulation script starts by updating every objects' transformation pointers, the pointers sent to the simulation module, with the current transformations. Afterwards, the simulation module's Simulate function is invoked, passing to it Unity's Time.deltaTime, the time in seconds that the previous frame took to execute. Since the vertex array pointer used by the simulation module only points to a copy, the mesh's vertex array is then updated with the one that was modified by the simulation module.

```
void Update()
{
        for (int i = 0; i < datas.Count; i++)
        {
               datas[i].position = datas[i].transform.position;
               datas[i].rotation = datas[i].transform.rotation;
               datas[i].scale = datas[i].transform.lossyScale;
               UpdateMesh(datas[i]);
        }

        PhysicsSimulation.Simulate(Time.deltaTime);

        for (int i = 0; i < datas.Count; i++)
        {
               datas[i].mesh.vertices = datas[i].vertices;
        }
}
```

Code 16 – Code run every frame in Unity's Update function

Since Unity doesn't unload the simulation module's DLL between executions, special care needs to be taken to make sure all of its data is cleared from memory every time the execution is stopped. Since Unity's OnApplicationQuit function is invoked at the end of execution, it is used to invoke the simulation module's CleanSimulationData, clearing its data.

6 Evaluation

This section describes how this dissertation was evaluated, namely, it describes the testing procedure and how the results of these test were gathered, evaluated, and what the conclusions were.

6.1 Testing procedure

The objective of this dissertation is to develop a new approach to soft body dynamics parameterization that, in comparison to existing approaches, gives more control to its users while being easier to use.

As these goals are subjective, a test group composed of people experienced in working with existing approaches was gathered and their opinion was the deciding factor in whether the goals were accomplished or not. While multiple development and testing iterations were planned, issues related to both development and finding testers meant that only one round of testing was done.

The test group was only asked to try the developed system and answer a survey after development had ended. This survey included questions focused on the dissertation's goals, tester experience, as well as system performance. The reason the survey included questions related to system performance, even if they aren't related to the proposed objectives, was to get an idea of the overall usability of the system.

6.1.1 Test project

After development ended, the test group was asked to try the developed system. To help with this, a Unity project was prepared. The testers were asked to either modify this project or to implement the system into another program and test it.

The Unity project, whose screen is presented in Figure 12, has twenty-four objects, each a sphere scaled horizontally in order to stretch it, divided into six columns and four rows.

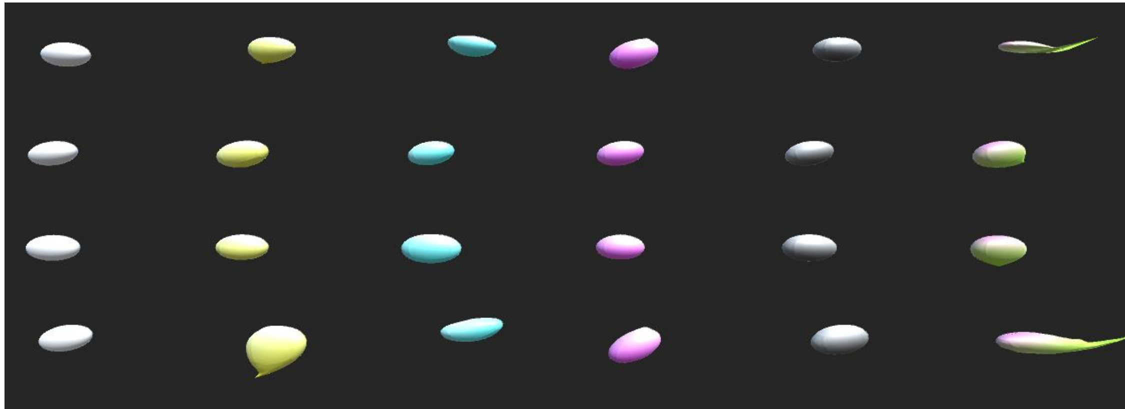


Figure 12 – Unity test project in motion

Each row transforms the objects in different way. The first translates them in the X and Y axes, the second rotates them in the Z axis, the third scales them in the X and Y axes, and the fourth combines all the previous ones.

The objects in each column except for the first have a physics texture specific to that column, those textures being, from left to right, variable damping, variable mass, variable stiffness, all previous combined, and everything randomly variable. The first column is not part of the simulation and serves as a way to visualize the transformations being applied to each row.

6.1.2 Survey questions

After the test group tried the system, they answered a survey. The survey included questions related to the tester's personal experience working with soft body dynamics systems, system performance as well as questions related to how it compares to other soft body dynamics systems.

The survey's questions were:

- What kind of experience do you have with existing soft body dynamics systems? – Multiple choice question with the following available answers: Professional, Amateur, Other;
- For how long have you been using soft body dynamics systems? – Multiple choice question with answers available in increments of one year, ranging from Less than 1 year to More Than 5 years;
- How many different soft body dynamics systems have you used? – Multiple choice question with answers available in increments of one, ranging from None to More Than 3;

- In what engine did you test the system? (Check all that apply) – Checkbox question with the following available answers: Unity 5, Unreal Engine 4, Other;
- Did you feel like the system's performance was limiting what you could do? – Multiple choice question with the following available answers: Yes, No;
- How hard is it to add an object to the simulation? – A linear scale question, ranging from 0 to 10, from Very Easy to Very Hard;
- How hard is it to create a new physics texture? – A linear scale question, ranging from 0 to 10, from Very Easy to Very Hard;
- How hard is it to edit an existing physics texture? – A linear scale question, ranging from 0 to 10, from Very Easy to Very Hard;
- In comparison to existing soft body dynamics systems, how hard was it to set all the simulation variables? – A linear scale question, ranging from 0 to 10, from Very Easy to Very Hard;
- In comparison to existing soft body dynamics systems, how much control over the simulation did you have? – A linear scale question, ranging from 0 to 10, from Very Little to Very High;
- What kind of improvement would you most like to see? – Multiple choice question with the following available answers: Better performance, Collision between objects or with self, Gravity simulation, More realistic physics simulation, Skeletal animation support, Use tessellation to increase simulation accuracy on highly deformed polygons, Other;
- If you have any comment about the system please leave it here – Optional free text question.

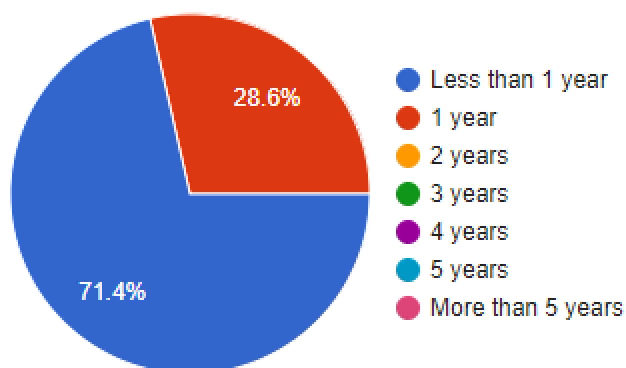
These surveys were analyzed and their results were used to understand if the dissertation's goals were accomplished or not.

6.2 Result analysis

Unfortunately, the test group was composed of only seven people who are self-identified amateurs. As such, the results of this survey are closer to general guidelines than to a definitive answer as to whether the developed system gives more control to its users while being easier to use in comparison to existing approaches. However, for the purposes of this dissertation, the results were still analyzed.

On the question "For how long have you been using soft body dynamics systems?", five people answered "Less than 1 year" and two answered "1 year", as seen in Figure 13. For the question

"How many different soft body dynamics systems have you used?", also seen in Figure 13, five people answered "1", one answered "2", and one answered "None".



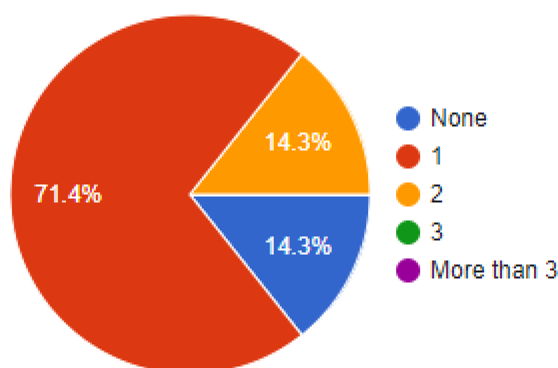


Figure 13 – Survey results for the questions "For how long have you been using soft body dynamics systems?" and "How many different soft body dynamics systems have you used?"

From these results, it's possible to extrapolate that the test group does not have much experience in using soft body dynamics systems.

The answers were unanimous in both the "In what engine did you test the system?" and the "Did you feel like the system's performance was limiting what you could do?" questions, with the former's answer being "Unity 5", the engine in which the test project is built, and the latter's being "No".

For the questions related to ease of use, results seen in Figure 14, the answers to the questions "How hard is it to add an object to the simulation?" and "In comparison to existing soft body dynamics systems, how hard was it to set all the simulation variables?" tend to the zero to two range, meaning that the testers considered the system easy to use. However, the answers to the questions "How hard is it to create a new physics texture?" and "How hard is it to edit an existing physics texture?" fell into the three to six range, meaning that the testers had trouble

working with the textures themselves, this could be for a variety of reasons, including user inexperience with texture painting and the complexity of editing each layer individually.

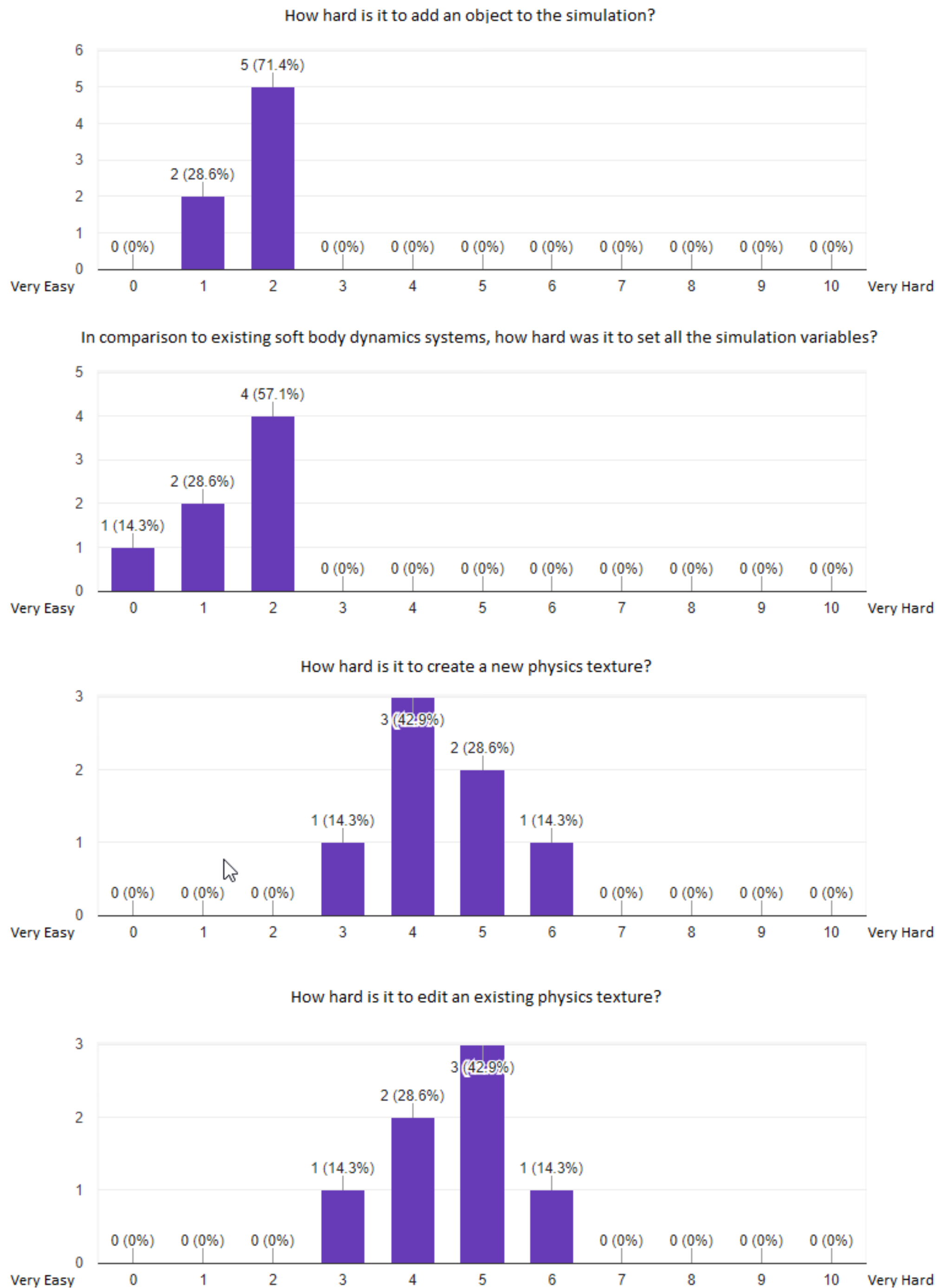


Figure 14 – Survey's ease of use questions

In the question "In comparison to existing soft body dynamics systems, how much control over the simulation did you have?", the answers tend to the six to ten range, as shown in Figure 15, meaning that the testers found that the developed system gives more control than existing ones.

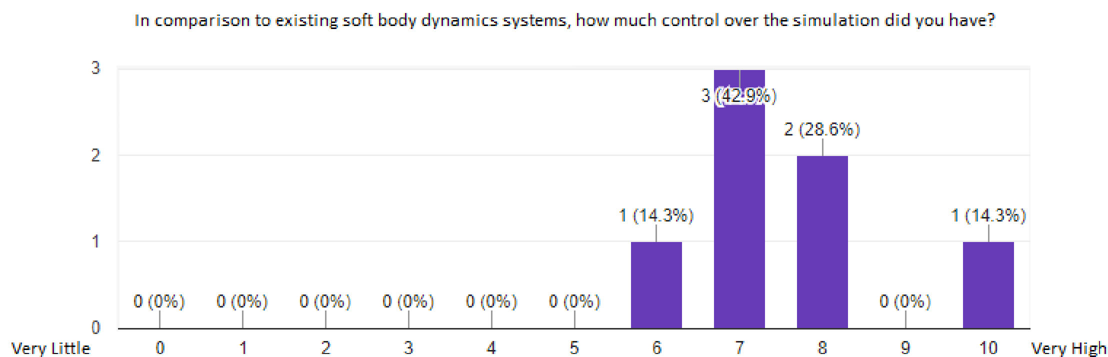


Figure 15 – Survey's answer about the amount of control over the simulation the system provides

In regards to the question "What kind of improvement would you most like to see?", seen in Figure 16, five people answered "Skeletal animation support", one person answered "More realistic physics simulation", and one answered "Collision between objects or with self".

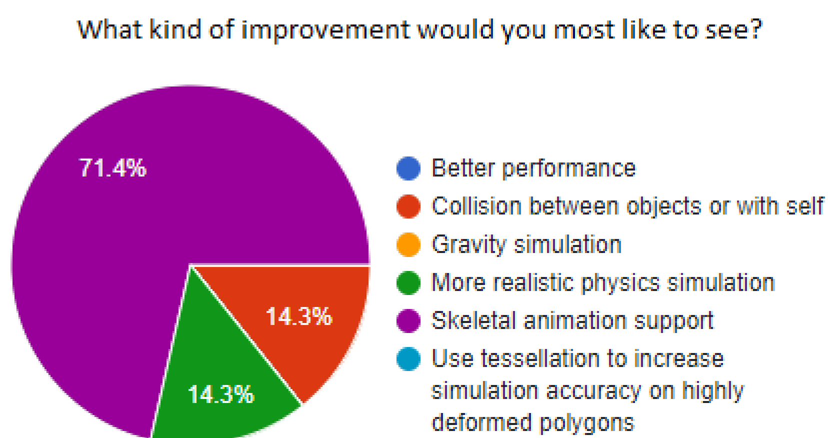


Figure 16 – Survey's answers to most wanted improvement

The last question, which asked for the tester's comments, did not get any answer.

While the survey's overall results very positive, the low number and high inexperience of the testers mean that these results could be misleading.

7 Conclusion

The purpose of this dissertation was to design and develop a new approach to soft body physics parameterization. This approach would use textures in order to allow an artist to set a mesh's physics simulation parameters individually for each vertex and be easier to use while allowing more control over the simulation.

In order to test this approach, a system composed of two self-contained modules was developed.

One module, the simulation module, receives the meshes' data and processes the simulation. This module was developed in C++ and compiled into a DLL.

The other module, the data module, is implemented in the software the system is being integrated to and acts as an intermediary between the software and the simulation module, making sure that the data is being correctly sent to the simulation module and that the simulation results are being correctly applied in the software. This module was developed in and integrated into Unity 5, using C#.

In order to know if this system is truly easier to use and gives more control than existing systems, a test group was tasked with trying the system and answering a survey. While the survey's results were very positive, in reality it's inconclusive due to the small and inexperienced group of testers.

While there were some issues during development, namely, not being able to find a way to correctly calculate the physics due to the vertices' coordinate system changes caused by skeletal animation, and not being able to find a way to properly solve a triangle-triangle intersection after it has been detected, the developed system has all the functionality needed to meet the dissertation's objectives.

7.1 Limitations and future work

While the current system implements all of the fundamental functionality, it could be greatly improved by implementing a number of changes:

- Change the mathematical model used to calculate the physics to a more physically accurate model;
- Support for skeletal animation;
- Support for world-collision and self-collision;
- Support for both constant or unstable forces, such as gravity and wind, respectively;
- Increase deformation accuracy on highly deformed polygons with tessellation;
- Parallelize the simulation algorithm.

Lastly, the system also needs to be properly tested by a large and experienced group, in order to determine if this dissertation's goal was achieved or not.

References

BEAM NG INC, 2016. *BeamNG | BeamNG Dev Blog.* [Online]
Available at: http://blog.beamng.com/
[Accessed 23 February 2017].

BEAM NG INC, 2017. *JBeam - BeamNG.* [Online]
Available at: https://wiki.beamng.com/JBeam
[Accessed 23 February 2017].

BEAM NG INC, 2017. *JBeam Introduction - BeamNG.* [Online]
Available at: https://wiki.beamng.com/JBeam_Introduction
[Accessed 23 February 2017].

BEAM NG INC, 2017. *JBeam Reference - BeamNG.* [Online]
Available at: https://wiki.beamng.com/JBeam_Reference
[Accessed 22 February 2017].

Blender Documentation Team, 2017. *Introduction — Blender Manual.* [Online]
Available at:
https://docs.blender.org/manual/en/dev/getting_started/about/introduction.html
[Accessed 22 February 2017].

Blender Documentation Team, 2017. *Settings — Blender Manual.* [Online]
Available at: https://docs.blender.org/manual/en/dev/physics/soft_body/settings.html
[Accessed 22 February 2017].

Cairns, M., 2012. *3D Model Deformation using Finite Elements.* [Online]
Available at:
https://nccastaff.bournemouth.ac.uk/jmacey/MastersProjects/MSc12/Cairns/finite_element_deformation.pdf
[Accessed 25 February 2017].

Microsoft Developer Network, 2017. *Tessellation Stages (Windows).* [Online]
Available at: https://msdn.microsoft.com/en-us/library/ff476340(v=VS.85).aspx
[Accessed 10 February 2017].

Muller, M., Heidelberger, B., Teschner, M. & Gross, M., 2005. *Final Report "Meshless Deformations Based on Shape Matching" for "COMPSCI 715 S2 C - Advanced Computer Graphics".* [Online]
Available at:
https://www.researchgate.net/profile/Matthias_Teschner/publication/220184721_Meshless_deformations_based_on_shape_matching/links/54a95f0e0cf256bf8bb95b36.pdf
[Accessed 21 October 2017].

Numerion Software, 2017. *Carbon Technology.* [Online]
Available at: http://www.numerion-software.com/~numerion/index.php/technology
[Accessed 22 February 2017].

Numerion Software, 2017. *Carbon Tetrahedron.* [Online]
Available at: http://www.numerion-software.com/~numerion/index.php/carbon-plugins/carbon-tetrahedron
[Accessed 22 February 2017].

Nvidia Corporation, 2017. *PhysX Clothing | NVIDIA Developer.* [Online]
Available at: https://developer.nvidia.com/clothing
[Accessed 22 February 2017].

Segal, M. & Akeley, K., 2016. *OpenGL 4.5 (Core Profile) - October 24, 2016.* [Online]
Available at: https://khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf
[Accessed 10 February 2017].

Support.BeamNG, 2016. *Forums | BeamNG.* [Online]
Available at: https://www.beamng.com/threads/anyone-agree-that-the-beamng-devs-need-to-add-more-realistic-efficient-shaders.26312/page-2#post-380019
[Accessed 23 February 2017].

Valve Developer Community, 2012. *Source Engine Features - Valve Developer Community.* [Online]
Available at: https://developer.valvesoftware.com/wiki/Source_Engine_Features
[Accessed 23 February 2017].

Valve Developer Community, 2013. *$jigglebone - Valve Developer Community.* [Online]
Available at: https://developer.valvesoftware.com/wiki/$jigglebone
[Accessed 23 February 2017].

Valve Developer Community, 2016. *QC - Valve Developer Community.* [Online]
Available at: https://developer.valvesoftware.com/wiki/Qc
[Accessed 23 February 2017].

Valve Developer Community, 2016. *Studiomdl Data - Valve Developer Community.* [Online]
Available at: https://developer.valvesoftware.com/wiki/SMD
[Accessed 23 February 2017].