Physical Experiments in School with Unity

Cable Simulation

Marian-Alexander Amiragov

Johannes Gutenberg University Mainz

Institute of Computer Science

Physical Experiments in School with Unity

Cable Simulation

Marian-Alexander Amiragov

1. Reviewer	Dr. Stefan Endler Faculty 08 (Computer Science) JGU Mainz
2. Reviewer	Prof. Dr. Klaus Wendt Faculty 08 (Physics) JGU Mainz
Supervisors	Dr. Stefan Endler and Prof. Dr. Klaus Wendt

February 2, 2020

Marian-Alexander Amiragov
Physical Experiments in School with Unity - Cable Simulation
Date: February 2, 2020
Reviewers: Dr. Stefan Endler and Prof. Dr. Klaus Wendt
Supervisors: Dr. Stefan Endler and Prof. Dr. Klaus Wendt

Johannes Gutenberg University Mainz
Institute of Computer Science

Abstract

This thesis attempts to conceptualize a usable and customizable simulation of cable physics. It was implemented as a Unity Component for *Virtual-Reality-Experiments (VRE)* - a simulation environment for selected physical experiments. Splines are used to define the initial state of the cable. Different types of splines were compared to each other. The Catmull-Rom Spline became thee spline of choice because of its smooth curvature and the fact that changing one of its control points only affects the neighbouring region. Then, some of the systems to simulate the cable were discussed and each of them implemented. The ones working best were the Jakobsen, PBD and XPBD based systems. XPBD offers some additional advantages. The highest time complexity is expected to be $O(n^2)$. Further work can be done in technical optimizations and parallelisation.

Contents

1 **Introduction** **1**
- 1.1 Motivation 1
- 1.2 Goal 1

2 **Theory** **3**
- 2.1 Curves & Splines 3
  - 2.1.1 Curves 3
  - 2.1.2 Splines 3
  - 2.1.3 Rendering Function 5
  - 2.1.4 Normalization 5
  - 2.1.5 Natural Cubic Splines 5
  - 2.1.6 Bézier Curves & Splines 6
  - 2.1.7 (Extended Cubic) Catmull-Rom Splines 6
- 2.2 Physics, Simulation and Algorithms 8
  - 2.2.1 Physics, Force & Mass in Game Engines 8
  - 2.2.2 Cable Model 9
  - 2.2.3 Algorithms Dependant on Unity's Ecosystem 10
  - 2.2.4 Physics, Variable Forces and Collisions in Game Engines . . . 12
  - 2.2.5 Constraint Solving Algorithms 14
  - 2.2.6 Other Systems and Algorithms 18

3 **Implementation** **19**
- 3.1 Unity's Architecture 19
- 3.2 Splines 20
  - 3.2.1 A Class for Functional Operations 20
  - 3.2.2 Common Spline Operations 22
  - 3.2.3 Catmull-Rom Splines 24
  - 3.2.4 Displaying the Spline and Cable Initialiser 27
- 3.3 Unity's Rigidbody Dependant Algorithms 27
  - 3.3.1 Elastic Spring System Model 29
  - 3.3.2 Infinitely Stiff Spring System Model 32
- 3.4 Constraint Solving Models 34
  - 3.4.1 Jakobsen's Advanced Character Physics 34

3.4.2 (Extended) Position Based Dynamics - (X)PBD 40
3.4.3 Comparision and Stiffness Effects 45

4 Efficiency **47**
4.1 Time Complexity 47
4.2 Inter-Collision 48
4.3 Ideas for Parallelization 49

5 Conclusion **51**
5.1 Summary 51
5.2 Further Work 51

Acknowledgements **53**

Bibliography **55**

List of Figures **57**

Introduction

1

In recent years, GPU capabilities grew to a level that is able to render 3D graphics on a low-power or embedded device like a smartphone. Therefore, nowadays a user can run virtual reality simulations on such devices. With that, virtual reality became a popular medium for education - even in academic branches [7, 24, 17, 14]. Creation of real-time 3D graphics with physics for games or simulations can be made without (in-depth) knowledge of computer graphics with the help of game engines free to use for personal, creative or educational purposes [19, 4]. One of those engines is *Unity*.

1.1 Motivation

VRE, which stands for “Virtual-Reality-Experimente” (translated: virtual-reality-experiments), aims to educate students about common physical experiments by simulating them in a virtual reality environment. It was created in the *Unity* engine [9]. Probably one of the most common components in a lab for physical experiments, is the cable. Until the creation of this, physics of cables in *VRE* fall short in terms of realism (although the functionality is fine). Since *VRE* aims for a more realistic experience, we will try to keep up with that goal by introducing realistically looking cable physics.

1.2 Goal

Our goal will be to conceptualize cable physics, implement them while keeping a balance between readability of code, connection to the mathematical concepts and efficiency. The model should create an illusion of realism. However, we will allow artificially set boundaries that contradict the laws of physics to increase usability and prevent unexpected or potentially unwanted behaviour. The developer should be able to define a starting position of the cable and adjust properties of the simulation according to her or his needs.

Theory

2

2.1 Curves & Splines

We would like to give the user the possibility to set a starting position for the cable and allow any kind of folding and any kind of length and thickness. For that reason, we will make a quick recap of the most relevant properties.

2.1.1 Curves

One way to look at curves is as continuous functions from an one-dimensional space to an n-dimensional space - in our case to a three-dimensional space [12, p.359].

Curves can be represented implicitly, parametrically, generatively or procedurally [12, p.359]. We will focus on curves which can be represented parametrically, since they are usually the norm in computer graphics [12, p.361].

To change its form, a curve has a number of points that control its flow. We will refer to those points as *control points*.

2.1.2 Splines

We will define splines as a series of curves. The points where two curves are connected in the spline will be called *glue points*. Control points include glue points. Usually, each of the curves are the same type.

For ease of use, we will borrow the definition of curves from Marschner et al. (see [12, p.359]) and define splines as the function $s(t)$, where t is a rational number. Now we can write the function as $\mathbf{s} : \mathbb{R} \to \mathbb{R}^3$.

In most cases, splines are kept continuous to a degree of two or three to look smooth. That means each curve piece should satisfy (at least the first two of) the following conditions at the glue points:

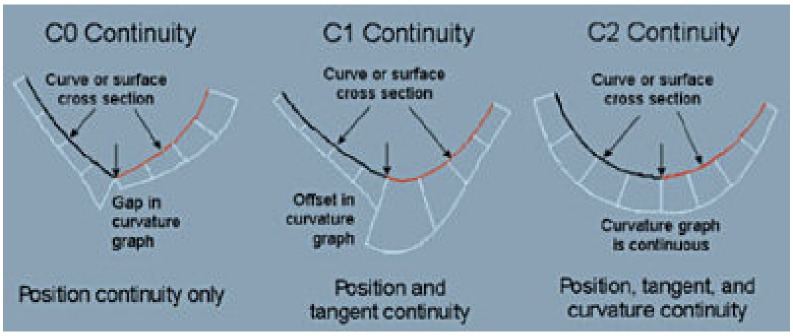
C0 Continuity
Curve or surface cross section
Gap in curvature graph
Position continuity only
C1 Continuity
Curve or surface cross section
Offset in curvature graph
Position and tangent continuity
C2 Continuity
Curve or surface cross section
Curvature graph is continuous
Position, tangent, and curvature continuity

Fig. 2.1: A sketch of how the degree of continuity affects the curvature [23].

$$C_i(x_{i+1}) = C_{i+1}(x_{i+1}) \tag{2.1a}$$

$$C_i'(x_{i+1}) = C_{i+1}'(x_{i+1}) \tag{2.1b}$$

$$C_i''(x_{i+1}) = C_{i+1}''(x_{i+1}) \tag{2.1c}$$

C_i is the i-th curve piece, defined over the interval $[x_i; x_{i+1}]$ [10, p.332]. In Figure 2.1, we can see how the degree of continuity affects the spline.

Please note that if we ensure a continuity in the n-th derivative, then continuity of the previous derivatives is implicitly given, since a derivative only exists if the function is continuous at every point.

For more clarity, we will describe those conditions with an example in physics: Let's assume we have a spline that describes the location of a car on a plane. If the first condition is satisfied, then that would mean that the car would not teleport itself from one location to another. If the second is satisfied, then that would mean the car will not abruptly change its velocity and direction at the glue points of the spline. If the third one is satisfied as well, then the car will keep its acceleration at those points.

Satisfying the first condition is trivial since we only have to set a curve's end point to the succeeding one's start point, thus creating the glue point. The second and third one on the other hand require a little work.

2.1.3 Rendering Function

Looking ahead at the implementation, we also need a function that displays our spline on a visual output, like a screen or VR glasses. We will call this function *rendering function*. Its implementation will be discussed in subsection 3.2.4.

2.1.4 Normalization

Cables are not infinitely long. So, it might be a good idea to define an interval where **s** is defined. We will define a start t_{start} and an end t_{end}, such that $\mathbf{s} : [t_{start}; t_{start}] \rightarrow \mathbb{R}^3$. But once again, looking ahead at the implementation, we would either always have to pass t_{start} and t_{end} to our rendering function or the implementation of **s** had to return some special value (like a null vector, a null pointer or an exception) if $t \notin [t_{start}; t_{start}]$. Therefore, we would like our spline **s** to return all of its points for $t \in [0; 1]$ [12, p.362]. We will call this *normalization* of the spline.

Normalizing a spline is trivial: We define $\mathbf{s}_{norm}$ as the normalized version of **s**, which holds: $\mathbf{s}_{norm}(t) = \mathbf{s}(n(t))$, where $n(t) = t_{start} + t * t_{end}$, such that $\mathbf{s}_{norm} : [0; 1] \rightarrow \mathbb{R}^3$. From this point, when we talk about **s**, we are specifically referring to $\mathbf{s}_{norm}$ unless specified otherwise.

2.1.5 Natural Cubic Splines

With given control points and the conditions mentioned above, we can deduce a basis matrix from which we can deduce a function consisting of cubic curves which wander through all control points and satisfy every condition from Equation 2.1. However, depending on the number of control points, the basis matrix can easily get very large. Looking ahead at the implementation, we might require an efficient algorithm to calculate matrix-vector multiplication, which would cost us more time to implement.

Additionally, one property of natural cubic lines is that whenever a control point's position is changed, the whole spline changes its shape (not only the changed control points immediate neighbouring region) [10, p.332]. The user will likely prefer to change the spline without affecting the look of the whole spline. Therefore, we will skip the mathematical details of its construction and eliminate it as our curve of choice to represent the cable's initial state.

2.1.6 Bézier Curves & Splines

Bézier curves of polynomial degree n are represented by $n+1$ control points, where all the points between the glue points define the curve's tangents. We will call these points *tangent points*. That means a Bézier curve of degree n has $n-1$ tangents [12, p.386]. In our case, picking degree 3 for the curve pieces will be sufficient, since cubic curves can already have extremas and inflection points. Lower degrees either miss at least one the two and higher degrees might provide more of each, but that can be substituted by just connecting together more cubic Cubic Bézier curves.

They can be calculated using the following formula:

$$\mathbf{B}(t) = \sum_{k=0}^{n} B_{n,k}\mathbf{P}_k \tag{2.2}$$

$B_{n,k}$ is the Bernstein polynomial, $\mathbf{P}_k$ the k-th control point and n is the polynomial degree (in our case 3).

According to our definition in subsection 2.1.2, a Bézier spline is just a spline consisting of Bézier curves.

To satisfy the first two conditions of continuity for the spline, we can just set the tangent points neighbouring each glue point to be on one line and have the same distance from the glue point. We will skip the third condition because even if it is not satisfied, the spline will look smooth. But even with that restriction, the tangent points can be set on a lot of locations, which might be unnecessary room of control for the user. We will solve that in the next section.

When changing a control point's position, it will only affect its immediate neighbours, which is at least one of the things we want for the user experience.

2.1.7 (Extended Cubic) Catmull-Rom Splines

Let's assume we had n control points

$$\mathbf{P} = \{\mathbf{p}_1, ..., \mathbf{p}_n\}.$$

Between each pair of points, we add two additional tangent points

$$\mathbf{T} = \{\mathbf{t}_{1,1}, \mathbf{t}_{1,2}, \mathbf{t}_{2,1}, \mathbf{t}_{2,2}, ..., \mathbf{t}_{n,1}, \mathbf{t}_{n,2}\}.$$

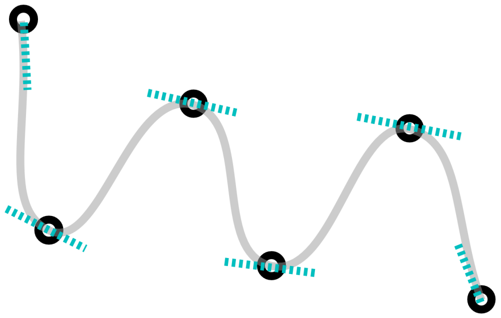

Fig. 2.2: A sketch of an Extended Cubic Catmull-Rom Spline. The dashed lines represent the tangents.

If we made a Bézier curve out of $\mathbf{p}_i, \mathbf{t}_{i,1}, \mathbf{t}_{i,2}, \mathbf{p}_{i+1}$ for every $i \in [1; n-1]$ and set the preceding and succeeding tangent points around a $\mathbf{p}_i$ (namely $\mathbf{t}_{(i-1),2}, \mathbf{t}_{i,1}$) in such a way that $\frac{1}{2}\mathbf{p}_{i+1} - \mathbf{p}_{i-1}$ is their direction each (positive for $\mathbf{t}_{i,1}$, negative for $\mathbf{t}_{(i-1),2}$) and $\frac{1}{2}|\mathbf{p}_{i+1} - \mathbf{p}_{i-1}|$ is their magnitude, we get so called *Catmull-Rom Splines*. In other words, the tangents of each $\mathbf{p}_i$ are parallel to the line connecting $\mathbf{p}_{i-1}$ and $\mathbf{p}_{i+1}$ and have a summed up magnitude of $\frac{1}{2}|\mathbf{p}_{i-1} - \mathbf{p}_{i+1}|$ [10, p.330-331].

Since the end points $\mathbf{p}_1$ and $\mathbf{p}_n$ are missing either a succeeding or preceding point, we can just set their tangents to a quarter (not the half) of the vector connected from the point itself and its only neighboring point, specifically

$$\frac{1}{4}(\mathbf{p}_2 - \mathbf{p}_1) \text{ for } \mathbf{p}_1 \text{ and}$$

$$\frac{1}{4}(\mathbf{p}_{n-1} - \mathbf{p}_n) \text{ for } \mathbf{p}_n$$

(notice the swapped order for $\mathbf{p}_n$ make the tangent face “inwards”). With that extension, we will just call them *Extended* Catmull-Rom Splines (see Figure 2.2).

Since the start and end points of each curve piece in the Catmull-Spline share the same location and tangent, the first and second condition mentioned in Equation 2.1 are already satisfied [10, p.334]. However, the third one may not, but we do not care about that, since the spline will still have a curvy shape. Also, when we change a control point’s position, it will not affect the whole curve and the user does not have to manually set the tangents.

For those reasons, Catmull-Rom splines will be our curve of choice to represent the cable’s initial state.

2.2 Physics, Simulation and Algorithms

We need to find a way to mathematically represent the cable's state in space, how it reacts to external forces and how to calculate the new position and state of the cable. Since there are multiple ways to do this, we will take a look at some simulation models and its corresponding algorithms and pick the one which is easiest and therefore probably quickest to implement while remaining accurate enough to provide the illusion of a real cable. With "easiest and quickest to implement", we do not only mean the simulation model and algorithm itself, but also how effectively we can integrate it into the Unity system by using as many functions that are already provided as possible. This will not only save time on implementation, but also improve performance, since Unity has hardware-optimized functions [22]. We will go deeper into that in the implementation section later.

2.2.1 Physics, Force & Mass in Game Engines

We will take a brief look at how physics in game engines work to get a better understanding of how the algorithms and the model of our cable work.

Inspired by Newtonian Physics, a physical simulation consists of a number of rigid bodies or particles, each having a mass on which forces act upon. If the force on one of those bodies is constant, then, with the help of the directional variant of Newton's second law $\mathbf{F} = m\mathbf{a}$, we can derive the velocity and the location of that body algebraically (i.e. by transforming the acceleration equation and integrating) [16, p.53].

Usually, that constant force is derived from the gravitational acceleration constant $g = 9.81m/s^2$, or more specifically

$$\mathbf{g} = g * \mathbf{d}, \text{ where } \mathbf{d} = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}, \tag{2.3}$$

which we will just call *gravity vector* for simplicity's sake. As an example, let's imagine we had a game controller with a mass of $2.3kg$. Then, its force will be

$$\mathbf{F} = m\mathbf{a} = 2.3kg * 9.81m/s^2 * \mathbf{d} = 22.563kg * m/s^2 * \mathbf{d} = 22.563N * \mathbf{d}.$$

One should note that, independent of its mass, the acceleration of an object is always the same

$$\mathbf{F}/m = \mathbf{a} = \mathbf{g}.$$

Therefore, so is its velocity function

$$\mathbf{v}(t) = \int \mathbf{a}\, dt = \int \mathbf{g}\, dt = \mathbf{a}t + \mathbf{v}_0 = \mathbf{g}t + \mathbf{v}_0,$$

and its location function

$$\mathbf{r}(t) = \int \mathbf{v}\, dt = \int \mathbf{g}t + \mathbf{v}_0\, dt = \frac{1}{2}\mathbf{g}t^2 + \mathbf{v}_0 t + \mathbf{r}_0 \text{ [16, p.52].}$$

Physical properties, like the velocity, position and mass (depending on the engine), are stored in an object with rigid body effects (for simplicity's sake from now on just referred to as *rigid bodies*). Other properties, like force and acceleration, are then calculated from those parameters whenever needed [8].

In a game engine, we update every body's location at a fixed time interval Δt independent of the (variable) frame rate at which the scene is being rendered [10, p.459-460]. Rendering means, with the positions of every object given, we compute the picture to display.

In most cases, many different forces from various sources will act on a body in a non-constant fashion. For example, when a collision with another body occurs or when the user interacts with the cable, new forces which cannot be foreseen are introduced to the object [16, p.54]. But for now, we will assume that the Unity Engine will handle this problem for us. We will discuss the problem of collision, interaction of the user and variable external forces in general in a later section.

2.2.2 Cable Model

In the previous section, we examined rigid bodies. However, a cable is not rigid. Therefore, we will now look at *soft bodies* and their simulation. Clothing, rubbery objects and cables are simulated soft bodies. Soft bodies can be simulated as constrained particles [8, 3, 11, 15, 10].

We will think of our cable as a bunch of equidistant particles, which wander through the core of our cable. That would mean, if we drew a line along those points, we would get the silhouette of the cable. Each particle will have a radius, which will be the radius of the cable it models.

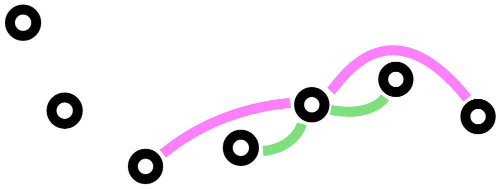

Fig. 2.3: A sketch of the cable model. On one particle, the distance constraints have been visualized. The green lines represent the single, the magenta lines the double distance constraints.

After every time step, forces will have acted on our particles (for example gravity). With every algorithm presented here, our goal will be to make sure that after each of those forces have acted on them, the particles still satisfy a number of constraints. That will create the illusion of a cable. Those constraints will be:

- Keep the same distance between the particles as initially set. That will make sure the cable keeps the same length throughout the simulation [10, p.458].
- Keep the same distance between every pair of particles where one particle is in between. This might seem odd at first, but that will make sure that our cable will not fold through itself [10, p.458].
- Do not penetrate other objects. We do not want the cable to fall through other objects.

See Figure 2.3 for a visualisation.

2.2.3 Algorithms Dependant on Unity's Ecosystem

First, we will explore the simple approaches we can take to achieve the desired cable effect. The idea behind the following algorithms is to use only internal functions and classes of Unity representing physical objects and their handling as much as possible.

Elastic Spring System Model

Since the elastic spring system model can be used for clothing simulation [10, p.457], it can be analogously adapted for the simulation of cables. The clothing mesh is divided in equidistant particles, like on a grid. Each particle is connected to its nearest horizontal and vertical neighbor and constrained to its distance to them. Additionally, it is connected to its second horizontal and vertical neighbor to prevent the cloth from folding through itself. Lastly, the distance between its immediate diagonal neighbors are kept to prevent shearing [10, p.458]. The connections would then act according to the physics of the spring:

$$\mathbf{F}_{spring} = k_{spring}(\|\mathbf{Q} - \mathbf{P}\| - d)\frac{\mathbf{Q} - \mathbf{P}}{\|\mathbf{Q} - \mathbf{P}\|}, \tag{2.4}$$

where **P** and **Q** are two connected points, $\mathbf{F}_{spring}$ is the force experienced by **P** ($-\mathbf{F}_{spring}$ by **Q**), d the resting distance (which is different depending on if **P** and **Q** are connected in double distance, diagonally or immediately) and k_{spring} the stiffness factor [10, p.459].

For our cable, we would implement the simulation the same way as the clothing simulation, leaving out the connections perpendicular and diagonal to the particles, since those do not exist.

Because Unity already provides a system to simulate forces on objects [22], the implementation of this algorithm should be very quick when utilised. We can represent the particles as spheres with predefined masses with Unity's Rigidbody Component, allowing us to use the built-in collision detection and resolving system [22, 21].

One should note that in this model, the constraints mentioned in subsection 2.2.2 will not necessary be satisfied after each time step. Although the spring tries to return to its resting distance, the distance constraints will not be enforced.

And that is the exact problem. The cable will behave more similarly to a rubber string than to a cable, since a cable's springing behaviour is minimal. Because the physical law of springs includes a k_{spring} constant, which defines the stiffness of the spring behaviour, we could counteract the effect by setting k_{spring} very high. Unfortunately, that tends to introduce unwanted vibrations or other problems due to numerical instability [10, p.459].

But to make sure we are not missing out on an opportunity of receiving acceptable results with the least amount of work, we will still implement this system and analyse the result.

Infinitely Stiff Spring System Model

To counteract the rubbery-like behaviour we fear in section 2.2.3, we can instead assume an infinitely stiff spring between each particle, which would mean that at every moment of time, the distance between two particles is the resting distance. Instead of applying forces to each particle, we will move it into position in every fixed time step.

More specifically, if we had a particle position $\mathbf{x}_1$ and a particle position $\mathbf{x}_2$, then we would equally move them closer together or further apart until the distance constraint is satisfied again. That means to get the correct distance one half of the distance would be applied to $\mathbf{x}_1$ and the other half to $\mathbf{x}_2$. This method is called local solving [5]. The same would be done for the double distance. While this does not ensure that the distance constraints are globally satisfied, it would still be close enough, even if we iterate only once over the particles [8].

Usually, Unity will restrict movement of objects with an attached Rigidbody Component from penetrating other objects. That means, if one particle collided with a non-moving object (like a table top) and its neighbouring particle did not, then when applying the corrective movement to both of them, the colliding particle will likely not move its required half compared to the non-colliding neighbour. That means, we cannot be sure that the distance constraints will be satisfied after each time step, not even approximately.

Although this also might seem to be an idea that is unlikely to work, we will implement it and check the results.

2.2.4 Physics, Variable Forces and Collisions in Game Engines

In subsection 2.2.1, we skipped how variable forces are handled by an engine. To gain a better understanding of the next algorithms, we will look at the problem with variable forces and how it relates to collision and user interaction.

Non-constant Forces

Usually, an object might get in a rigid body's way or the user might apply some sort of force through interaction. In other words, we try to answer the question the question what the location of an object will be when it collides against something or when the user drags it away, for example. To answer this question, we will expand

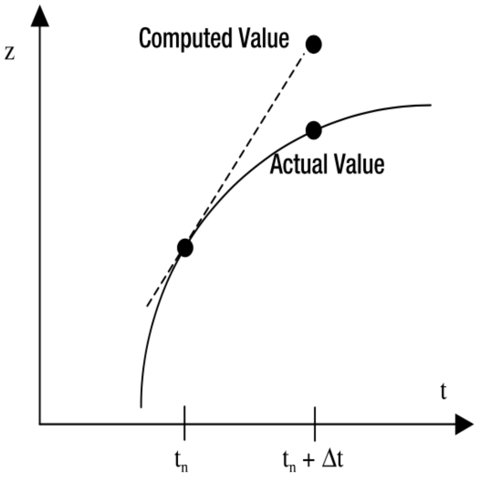

Fig. 2.4: Euler's method - If the location, in this figure the z-axis, changes non-linearly, the approximation is inaccurate [16, p.57, fig.4-4].

our function for the velocity $\mathbf{v}$ with an additional parameter $\mathbf{r}$, which will be the object's current location. This is the method used by Palmer [16, p.56]. Thus an object's velocity $\mathbf{v}(t)$ becomes $\mathbf{v}(\mathbf{r}, t)$. Since the velocity now also depends on the current location, the function $\mathbf{v}(\mathbf{r}, t)$ can give us a different value, if for example $\mathbf{r}$ is on the surface of or inside another object. That would mean a collision occured.

From subsection 2.2.1, we know that the velocity is the derivative of the location:

$$\frac{d\mathbf{r}}{dt} = \mathbf{v}(\mathbf{r}, t) \tag{2.5}$$

Our time steps Δt for the simulation are not (and cannot be) infinitesimally small. However, if we choose Δt small enough ($\Delta t \neq 0$), we can approximate the change of the object's location $\Delta\mathbf{r}$:

$$\frac{d\mathbf{r}}{dt} \approx \frac{\Delta\mathbf{r}}{\Delta t} = \mathbf{v}(\mathbf{r}, t) \iff \Delta\mathbf{r} = \mathbf{v}(\mathbf{r}, t)\Delta t \tag{2.6}$$

We could substitute $\mathbf{r}$, t and Δt to calculate $\Delta\mathbf{r}$. That method is known as Euler's method, but it is inaccurate for non-linear location functions according to Palmer (see Figure 2.4) [16, p.57]. However, since the approximation of the location does not necessarily need to depend on the velocity, there are other approximations for $\Delta\mathbf{r}$. It could also depend on the acceleration [8] (with an added parameter for the current location). We will see another approach in subsection 2.2.5.

2.2.5 Constraint Solving Algorithms

Implementing the following algorithms requires more work, since we will have to solve some problems like collision handling ourselves. However, the trade-off will be that the following algorithms are probably more likely to provide usable results.

Jakobsen's Advanced Character Physics

In November 2000, IO Interactive released the game *Hitman: Codename 47*, which used a simplified approach to achieve realistically looking physics [8, 6]. It was used for plants, for solid objects (like boxes) and for ragdoll physics for dead bodies [8]. The system was invented by Jakobsen. Jakobsen's primary goal of the model was to achieve "believability" [8]. Due to its broad spectrum of applicability to common challenges of physical simulation in games, including soft and rigid body simulation, it suits our needs well.

The core of the model is the Verlet integration scheme (written in pseudo-code):

$$\mathbf{x'} \leftarrow 2\mathbf{x} - \mathbf{x^*} + \mathbf{a} * \Delta t^2 \tag{2.7a}$$

$$\mathbf{x^*} \leftarrow \mathbf{x}, \tag{2.7b}$$

where $\mathbf{x}$ is the current, $\mathbf{x^*}$ the last and $\mathbf{x'}$ the approximated position of the particle after fixed time step Δt (as if there were no constraints at all). One should note that the algorithm does not include masses in its calculation [8].

For each particle, we store its last position and its current position in an array each. For the acceleration $\mathbf{a}$, we just use the gravity vector from Equation 2.3.

After the new positions have been calculated, we apply a row of functions, which we will call *solver functions*, each iterating over all neighbouring tuples of points. The solver functions ensure that our constraints from subsection 2.2.2 are satisfied. Since the solver functions only verify the neighbouring tuples of particles independently of other tuples, they make up a local solving system [5]. We will use the same method we used in the infinitely stiff cable model for the distance constraint because it is the same method used by Jakobsen [8]:

$$\Delta \mathbf{p}_1 = -\frac{1}{2}(|\mathbf{p}_1 - \mathbf{p}_2| - d)\frac{\mathbf{p}_1 - \mathbf{p}_2}{|\mathbf{p}_1 - \mathbf{p}_2|} \tag{2.8a}$$

$$\Delta \mathbf{p}_2 = \frac{1}{2}(|\mathbf{p}_1 - \mathbf{p}_2| - d)\frac{\mathbf{p}_1 - \mathbf{p}_2}{|\mathbf{p}_1 - \mathbf{p}_2|} \tag{2.8b}$$

It should be noted that the paper also proposes an approach that respects different weights [8]. However, we do not need it, since we assume our cable has the same density everywhere. Therefore, the particle's weights are all the same.

To prevent penetration of other objects, we will rely on a given collision system, as it is suggested by Jakobsen [8]. We can do that since Unity provides a class to resolve collisions [21]. It will be included as a solver function.

Since we satisfy the constraints only locally, another (or likely even the same) constraint might not be satisfied on another tuple anymore. Therefore, we iterate multiple times over all tuples achieving (good enough) satisfaction of all constraints for all tuples due to convergence [8]. That means the more iterations, the more strictly the constraints are satisfied. The more strictly constraints are satisfied, the stiffer the cable will appear to be. We will call this effect the *accuracy-stiffness effect*.

By decreasing the time step Δt for physics calculations, cable stiffness will be increased. This happens because more physical calculations are made between each time a new frame is rendered. We will call this phenomenon *timescale-stiffness effect*.

Since the primary goal of this method is to achieve "believability", it is implied that the simulation might not be physically accurate (especially keeping in mind that mass is not integrated into the calculation). However, Müller et al. were able to prove that at least the proposed solver to satisfy the distance constraint does indeed respect linear and angular momentum [15]. We will see more on that later in section 2.2.5.

Additionally, the accuracy-stiffness and the timescale-stiffness effect might be something undesirable for user and developer, since with an increasing number of solving iterations or a smaller time step, a more accurate simulation is expected, not a stiffer one.

However, due to its simplicity and effectiveness in *Hitman: Codename 47*, we will implement this method and inspect the results afterwards.

(Extended) Position Based Dynamics - (X)PBD

Just like Jakobsen's Model in section 2.2.5, we set up a bunch of constraints that need to be satisfied for each neighbouring tuple. But instead of writing a solver for each, we would either derive one through mathematical transformations or approximate one. This is the approach taken in *Position Based Dynamics* (PBD) [15] and its extended variant *Extended Position Based Dynamics* (XPBD). Both approaches base on a variant of Quasi-Newton approximation. That means, we can only define the constraints themselves, not how they are satisfied. Just like Jakobsen's method, the algorithm starts by first estimating the particle positions as if there were no constraints at all, then corrects their positions. But instead of Verlet integration, PBD and XPBD estimate the new positions with Euler's method.

The primary aim of this algorithm is achieving approximately realistic results at an inexpensive cost [11]. It is based on Position Based Dynamics (PBD), which was created for games primarily [15]. Also, like Jakobsen's method, XPBD and PBD converge to a solution for our particle positions, satisfying the cable constraints by iterating over the positions multiple times [11].

To get a better grasp of the system, we will explain it along with an example: The system requires our constraints to be formulated in such a way that they either have to be zero or larger than zero. Our distance constraint for example would be formulated as:

$$C_{distance} = |\mathbf{p}_1 - \mathbf{p}_2| - d, \tag{2.9}$$

where d is the resting distance. Now, if $C_{distance}$ equals zero, that means the distance between $\mathbf{p}_1$ and $\mathbf{p}_2$ is exactly the resting distance d, which means the constraint would be satisfied.

But if the constraint was not satisfied, how would we have to change $\mathbf{p}_1$ and $\mathbf{p}_2$ in order to do so while also preserving linear and angular momentum (assuming equal weights for all particles)? Müller et al. propose the following approximative equation for all points $\mathbf{p} = [\mathbf{p}_1^T, ..., \mathbf{p}_n^T]$ required for any constraint C [15]:

$$C(\mathbf{p} + \Delta\mathbf{p}) \approx C(\mathbf{p}) + \nabla_{\mathbf{p}} C(\mathbf{p}) * \Delta\mathbf{p} = 0, \tag{2.10}$$

where $\Delta\mathbf{p}$ is the change we apply to the particles (that C currently checks) in order to satisfy the constraint, respecting linear and angular momentum. So for $C_{distance}$, $\mathbf{p}$ would be equal to $[\mathbf{p}_1^T, \mathbf{p}_2^T]$, since $C_{distance}$ requires two points. Through a series of substitutions and transformations of equations, the following corrective value $\Delta\mathbf{p}_i$ for a particle $\mathbf{p}_i$ is calculated:

$$\Delta\mathbf{p}_i = -s\nabla_{\mathbf{p}_i}C(\mathbf{p}_1, ..., \mathbf{p}_n) \tag{2.11}$$

$-\nabla_{\mathbf{p}_i}C(\mathbf{p}_1, ..., \mathbf{p}_n)$ gives us the direction of change, respecting both types of momentum, and s gives us the magnitude, which is (disregarding different weights of particles):

$$s = \frac{C(\mathbf{p}_1, ..., \mathbf{p}_n)}{\Sigma j|\nabla_{\mathbf{p}_j}C(\mathbf{p}_1, ..., \mathbf{p}_n)|^2} \tag{2.12}$$

s is calculated for each constraint.

So for our example $C_{distance}$, $\Delta\mathbf{p}$ would be

$$\Delta\mathbf{p}_1 = -\frac{1}{2}(|\mathbf{p}_1 - \mathbf{p}_2| - d)\frac{\mathbf{p}_1 - \mathbf{p}_2}{|\mathbf{p}_1 - \mathbf{p}_2|},$$

$$\Delta\mathbf{p}_2 = \frac{1}{2}(|\mathbf{p}_1 - \mathbf{p}_2| - d)\frac{\mathbf{p}_1 - \mathbf{p}_2}{|\mathbf{p}_1 - \mathbf{p}_2|}$$

for $\mathbf{p}_1$ and $\mathbf{p}_1$ respectively, which match Jakobsen's suggestion for keeping the distance constraint satisfied (see Equation 2.8). This proves that linear and angular momentum are respected Jakobsen's suggestion for the distance constraint.

While s and $\Delta\mathbf{p}_i$ can be calculated algebraically, it can also be calculated numerically. This might be interesting for cases where the gradient function is unknown.

Finally, if C is an inequality constraint, then the deltas will only be applied if $C(\mathbf{p}_1, ..., \mathbf{p}_n) < 0$.

The difference between PBD and XPBD is that PBD also suffers from the accuracy- and timescale-stiffness effect mentioned in section 2.2.5, while XPBD does not due to the introduction of a Lagrange multiplier for the constraint functions [11]. Müller et al. are aware of the effect and even propose multiple solutions [15], but they do not counteract the timescale-stiffness effect and make convergence worse [11]. Macklin et al. [11] instead provide another solution, by finding the equation for $\Delta\mathbf{p}$ from a different approach and deriving the following equation for the corrective magnitude s:

$$s = \frac{C(\mathbf{p}_1, ..., \mathbf{p}_n) - \tilde{\alpha}\lambda_j}{\Sigma j|\nabla_{\mathbf{p}_i}C(\mathbf{p}_1, ..., \mathbf{p}_n)|^2 + \tilde{\alpha}} \tag{2.13}$$

Notice the new $\tilde{\alpha}$ and λ_j in the numerator and the denominator. j is the index of the iteration (Remember: We iterate multiple times over all point tuples to converge to the correct solution). $\tilde{\alpha}$ is equal to $\frac{\alpha}{\Delta t^2}$, where α is the inverse of the stiffness for the current constraint. So when the user sets the stiffness $\frac{1}{\alpha}$, then $\tilde{\alpha}$ will calculate the stiffness independent of the time step Δt. The inability for α to be zero makes also sense, since that would mean the constraint is so elastic it is not enforced at all. λ_j is the Lagrange multiplier for the j-th iteration of the constraint and is updated with each iteration over all points. It starts with $\lambda_0 = 0$ for each constraint and is updated with each iteration:

$$\lambda_{j+1} = \lambda_j + s \tag{2.14}$$

So with every iteration, s is modified with λ_j to counter the accuracy-stiffness effect. In each constraint every tuple has its own lambda value.

While PBD and especially XPBD seem to be very powerful, the disadvantage would be that both of them are more complicated than the Jakobsen method. However, they share the same idea: Estimate positions, then correct them. So if the Jakobsen model works, we can build the XPBD and the PBD model on top of it. The accuracy- and the timescale-stiffness effect are still present in PBD, however counteracted in XPBD (if $\tilde{\alpha} > 0$). Additionally, the estimation of the new positions is calculated using Euler's method, which has a larger room for error if the particle changes its velocity over time [16]. But we can easily fix it by using Verlet integration for the estimation as well. Also, we will have to write a function (or a class) that can construct gradient functions for the constraints.

Although our primary idea of this system is to automatically generate a solver function, we will provide an option to mount one manually. That way, we can ensure numerical stability for specific constraints if desired.

2.2.6 Other Systems and Algorithms

There are other systems and algorithms that could help achieve realistically looking cable simulation like the Featherstone algorithm (see [13]) or the particle-system model for clothing by Eberhardt et al., which even respects air resistance (see [3]). Unfortunately, the workload would be too high to summarize, discuss and implement all of them. However, due to their powerful nature, they at least deserve to be mentioned.

3

Implementation

Before we begin, we will briefly examine the architecture of the Unity Engine and its most important classes. Then, we will go through how we will implement our splines. Finally, we will get to the most important part and explain the implementation of the cable physics, comparing each algorithm.

The full source code for the implementation is available at `https://gitlab.rlp.net/mamirago/bachelor-thesis`.

3.1 Unity's Architecture

Unity is a 3D- and 2D-capable software suite, which includes a game engine for making games and simulations. It comes with an editor for building and creating 3D or 2D scenes. It is an *Entity-Component* based architecture. A number of *Game Objects* are placed in environments called *Scenes*. They represent the entities in the architecture, on which various *Components* are attached. Depending on the Components attached, each Game Object gets its own purpose or property in the game. With Components, a Game Object becomes a playable character, an enemy, an obstacle, a landscape, a camera, a light, etc. Game Objects can (and in most cases have) multiple components. Components have *Properties*, which customize a Components behaviour according to the developers needs. Every Game Object has a *Transform* Component, which is used to define a Game Objects scaling, rotation and translation in the world. One can think of it as a Game Object's own coordinate system. A Game Object can have a *Parent* Game Object, which makes it a *Child* Game Object. By being a Child, the Game Object's Transform component (i.e. its coordinate system) becomes dependent on the Parent's. Among the built-in components, a developer can create his own ones with its own Properties, which are then called *Scripts*. We will also use Scripts and attach them to a Game Object, so it becomes a simulated cable. We will create a Script that describes how the cable will look when the simulation begins, how it behaves with gravity, how it reacts to collisions, etc.

In the Scripts, we add our own code to define those behaviours. The programming language for Scripts is *C#*. Unity provides a number of classes to access other components (like the Rigidbody component) to interact with other Game Objects. Additionally, classes to perform algebraic calculations for 3D and 2D like matrices and vectors are provided. Scripts are required to inherit from the *MonoBehaviour* class to be recognized as Components. Since Scripts inherit from MonoBehaviour, they are classes themselves [22].

A built-in class, that we are going to use a lot, is the *Vector3* class, which represents a vector in 3D space and provides common vector operations, like the dot product, cross product, multiplications with scalars, get a vector's magnitude etc. Other classes will be explained on the fly.

This section is kept very short, leaving out details on purpose to prevent redundancy. For a more in-depth look, see [22].

3.2 Splines

For our spline, we will take a functional programming approach, as this allows us to prepare a function representing a spline and delay its execution as late as possible. That way, we won't run into problems caused by aliasing. Luckily, C# provides built-in functions to create functions dynamically in run-time called λ-*functions*. First, we will create a class to provide us with some of the common functional operations (more specifically the *map* function, with and without index), then a class for spline operations like normalizing, aliasing and gluing (see section 2.1). After that, we create another class to create a function representing a user defined Catmull-Rom Spline (see subsection 2.1.7). Then finally, we create a class to make the spline visible in the Unity Editor.

Definitions for helper functions and common checks for abnormalities were omitted on purpose since they either do not contribute to the comprehension, are just not important enough or can be easily explained on the fly.

3.2.1 A Class for Functional Operations

We will wrap the class for functional operations in the namespace `Functional` (see Figure 3.1). This class class will help us define functions that return functions or expect them as parameters. `R` in `Returns<R>` is passed as a generic type to the delegates, so we can define what the function should return. With `Expects` we can define what type of parameters our function should expect. We overload it with a

```
namespace Functional
{
    public class Returns<R>
    {
        public delegate R Expects();
        public delegate R Expects<T1>(T1 t);
        public delegate R Expects<T1, T2>(T1 t1, T2 t2);

        //Map over Array with index provided
        public static R[] Map<T>(T[] ts, Expects<T, int> map)
        {
            R[] rs = new R[ts.Length];
            for (int i = 0; i < ts.Length; i++)
            {
                rs[i] = map(ts[i], i);
            }

            return rs;
        }

        //Map over Array without index provided
        public static R[] Map<T>(T[] ts, Expects<T> map)
        {
            Expects<T, int> map_ = (e, i) =>
            {
                return map(e);
            };
            return Map<T>(ts, map_);
        }
    }
}
```

Fig. 3.1: The `Functional` namespace provides functions to include typesafe functional programming paradigms.

type that expects no, one or two arguments, where each type can be set by a generic type.

This seems confusing at first, but the following examples will clarify the purpose of this class. For example, if we defined a function `MyFunction` that returned a function which expects one `int` parameter and returns a `Vector3`, we can define it the way shown in Figure 3.2.

It also enables us to pass functions as parameters (known as *higher order* functions). For example, we want a function that just executes a function, which expects an `int` and a `string` parameter and returns an `bool` (see Figure 3.3).

We do not need to know how and what exactly `f` does, we just make sure it receives and returns the correct types.

```
public Returns<Vector3>.Expects<int> MyFunction(){
    return (myInt) => {
        return new Vector3(myInt, myInt, myInt);
    }
}
```

Fig. 3.2: Example: `MyFunction()` - Returns a function that expects an `int` and returns a `Vector3`, where each component of it is the given `int`.

```
public float ExecuteAndReturnResult(Returns<bool>.Expects<int,
    string> f){
    return f(12,"Hello Unity!") ? 3.1415f : 1.4142f;
}
```

Fig. 3.3: Example: `ExecuteAndReturnResult()` - A demonstration of how the signature of parameter `f` can now be defined.

Finally, we can use this class to iterate over arrays. An example would be a function that receives an array of `int`s and returns an array where every element has been powered by `2.3f`. Usually, the change to the array is applied by reference, but for later convenience, we instead create a new array (see Figure 3.4).

Please note that the generic type in `Returns` is about the type of each element, not the array. `Mathf` is a class by the Unity Engine for common mathematical operations. We could omit `float` in `Map<float>`, but we decided to keep it there for better comprehension.

Of course, there are other common functional operations like `fold`, but we won't be needing them, so we skip their implementation.

3.2.2 Common Spline Operations

We create another class called `SplineTools` that will help us in creating the Catmull-Rom splines.

```
public float[] Power2Point3(float[] myNumbers){
    return Returns<float>.Map<float>(myNumbers, (element) => {
        return Mathf.Pow(element, 2.3f);
    });
}
```

Fig. 3.4: Example: `Power2Point3()` - A demonstration of how `Map()` can be used to iterate over an array.

```
public static Returns<Vector3>.Expects<float> CreateGluedFunction(
    Returns<Vector3>.Expects<float>[] fs
)
{
    return (float t) =>
    {
        //edge cases first
        //first function
        if (t < 1f)
        {
            return fs[0](t);
        }

        //last function
        var lastIndex = fs.Length - 1;
        var fLength = (float)lastIndex;
        if (t >= fLength)
        {
            return fs[lastIndex](t - fLength);
        }

        //in between function
        var t_ = t % 1f;
        var fIndex = Mathf.FloorToInt(t);

        return fs[fIndex](t_);
    };
}
```

Fig. 3.5: `CreateGluedFunction()` - Turns multiple curves into one spline.

`CreateGluedFunction()` (see Figure 3.5) expects an array of (normalized) spline functions `fs`, which it will turn into one function. The function it returns receives a parameter `t`, for which it finds out which of the glued functions should return the result. It then transforms `t` to `t_`, since internally, the selected glued functions expects `t` to be between `0f` and `1f`.

Please note that the returned function of `CreateGluedFunction()` is not normalized and expands the loosely defined input space for the spline it represents, from `0f` to `fs.Length`. With “loosely”, we mean that while the input space is set, the function still returns a valid value for inputs out of range. Also, it does not ensure continuity (see Equation 2.1).

As the name already implies, `CreateNormalizedFunction()` normalizes a given function by working as a medium between a given `t` and the non-normalized function. It transforms `t` to the definition space of `f`. The code is shown in Figure 3.6.

```
public static Returns<Vector3>.Expects<float> CreateNormalizedFunction(
    float start,
    float end,
    Returns<Vector3>.Expects<float> f
)
{
    return (t) =>
    {
        var t_ = start + t * (end - start);
        return f(t_);
    };
}
```

Fig. 3.6: `CreateNormalizedFunction()` - Normalizes the function while allowing `t < 0f` and `t > 1f`.

3.2.3 Catmull-Rom Splines

Now with the creation of the class `CatmullSpline`, we finally get to the actual creation of the Catmull-Rom spline.

The purpose of `CreateAutoCubicBezierPoints` is to create the in-between tangent points from `controlPoints`. See the code in Figure 3.7. Just like in subsection 2.1.7 described, we create the tangent vector `tangent` from the neighbouring points of a point **p** (here: `controlPoints[i]`), then move from **p** in the directions `tangent` and `-tangent` with a magnitude of `.25f`, which is set in the constant `TANGENT_MAG_FACTOR`. That way, the whole tangent at the glue point gets a magnitude of `.5f`.

For the end points we move in the direction of the next neighbour for the tangent, but only with a magnitude of `.25f` since the tangent only moves in one direction (see Figure 3.8).

Now with `CreateAutoCubicBezierPoints`, we have the points needed to create an array of functions representing Bézier curves and just glue them together so they become a continuous spline. We do that in `CreateCubicBezierSplineFunction` (see Figure 3.9). Since the tangents are exactly the ones of a Catmull-Rom spline, the result will be the full-length Catmull-Rom spline. We use `CreateNormalizedFunction()` to loosely transform the definition space to `[0f; 1f]`.

```
static Vector3[] CreateAutoCubicBezierPoints(
    Vector3[] controlPoints
)
{
    //... assume end points have already been taken care of here

    for (int i = 1; i < lastIndex; ++i)
    {
        var tangent = TANGENT_MAG_FACTOR * (
            controlPoints[i + 1] - controlPoints[i - 1]
        );

        var leftBound = 3 * (i - 1) + endPointsLen;

        r[leftBound] = controlPoints[i] - tangent;
        r[leftBound + 1] = controlPoints[i];
        r[leftBound + 2] = controlPoints[i] + tangent;
    }

    return r;
}
```

Fig. 3.7: `CreateAutoCubicBezierPoints()` - Calculates the tangent points and includes them in the control points, such that they become points of a Catmull-Rom Spline.

```
var len = controlPoints.Length;
var endPointsLen = 2;
var rLen = GetNumberOfCubicBezierPoints(len, endPointsLen);

var r = new Vector3[rLen];

//Set first two and last two points each
var startTan = TANGENT_MAG_FACTOR * (controlPoints[1] -
    controlPoints[0]);
r[0] = controlPoints[0];
r[1] = controlPoints[0] + startTan;

var lastIndex = len - 1;
var lastRIndex = rLen - 1;
var endTan = TANGENT_MAG_FACTOR * (controlPoints[lastIndex] -
    controlPoints[lastIndex - 1]);
r[lastRIndex - 1] = controlPoints[lastIndex] - endTan;
r[lastRIndex] = controlPoints[lastIndex];
```

Fig. 3.8: `CreateAutoCubicBezierPoints()` - Taking care of the end points.

```
static Returns<Vector3>.Expects<float> CreateCubicBezierSplineFunction(
    Vector3[] controlPoints
)
{
    var fArrayLength = (controlPoints.Length - 1) / 3;
    var fArray = new Returns<Vector3>.Expects<float>[fArrayLength];

    for (int i = 0; i < fArrayLength; ++i)
    {
        var ps = Arr<Vector3>.Extract(controlPoints, 3 * i, 4);
        fArray[i] = CreateCubicBezierFunction(ps);
    }

    var end = (float)fArrayLength;

    return SplineTools.CreateNormalizedFunction(
        0f,
        end,
        SplineTools.CreateGluedFunction(
            fArray
        )
    );
}
```

Fig. 3.9: `CreateCubicBezierSplineFunction()` - If the points for the Spline are generated, we create the function representing the spline by first creating multiple Bezier curves, transforming them into one spline function, then normalize the spline.

```
private void OnSceneGUI()
{
  //...

    // Create and alias Catmull-Rom Spline
  var cps = line.controlPoints;
  var aliasPoints = SplineTools.AliasFunctionDynamically(
    CatmullSpline.CreateCatmullSpline(cps),
    cps.Length,
    inbetweenRes
  );

  // Draw line between each point in world space
  Handles.DrawAAPolyLine(
    lineWidth, Returns<Vector3>.Map(
      aliasPoints,
      GetPointPosInWorld
    )
  );
}
```

Fig. 3.10: `Handles.DrawAAPolyLine` in `OnSceneGUI()` allows us to display the spline in the scene.

3.2.4 Displaying the Spline and Cable Initialiser

While we now examined the classes necessary to create the spline function, we additionally need to display them in Unity. The implementation will loosely be based on the tutorial by Catlike Coding (see [2]).

Among other minor steps, we need to inherit from Unity's built-in `Editor` class and define `OnSceneGUI()`, which will be called as an event (see [22]). In there, we first create our Catmull-Rom Spline with `CatmullSpline.CreateCatmullSpline()` and then draw them into the Editor view with `Handles.DrawAAPolyLine()`. See Figure 3.10 for the code. Since the created spline is defined in local space, we need to transform it into global space with `GetPointPosInWorld()` and `Map()`.

Now a developer is able to set the cable's initial position (see Figure 3.11). When the particles are initialised, they are placed along the spline.

3.3 Unity's Rigidbody Dependant Algorithms

In section 3.1, we briefly summarized Unity's architecture. For the algorithms, it might be useful to know a little more about the run-time of the architecture. When the game starts running, the engine (among other operations) calls the `Start()` of

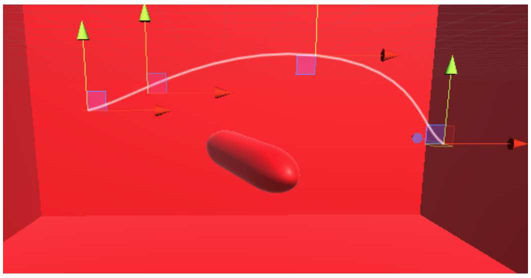

Fig. 3.11: The white line displays the starting position of the cable

every Script and Component. In this phase, we can initialize data we need for our Script. Then, whenever a new frame is rendered, the engine calls `Update()`.

From subsection 2.2.1 we know that rendering and physics calculations occur independently of each other. Therefore, whenever a physical recalculation is needed, the engine calls `FixedUpdate()` of every Component and Script. In there, we will write the code for every model we want to implement.

Unity provides a Component used for collision detection, called *Collider*. There are colliders for every primitive shape, like capsules, boxes, etc. They are each called *Box Collider*, *Capsule Collider*, etc. For non-primitive shapes, Unity provides *Mesh Colliders*. However, whenever possible, Unity recommends using primitive Colliders, since the Mesh Collider performs slower than the primitive ones. Additionally, Mesh Colliders are supposed to be only used for static shapes [20]. Our cable however changes its shape throughout the simulation. Therefore, they are unfit for our purpose.

For physics, Unity provides the *Rigidbody* Component. It requires a Collider component to work. With that Component attached, whenever a collision is detected, the Rigidbody component resolves it by applying forces to the Game Object it is attached to. In other words, when the Game Object collides with another Game Object, it bounces off, depending on its set mass. It can also apply gravitational force on the Game Object.

We will use the Rigidbody Component for both the Elastic and Infinitely Stiff Spring System Model to detect and solve collisions. For the constraint solving models, we will use the Capsule Collider Component to detect and inspect collisions, then resolve them manually.

```
void Start()
{
    // ... some other initialization steps

    //Initialize all child object colliders
    for (int i = 0; i < numberOfParticles; ++i)
    {
        var newT = CreateTransformWithCap();
        capParents[i] = newT;
        SetParticleCapsule(
            ref newT,
            jointPoints[i]
        );

        rigidbodies[i] = newT.GetComponent<Rigidbody>();
    }

    //Initialize length of currentForcesBetween variable
    currentForcesBetween = new Vector3[numberOfJoints];
}
```

Fig. 3.12: `Start()` in the Elastic Spring System model initializes the particles as Child Game Objects with Rigidbodys.

3.3.1 Elastic Spring System Model

Implementation

We first initialize the cable by attaching `numberOfParticles` Child Game Objects to the Game Object representing the cable. The Child Game Objects represent the particles of the simulation and contain an additional Rigidbody Component each. That way, we can apply forces to each of them. Each particle will be constrained not to rotate by setting the corresponding property in its Rigidbody Component. See the code in Figure 3.12.

Instead of recalculating the experienced force for each particle, we store the forces between each pair of particles in the array `currentForcesBetween`.

As mentioned in section 3.3, the `FixedUpdate()` function is called every time when physics are calculated in Unity. For that reason, forces are calculated and applied here. For the second distance constraint (see subsection 2.2.2), we do the same for twice the distance. We do that by iterating from `0` to `maxReach (= 2)`. See Figure 3.13 for the code.

In `UpdateCurrentForces()`, we calculate the forces between two points and store them in the array `currentForcesBetween` (see Figure 3.14). Distance is considered with `distanceFactor`.

```
void FixedUpdate()
{
    for (int i = 0; i < maxReach; i++)
    {
        UpdateCurrentForces( i + 1 );
        ApplyCurrentForces( i + 1 );
    }
}
```

Fig. 3.13: In here, `FixedUpdate()` calculates and applies the forces for each particle.

```
void UpdateCurrentForces(int distanceFactor = 1)
{
    var e = distanceFactor - 1;

    for (int i = e; i < numberOfJoints - e; i++)
    {
        var pT = capParents[i];
        var qT = capParents[i + (int) distanceFactor];

        var sf = CalcSpringForce(pT.position, qT.position,
            distanceFactor);

        currentForcesBetween[i] = sf + df;
    }
}
```

Fig. 3.14: `UpdateCurrentForces()` calculates the in-between forces for single and double distance.

```
void ApplyCurrentForces(int reach = 1)
{
    //end particles (between forces are in the mid of array)
    for (int i = 0; i < reach; i++)
    {
        var bi = i + reach - 1;
        rigidbodies[i].AddForce(
            currentForcesBetween[bi],
            forceMode
        );
        rigidbodies[numberOfParticles - 1 - i].AddForce(
            -currentForcesBetween[numberOfJoints - 1 - bi],
            forceMode
        );
    }

    for (int i = reach; i < numberOfParticles - reach; i++)
    {
        rigidbodies[i].AddForce(
            currentForcesBetween[i] - currentForcesBetween[i -
                reach],
            forceMode
        );
    }
}
```

Fig. 3.15: `ApplyCurrentForces()` applies the forces to the Rigidbodys of the particles calculated in `UpdateCurrentForces()`.

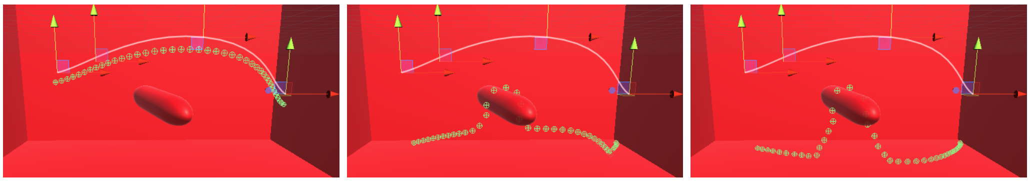

Fig. 3.16: The green dots represent the cable's particles - the distance constraint is enforced too softly, therefore it appears rubbery.

Assuming the forces have been calculated, they are then applied to each pair of particles while keeping the distance in mind with parameter `reach` (see Figure 3.15). Compared to the other particles which experience forces from both sides, the particles at the start and the end of the cable experience only one force. In `ApplyCurrentForces()`, they are taken into consideration as well.

Results

As expected, the cable looks too rubbery (see Figure 3.16). Additionally, collision is not checked between two particles, allowing other bodies to pass through the cable in some cases. We could counteract that problem by setting the Capsule Colliders

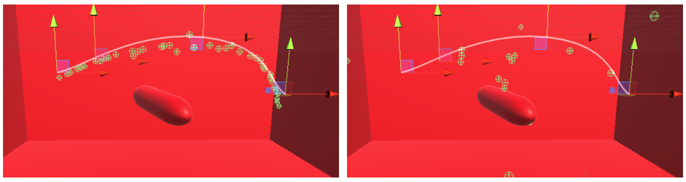

Fig. 3.17: A high spring constant on the other hand breaks the system. Note how the particles are placed seemingly randomly in the room.

between two particles instead of just one. Because the distance between the particles varies throughout the simulation, the shape of the Capsule Colliders needs to change every fixed time step. But that would cause problems with the collision detection, since automatic collision detection happens before the shape shifting. Thirdly, when the spring constant is set too high, not only the vibrations are introduced, but the system also collapses, letting them all fly into space (see Figure 3.17). Therefore, we will stop elaborating that approach.

3.3.2 Infinitely Stiff Spring System Model

Implementation

We can almost completely reuse the code from the Elastic Spring System Model with only minor modifications. First, instead of calculating the actual spring forces, we calculate the distances required between the particles to satisfy the distance constraint. Then, we store them in an array (see Figure 3.18).

Then we can use `MovePosition()` from Unity's Rigidbody class to correct the particle's positions each (see Figure 3.19). The engine will prevent movement that would cause penetration.

Results

The cable first adapts to the colliding objects form. Then, it starts to stretch out. After that, it falls through, because collision between two particles is not checked (see Figure 3.20). So as suspected, the distance constraint is not always satisfied as soon as a collision occurs (see Equation 2.2.5). That means this approach still is not the right one yet.

```
void UpdateCurrentForces(int distanceFactor = 1)
{
    var e = distanceFactor - 1;

    for (int i = e; i < numberOfJoints - e; i++)
    {
        var pT = capParents[i];
        var qT = capParents[i + distanceFactor];

        //instead of forces, we calculate the corrective distance
        currentForcesBetween[i] = CalcDistance(
            pT.position,
            qT.position
        );
    }
}

Vector3 CalcDistance(Vector3 p, Vector3 q)
{
    var dir = q - p;
    return .5f * ( dir - dir.normalized * restDistance );
}
```

Fig. 3.18: `UpdateCurrentForces()` and `CalcDistance()` - In the stiff model, the force between two particles is implicitly given by calculating the necessary correction to satisfy the distance constraint.

```
void ApplyCurrentForces(int reach = 1)
{
    //end particles (between forces are in the mid of array)
    //...

    for (int i = reach; i < numberOfParticles - reach; i++)
    {
        rigidbodies[i].MovePosition(
            capParents[i].position +
            currentForcesBetween[i] - currentForcesBetween[i -
                reach]
        );
    }
}
```

Fig. 3.19: `ApplyCurrentForces()` - In the stiff model, force application happens implicitly through repositioning.

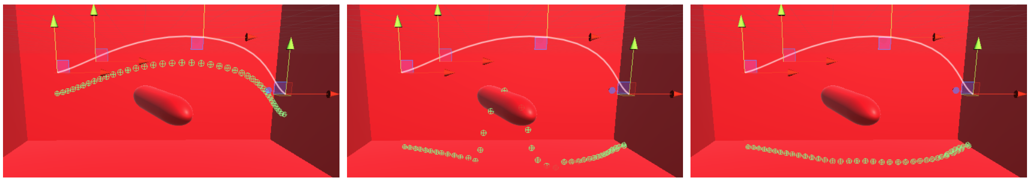

Fig. 3.20: The green dots represent the cable's particles - In the stiff model, the cable begins stretching, then falls through.

```
void Simulate(){
    Integrate();
    for (int i = 0; i < solverIterations; ++i)
    {
        SatisfyConstraints();
    }
}
```

Fig. 3.21: `Simulate()` - Called in `FixedUpdate()`.

3.4 Constraint Solving Models

While we can rely on Unity's collision detection system for the next two systems (see Equation 2.2.5), we have to resolve collisions manually.

3.4.1 Jakobsen's Advanced Character Physics

Implementation - Core Mechanics

Apart from some calls to functions, which are omitted on purpose, because they are not important for the system itself, we only call one method in `FixedUpdate()`. That is `Simulate()` (see Figure 3.21).

In `Simulate()`, we first call `Integrate()`, where the new particle positions are estimated. The function is called so, because from a mathematical viewpoint, it is an approximated integration. Then, as explained in section 2.2.5, we iterate over all points `solverIterations` times, trying to satisfy all constraints we set.

While Jakobsen proposed to store the accelerations in an array for all particles each (see [8]), we will instead store the gravitation vector as the constant `grav` only, because it is the only acceleration applied to every particle. Since the integration relies on Δt^2, we precompute its value and store it in `dt_squared`. `dampening` is yet another constant that can be set to apply dampening forces to particles the way Jakobsen proposes [8]. It can be set by a property in the Component. See Figure 3.22 for the code.

In `SatisfyConstraints()`, we iterate over each particle and pass the index to all our constraints where their satisfactions occur (see Figure 3.23).

`DistanceConstraint()` ensures the satisfaction of the distance constraint between the particle at index `i` and `i + 1 + interD` (see Figure 3.24). By setting `interD`

```
void Integrate()
{
    for (int i = 0; i < numberOfParticles; ++i)
    {
        ref var x = ref currentXs[i];
        ref var x_ = ref previousXs[i];

        var temp = x;
        var deltaPosGrav = grav * dt_squared;
        x = x * (2 - dampening) - (1 - dampening) * x_ +
            deltaPosGrav;
        x_ = temp;
    }
}
```

Fig. 3.22: `Integrate()` - The heart of Jakobsen's method.

```
void SatisfyConstraints()
{
    for (int i = 0; i < numberOfParticles; ++i)
    {
        ref var p = ref currentXs[i];

        //connect particles and prevent folding through itself
        DistanceConstraint(i);
        DistanceConstraint(i, 1);

        //solve collisions
        JointCollisionNative(i);
    }
}
```

Fig. 3.23: `SatisfyConstraints()` - For every tuple, the constraints are satisfied.

```
void DistanceConstraint(int i, int interD = 0)
{
    //requirements
    if (RequirePoints(2 + interD, i)) return;

    var interD1 = interD + 1;

    //points
    ref var p1 = ref currentXs[i];
    ref var p2 = ref currentXs[i + interD1];

    var dir = p2 - p1;
    var cDir =
        .5f * (dir - dir.normalized * restDistance * interD1);

    p1 += cDir;
    p2 -= cDir;
}
```

Fig. 3.24: `DistanceConstraint()` ensures the distance between a particle pair is kept.

```
void JointCollisionNative(int i)
{
    //...

    var cols = Physics.OverlapCapsule(p1, p2, radius, layerMask);
    SetCapsuleFromTo(p1, p2);

    //...
}
```

Fig. 3.25: `JointCollisionNative()` makes use of Unity's built-in `Physics.OverlapCapsule()` function.

once to `0` and another time `1`, we satisfy both distance constraints (see subsection 2.2.2). `RequirePoints()` is a small helper function that just ensures that the particle at index `i + interD1` exists in the array. The distance is then restored exactly the way as described in subsection 2.2.5.

We will take a simple approach at solving collisions and rely on Unity's built-in functions for that. A Child Game Object with an Capsule Collider Component will help us calculating the penetration depth. We will call it *Collision Helper*. The projection happens in `JointCollisionNative()` (see Figure 3.25).

First, we narrow down our search by getting all Colliders overlapping the capsule from particle `p1` to `p2` with the cable radius `radius`. `p1` is the current position of the particle at index `i` and `p2` the one of the particle at index `i+1`. The Collision Helper will also be set from `p1` to `p2` (assuming the radius is already correctly set).

```
void JointCollisionNative(int i)
{
    //...

    for (int j = 0; j < cols.Length; ++j)
    {
        var col = cols[j];

        if (col == cc)
            continue; // skip ourselves

        Vector3 direction;
        float distance;

        bool overlapped = Physics.ComputePenetration(
            cc, //...
            col, //...
            out direction, out distance
        );

        var correctionVector = direction * distance;

        //should always be true, but still
        //checked to prevent potential hazards
        if (overlapped){
            p1 += correctionVector;
            p2 += correctionVector;
        }
}
```

Fig. 3.26: After getting the overlapping colliders, the positions are corrected depending on the penetration in `JointCollisionNative()`.

Then, for each overlapping Collider, we will calculate the penetration depth by using `Physics.ComputePenetration()` from Unity's built-in `Physics` class (see Figure 3.26). This class requires a Collider Component, which is why we introduced the Collision Helper. `Physics.ComputePenetration()` returns the information for the shortest path to resolve the collision. Since `OverlapCapsule()` will likely include the Collision Helper, we need to make sure to skip it. After that, we can apply the correction. This approach is called *discrete collision checking*. `correctionVector` is the minimal vector we need to move the current joint to resolve the collision. We will refer to it as *correction vector* in later sections.

Please note that this method to resolve collisions does not take inter-collision into account. This is to keep the calculation efficient. More on that in chapter 4.

While the discrete collision checking will probably suffice for our purposes, it might fail in certain circumstances. For example, if the Capsule Collider starts before another Collider and moves behind the Collider after one fixed time step, no collision

```
void ParticleCollisionConstraintConSim(int i)
{
    //continuous collision detection for particles
    ref var p1 = ref currentXs[i];
    ref var p1_ = ref previousXs[i];

    var direction = p1 - p1_;

    var hits = Physics.SphereCastAll(p1_, radius, direction,
        direction.magnitude, layerMask);

    for (int j = 0; j < hits.Length; ++j)
    {
        ref var hit = ref hits[j];
        if (hit.distance == 0f) continue; //according to Unity Docs

        //projection to correct distance
        var tangent = Vector3.ProjectOnPlane(p1 - hit.point,
            hit.normal);
        p1 = hit.point + hit.normal * radius + tangent;
    }
}
```

Fig. 3.27: An implementation for continuous collision checking.

would be detected. This can sometimes happen if the cable moves very fast or is approached by another fast object. Müller et al. therefore propose a solution sometimes called *continuous collision detection* [15]. It works by shooting a ray from a particle's before-position to a particles after-position. In case a intersection is detected, it means a collision occured. Since Jakobsen requires that every particle's last position is saved, implementing it would not require additional space. In Unity, the ray can also have a radius. The corresponding function is called `Physics.SphereCastAll()`. The implementation is shown in Figure 3.27.

One disadvantage would be that penetration between particles would once again not be detected. We would require a ray that morphs from one capsule to another, checking for intersections. Unity does not provide such a function (at least not for morphing capsules) and writing a function ourselves would likely require too much time. Alternatively, both collision resolvers could be used simultaneously. But that would cause a large overhead and still not cover all cases of collisions. Therefore, we will leave out continuous collision detection.

Results - Core Mechanics

Compared to the elastic and infinitely stiff spring system model, collision resolving now seems to work well. The cable slides along the obstacle instead of falling

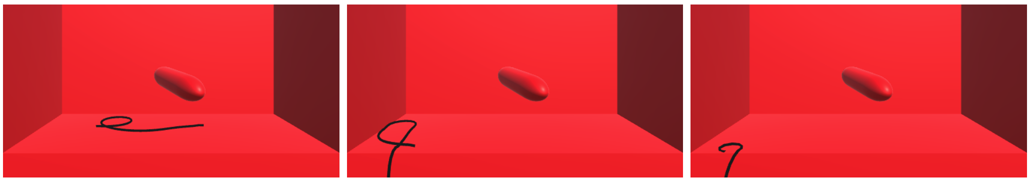

Fig. 3.28: The cable model suffices, but it slides like a puck on an air hockey table.

```
void JointCollisionNative(int i)
{
    //...

    for (int j = 0; j < cols.Length; ++j)
    {
        //...

        if (overlapped){
            p1 += correctionVector;
            p2 += correctionVector;
        }

        FrictionForNative(i, distance, direction);
        FrictionForNative(i + 1, distance, direction);
    }
}
```

Fig. 3.29: Since friction happens while colliding, it makes sense to apply it in `JointCollisionNative()`.

through. Also the distances between the particles are kept. However, because it lacks friction, the cable does not stop moving once it reaches the floor (see Figure 3.28).

Implementation - Friction

Since friction only occurs when an object collides on another, it makes sense to include it in the collision detection (see Figure 3.29).

Jakobsen proposes an algorithm to simulate friction, which will be implemented loosely [8]. Depending on the penetration depth, the velocity is implicitly changed by moving $x*$ closer to x (see Figure 3.30). If the tangential velocity is below a certain threshold, we will simulate static friction by setting $x*$ to x, stopping movement of the particle altogether.

While this model is not accurate, it suffices to the illusion.

```
void FrictionForNative(int i, float distance, Vector3 direction)
{
    ref var p1 = ref currentXs[i];
    ref var p1_ = ref previousXs[i];

    var p1_p1 = p1 - p1_;

    var t1 = Vector3.ProjectOnPlane(p1_p1, direction);

    if (t1.magnitude < staticFriction)
    {
        p1_ = p1;
        return;
    }

    p1_ += p1_p1.normalized * slidingFriction * distance;
}
```

Fig. 3.30: `FrictionForNative()` calculates and applies friction depending on the penetration depth.

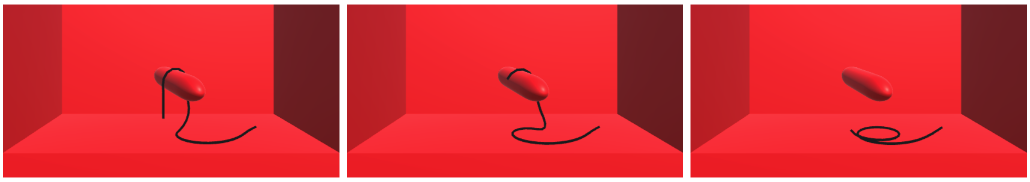

Fig. 3.31: The cable coming to a stop.

Results - Friction

Now, as shown in Figure 3.31, the cable slows down movement when sliding along a surface and comes to a stop.

Accessory Work

By introducing soft and hard position constraints, where former breaks on a certain threshold while the latter does not, we can simulate the effect of pulling a cable out of a socket (see Figure 3.32).

3.4.2 (Extended) Position Based Dynamics - (X)PBD

Derivatives of a function can be calculated either algebraically or approximated numerically. The former however, is computationally more expensive, so therefore

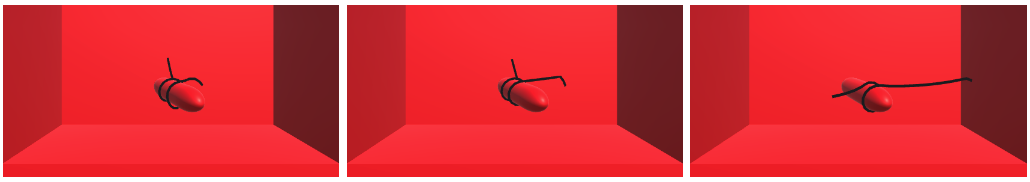

Fig. 3.32: With added location constraints, when pulling the cable strong enough to the right, it frees itself from the center, looking like it has been pulled from a socket.

we will opt for the latter. We will use the approximated three-point centered-difference formula by Sauer to calculate it [18, p.246]:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{3.1}$$

Approximating derivatives tends to be numerically unstable for a very small h [18]. But increasing it, causes loss of precision. We will therefore allow the developer to set the value for h.

We will create a class `Derivatives` in the namespace `NumericalVectors` that will provide us with the necessary functions to create the gradient of a constraint. Just like with the splines in section 3.2, we will opt for a functional approach. That way, we can create the gradient function once on start of the simulation and then use it every time, we want to satisfy the corresponding constraint. See the code in Figure 3.33.

Please note that `vs` is an array of `Vector3`s, not a single one.

In subsection 3.4.1, we implemented the constraints as methods of the Script. For (X)PBD however, we will instead create another class `Constraint` where we will create the solver function in the constructor with `CreateNablaAndSolver()` and (among other data) store the λ-values for every constraint. See Figure 3.34 for the code.

Now, we can pass the constraints as λ-functions to the constructor. See the code for the single distance constraint in Figure 3.35 and for the collision constraint in Figure 3.36. Please note that friction is simulated by calling `FrictionForNative()`. Usually we should either include the velocity or the previous position in the `Vector3` array passed to the `vs` parameter in the λ-function. Then we could change the velocity with the gradient. In our case however, we somewhat abuse the possibility of inducing side-effects to simulate friction because else, we would have to restructure, how the data is internally organized, which would take a lot of time. Therefore, we have chosen the other option.

```
public static Returns<Vector3[]>.Expects<Vector3[]> Nabla(
        Returns<float>.Expects<Vector3[]> f,
        int arity,
        float epsilon = .01f
    )
    {
        //arity * 3, because we need the
        //partial derivative for each coordinate
        var arity3 = arity * 3;
        var nabla = new Returns<float>.Expects<Vector3[]>[arity3];

        for (int i = 0; i < arity3; ++i)
        {
            nabla[i] = PartialDerivative(f, i, epsilon);
        }

        return (vs) =>
        {
            var rs = new Vector3[arity];

            for (int i = 0; i < arity; ++i)
            {
                var i3 = i * 3;
                var v = new Vector3(
                    nabla[i3](vs),
                    nabla[i3 + 1](vs),
                    nabla[i3 + 2](vs)
                );

                rs[i] = v;
            }

            return rs;
        };
    }
```

Fig. 3.33: `Nabla()` creates a function, calculating the gradient at `vs` of function f.

```
void CreateNablaAndSolver()
{
    nabla = Derivatives.Nabla(
        constraintFunction,
        cardinality,
        precision
    );

    constraintSatisfier = (vs) =>
    {
        //get directional nablas
        var directionalNablas = nabla(vs);

        //catch zero (no projection needed then)
        var sum = SumOfVectorsQuad(directionalNablas);
        if (sum == 0f)
        {
            return vs;
        }

        float s;
        if (extended)
        {
            //recalculated, so it can be changed in play mode
            var alphaSnake = 1f / (deltaTSquared *
                (minStiffnessExt + XPBDStiffness())
            );

            s = XPBDScale(ref vs, alphaSnake, sum);

            lambdas[particleIndex] += s;
        } else
        {
            s = -stiffness * constraintFunction(vs) / sum;
        }

        //correct each particle's position
        for (int i = 0; i < cardinality; i++)
        {
            vs[i] += s * directionalNablas[i];
        }

        return vs;
    };
}
```

Fig. 3.34: `CreateNablaAndSolver()` uses `Nabla()` to dynamically create the solver function for any constraint. It includes λ in the calculation, if `extended` is `true`.

```
//single distance constraint
constraints[0] = new Constraint(2, (vs) =>
{
    return Mathf.Abs((vs[0] - vs[1]).magnitude) - restDistance;
}, /* ... */ );
```

Fig. 3.35: The single distance constraint for (X)PBD.

```
constraints[2] = new Constraint(2, (vs) =>
{
    ref var p1 = ref vs[0];
    ref var p2 = ref vs[1];

    //...

    for (int j = 0; j < cols.Length; ++j)
    {
        //...

        if (overlapped)
        {
            p1 += correctionVector;
            p2 += correctionVector;

            FrictionForNative(particleIndex, distance, direction);
            FrictionForNative(particleIndex + 1, distance,
                direction);
        }
    }

    return correctionVector.magnitude;
}, /* ... */ );
```

Fig. 3.36: The collision constraint for (X)PBD.

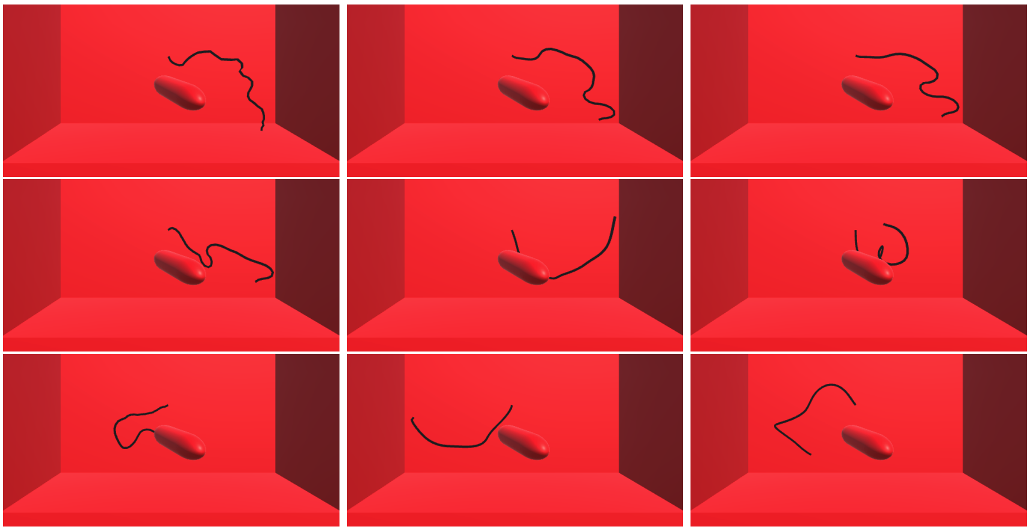

Fig. 3.37: The stiffness effects in Jakobsen's model.

3.4.3 Comparision and Stiffness Effects

So far, we have not taken a look at how increasing the number of iterations and shortening the fixed time span Δt affects the cable's physics, especially regarding the stiffness effects explained in section 2.2.5.

Figure 3.37, Figure 3.38 and Figure 3.39 show, how with different values for `solverIterations` and Δt the behaviour of the cable changes with equal properties. The pictures were taken $0.25s$ after the simulation started. The first, second and third column represent `solverIterations = 1, = 12` and `= 25`, and the first, second and third row represent $\Delta t = 0.02s, = 0.002s$ and $= 0.0002s$ respectively. With increasing stiffness due to the accuracy- and timescale-stiffness effect, the tension in the cable increases at the start, causing it swing more to the opposite direction. However, notice how the swinging does not appear in XPBD (Figure 3.39). Additionally, notice that all systems produce (very) similar results for $\Delta t = 0.02s$, even with an increasing value for `solverIterations`.

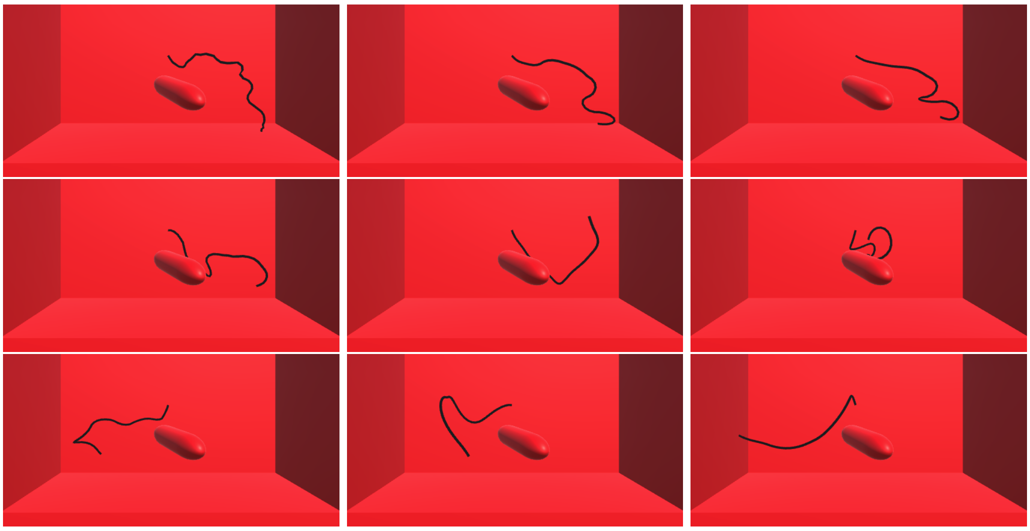

Fig. 3.38: The stiffness effects in PBD.

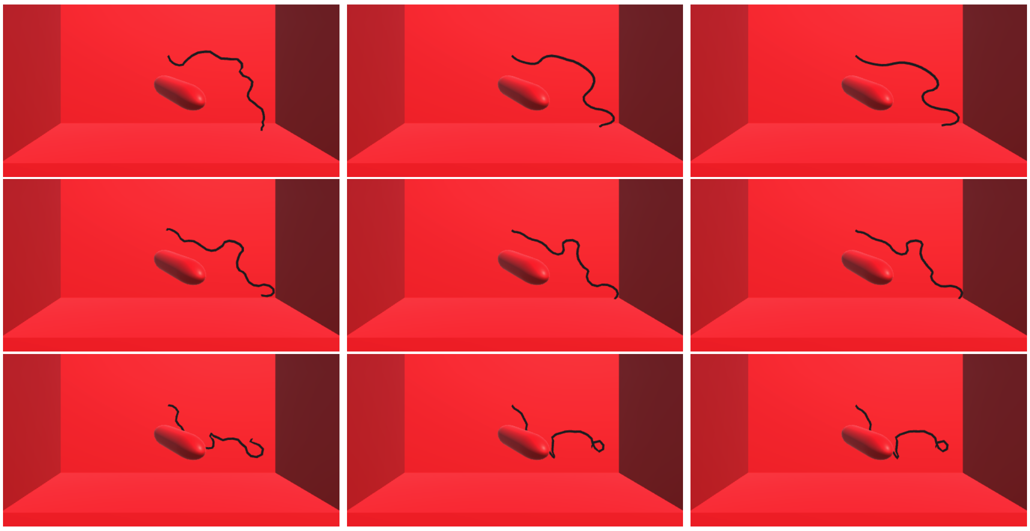

Fig. 3.39: The stiffness effects in XPBD.

Efficiency

4

We will take a look at the time complexities, ideas for parallelization and finally inter-collision. Because we skipped the full implementation of Unity's Rigidbody dependant algorithms, we will also skip their efficiency analysis.

4.1 Time Complexity

While we can analyse the time complexities of the algorithms, we cannot analyse the ones of the Unity Engine due to its proprietary nature. The most important one is probably the collision detection system. Whenever needed, we will include it as an unknown function $t(n)$ into the Big-O-Notation. More precisely, $t(n)$ is the time complexity of a collision check for one Collider if there are n Colliders in the Scene.

Among others, there are two ways to detect collisions. The first one is by just comparing the distance for each object with every other object. That would mean that if we had n Colliders, it would take $O(n)$ checks for one Collider to check against the others. That means it would be $O(n^2)$ for all objects. We could optimize it by first checking for overlapping Colliders, then calculate the penetration depth (like we did in subsection 3.4.1), but that would still leave the complexity at $O(n^2)$, since every Collider still needs to be matched against every other one. However, there are collision detection algorithms, where colliders are organised in trees depending on their distances to each other. Collision detection would then be reduced to $O(n * log(n))$, since for every Collider, the tree has to be searched for other Colliders (assuming the tree is balanced). If objects move, the tree needs to be rebalanced [1]. Unlike in AVL-Trees (binary search trees that balance themselves after inserting a node), rebalancing would not require correcting only one but multiple wrongly placed nodes, because with movement, more than one node might become wrongly placed. In the worst case, the whole tree would have to be reconstructed, which takes linear time $O(n)$. So we can assume that $t(n) \in O(n) \cap \Omega(log(n))$.

Let n be the number of Colliders excluding the ones of the cable, k be the number of particles and c be the number of constraints excluding the collision constraints. All but the collision constraint run in constant time $O(1)$ for each particle, because they run one operation on a tuple of particles. This confirms that the collision checks are the bottleneck of the system, as Jakobsen states [8]. Since in all algorithms, we iterate over every particle, apply a constant number of operations and then check for collisions against other Colliders, the time complexity is $O(n * (c + t(k))) = O(n * t(k))$. This excludes inter-collision checks. If we included them, it would be $O(n * (c + t(k + n))) = O(n * t(k + n))$.

4.2 Inter-Collision

In subsection 3.4.1 and subsection 3.4.2 we programmed a simple system to resolve collisions of the cable with other objects. However, due to the nature of our implementation, it does not include inter-collision checks. That means if one part of the cable collides with another part of the cable, it would wander right through it. This also makes the user unable to create knots. However, it reduces the number of collisions checks significantly, because otherwise, it would add the number of particles the cable consists of. This is relevant because in section 4.1, we learned that resolving collisions can have a time complexity of up to $O(n^2)$.

However, in case we would want to include it, we could apply a simple way to reduce the number of checks without using trees. Assuming we had n particles for the cable, let $c_1, ..., c_{n-1}$ be all Capsule Colliders representing the cable's joints. First, we can skip collision checks for the neighbours of c_i for any $i \in \{1, ..., n-1\}$ (more specifically c_{i-2}, c_{i-1} and c_{i+1}), because of both the distance constraints (see subsection 2.2.2). Additionally, we can approximately reduce collision checks by half inductively:

Let c_i be the Collider which we would like to check for collisions with c_j for any $j \in \{i + 2, ..., n - 1\}$, assuming collisions with c_k for any $k \in \{1, ..., i - 3\}$ have already been resolved. Then, to resolve the collision between c_i and c_j, we would move c_i by half of the correction vector (for the definition, see subsection 3.4.1) and c_j by half of the negative correction vector, thus resolving the collision. The base case would be the collision check and resolving of c_1 and c_j for any $j \in \{3, ..., n-1\}$, which is the correct way to check and resolve collisions.

■

This method relies mainly on the fact, that if we checked whether c_i collides with c_j ($i \neq j$), then we do not have to check it again the other way around.

Checking c_i for collisions with c_j takes $n-1-i-1 = n-i-2$ checks and position adjustments because of the $n-1$ Capsule Colliders, from which we only check for collision with the remaining $n-1-i-1$ Colliders. The neighbouring ones, where collision checks are not necessary, are excluded. While the time complexity T_{inter} for checking and resolving all inter-collisions would still be

$$\begin{aligned} T_{inter} &= \Sigma_{i=1}^{n-3}(n-i-2) \\ &= (n-4)+(n-5)+...+(n-(n-3)-2) \\ &= (n-4)+(n-5)+...+1 \\ &= \frac{(n-4)(n-3)}{2} \\ &= \frac{n^2-7n+12}{2} \in O(n^2), \end{aligned} \tag{4.1}$$

the overhead would still be significantly reduced.

4.3 Ideas for Parallelization

If all constraints only required one point, parallelization would be trivial: We could divide the array in n equal parts and let each part be parsed by a thread satisfying the constraints. n would depend on the time it takes to start the threads and prepare their data. Ideally, n should depend on the time $t_{threads}$ it takes to prepare, start and finish executing all threads. n needs to be set such that $t_{threads}$ is smaller than the time it takes to just serially iterate over all particles. We will refer to this n as n_{ideal}. If $n_{ideal} = 1$, then no thread will be created.

In our case however, we also have constraints requiring more than one point. That causes a problem. The result of $\Delta\mathbf{p}_i$ and of a particle $\mathbf{p}_i$ depends on the result of $\Delta\mathbf{p}_{i-1}$. So if we divided the particle array into $n > 1$ parts, every thread t_i transforming array part a_i would have to wait until t_{i-1} finished its execution. To counteract this problem, we could divide the satisfaction of the constraints into two phases. In the first phase, we apply all transformations to each a_i, treating them as one separate cable each, thus allowing us to transform each a_i in a separate thread. In the second phase, we would calculate the Δp_i and $\Delta\mathbf{p}_{i+1}$ at the places where the array was separated. Then we would apply $\Delta\mathbf{p}_i$ and $\Delta\mathbf{p}_{i+1}$, not only to $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$ respectively, but to the all points in a_i and a_{i+1} each. This is possible because all constraints project at most two particles, even if they require more than two (like the second distance constraint).

Then, just like before, we would repeat the process multiple times, until sufficient accuracy has been achieved.

Conclusion

5

5.1 Summary

Our goal was to create a realistically looking cable. We analysed how cable physics can be simulated, wrote code as efficient as time allowed and stayed close to the mathematical concepts by using functional programming concepts, where useful.

At first, we took a look at different types of splines to describe the initial position of the cable. We picked the Catmull-Rom spline because it does not change the whole spline when a control point's position is changed and because of its smoothness, without manually setting its tangents. After that, we took a look at a handful of systems for simulation. We analyzed their advantages and disadvantages and compared them to the other systems we discussed. Then, we jumped to the implementation.

First, we created a Component to set the initial position of the cable in the Scene. This was where we made use of our Spline class, allowing us to let the line representing the starting position to look smooth. With the initialisation done, we took the next step by implementing all the algorithms we discussed in theory. Conclusively, we ascertained that the Jakobsen, PBD and XPBD system delivered the most satisfactory results. However, compared to Jakobsen and PBD, XPBD delivered the most easily configurable results, because it counteracts both the accuracy- and timescale-stiffness effect. All three systems delivered comparable and satisfying results.

Finally, we analyzed the algorithms in terms of efficiency. We concluded the collision check is the bottleneck of the simulation because of its time complexity compared to the other constraints.

5.2 Further Work

One could implement the other systems mentioned in subsection 2.2.6 and compare the results to the ones we picked. That would give a broader look at the approaches that can be taken and how they perform compared to each other.

In subsection 3.4.1, we briefly explored an idea for continuous collision detection. We discussed the possibility of using Unity's built-in function `Physics.SphereCastAll()`. However, it was deemed unfit because it lacks collision detection between particles. Further work could be done by conceptualizing and implementing a Capsule Cast function which morphs from one Capsule to another. While Unity provides the built-in `Physics.CapsuleCastAll()` function, it lacks the morphing ability.

We used `Vector3`s to represent the particle's locations. However, it would make sense to represent the whole state of the system as a `float` array. It would not only contain the locations but other properties like the velocities or, even better, the last positions as well. That way, the system would not have to rely on side effects to simulate friction (see subsection 3.4.2). Representing the system state that way is also suggested by Macklin et al. [11].

Additionally, the friction model itself was weak, since it was not accurate. Therefore working out a better one would be useful. While it could have been implemented more closely to Jakobsen's suggested model, it would still be inaccurate. In fact Jakobsen himself said about his model: "Other and better friction models than this could and should be implemented" [8].

Another aspect that could be further explored is air resistance. For example Eberhardt et al. propose a solution included in their clothing simulation system [3]. It could also be applied to the cable model.

In our (X)PBD implementation (see subsection 3.4.2), we pass the `Vector3` instances directly to the solver function instead. Alternatively, we could pass references to the arrays containing the current and the last positions each with an additional index. That way, we could change the values directly, skipping the overhead of copying the tuple.

As discussed in section 4.3, parallelisation is difficult to achieve because the results of the constraint solvers depend on each other. However in Macklin et al., Müller et al. and Jakobsen, constraints are bound per tuple instead of being defined globally [11, 15, 8]. In other words, we have for example a distance constraint for $\mathbf{p}_1$ and $\mathbf{p}_2$, another one for $\mathbf{p}_2$ and $\mathbf{p}_3$ and so on. We will call them *local constraints*. There is no specification, in which order local constraints have to be solved to converge to a working solution. That implies the order, in which we solve the local constraints, does not matter. This is an important fact for parallelisation since that means, every thread t_i can solve every local constraint without waiting for another thread to finish. It also implies, we can skip the second phase mentioned in section 4.3 to unify the result. Therefore, there is further work that can be done in terms of parallelisation.

Acknowledgements

The author would like to sincerely thank his first reviewer Dr. Stefan Endler for his intensive support, taking the time walking through the content, giving suggestions & ideas, helping with time management and generally always offering a helping hand. Additional thanks to Henrik Winnemöller for helping with ideas, checking in regularly and always offering a helping hand as well. More thanks to Prof. Dr. Klaus Wendt for making sure, priorities are set right.

Finally, the author wishes to thank Arkady Amiragov and Severin Hasselbach for proof-reading extensively and Batiste Erdmann for additional suggestions.

Bibliography

[1] James Bruce. *Chapter 3 - Collision Detection* (cit. on p. 47).

[2] *Curves and Splines, a Unity C# Tutorial* (cit. on p. 27).

[3] Bernhard Eberhardt, Andreas Weber, and Wolfgang Strasser. “Particle-Syst ode1for CI raping”. en. In: *IEEE Computer Graphics and Applications* (1996), p. 8 (cit. on pp. 9, 18, 52).

[4] Epic Games. *Download Unreal Engine*. en-US (cit. on p. 1).

[5] GDC. *Physics for Game Programmers: Understanding Constraints*. Mar. 2018 (cit. on pp. 12, 14).

[6] *Hitman: Codename 47 - IO Interactive*. Aug. 2020 (cit. on p. 14).

[7] *How Virtual Reality Military Applications Work*. en. Aug. 2007 (cit. on p. 1).

[8] Thomas Jakobsen. *Advanced Character Physics*. Apr. 2008 (cit. on pp. 9, 12–15, 34, 39, 48, 52).

[9] JGU-Mainz. *Virtual-Reality-Experimente* (cit. on p. 1).

[10] Eric Lengyel and Emi Smith. *Mathematics for 3D game programming and computer graphics, third edition*. English. OCLC: 804868250. Boston: Cengage Learning, 2011 (cit. on pp. 4, 5, 7, 9–11).

[11] Miles Macklin, Matthias Müller, and Nuttapong Chentanez. “XPBD: position-based simulation of compliant constrained dynamics”. en. In: *Proceedings of the 9th International Conference on Motion in Games - MIG ’16*. Burlingame, California: ACM Press, 2016, pp. 49–54 (cit. on pp. 9, 16, 17, 52).

[12] Steve Marschner and Peter Shirley. *Fundamentals of Computer Graphics, 4th Edition*. English. OCLC: 1105773916. Place of publication not identified: A K Peters/CRC Press, 2018 (cit. on pp. 3, 5, 6).

[13] Brian Vincent Mirtich. “Impulse-based Dynamic Simulation of Rigid Body Systems”. en. In: (), p. 259 (cit. on p. 18).

[14] Christian Moro, Zane Štromberga, and Allan Stirling. “Virtualisation devices for student learning: Comparison between desktop-based (Oculus Rift) and mobile-based (Gear VR) virtual reality in medical and health science education”. en. In: *Australasian Journal of Educational Technology* 33.6 (Nov. 2017). Number: 6 (cit. on p. 1).

[15] Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. *Position Based Dynamics* (cit. on pp. 9, 15–17, 38, 52).

[16] Grant Palmer. *Physics for game programmers: [a one-stop resource for building pysics-based realism into your games ; sample games in Java, C# and C*. English. OCLC: 315547427. Berkeley, Calif.: APress, 2005 (cit. on pp. 8, 9, 13, 18).

[17] Bashir Salah, Mustufa Haider Abidi, Syed Hammad Mian, et al. “Virtual Reality-Based Engineering Education to Enhance Manufacturing Sustainability in Industry 4.0”. en. In: *Sustainability* 11.5 (Jan. 2019). Number: 5 Publisher: Multidisciplinary Digital Publishing Institute, p. 1477 (cit. on p. 1).

[18] Timothy Sauer. *Numerical analysis*. English. OCLC: 1113352962. Boston, Mass: Pearson, 2012 (cit. on p. 41).

[19] Unity Technologies. *Powerful 2D, 3D, VR, & AR software for cross-platform development of games and mobile apps.* en (cit. on p. 1).

[20] Unity Technologies. *Unity - Manual: Colliders*. en (cit. on p. 28).

[21] Unity Technologies. *Unity - Manual: Creating Gameplay*. en (cit. on pp. 11, 15).

[22] Unity Technologies. *Unity - Scripting API: Editor.OnSceneGUI()*. en (cit. on pp. 8, 11, 20, 27).

[23] John E. Wilson. *Digging Deeper Into Rhino Surfaces | Cadalyst*. Feb. 2002 (cit. on p. 4).

[24] Wirtschaftswoche. *Virtual Reality: Lufthansa schult Flugbegleiter mit Datenbrillen und VR*. de (cit. on p. 1).

List of Figures

2.1 A sketch of how the degree of continuity affects the curvature [23]. . . . 4
2.2 A sketch of an Extended Cubic Catmull-Rom Spline. The dashed lines represent the tangents. 7
2.3 A sketch of the cable model. On one particle, the distance constraints have been visualized. The green lines represent the single, the magenta lines the double distance constraints. 10
2.4 Euler's method - If the location, in this figure the z-axis, changes non-linearly, the approximation is inaccurate [16, p.57, fig.4-4]. 13

3.1 The `Functional` namespace provides functions to include typesafe functional programming paradigms. 21
3.2 Example: `MyFunction()` - Returns a function that expects an `int` and returns a `Vector3`, where each component of it is the given `int`. 22
3.3 Example: `ExecuteAndReturnResult()` - A demonstration of how the signature of parameter `f` can now be defined. 22
3.4 Example: `Power2Point3()` - A demonstration of how `Map()` can be used to iterate over an array. 22
3.5 `CreateGluedFunction()` - Turns multiple curves into one spline. 23
3.6 `CreateNormalizedFunction()` - Normalizes the function while allowing `t < 0f` and `t > 1f`. 24
3.7 `CreateAutoCubicBezierPoints()` - Calculates the tangent points and includes them in the control points, such that they become points of a Catmull-Rom Spline. 25
3.8 `CreateAutoCubicBezierPoints()` - Taking care of the end points. . . . 25
3.9 `CreateCubicBezierSplineFunction()` - If the points for the Spline are generated, we create the function representing the spline by first creating multiple Bezier curves, transforming them into one spline function, then normalize the spline. 26
3.10 `Handles.DrawAAPolyLine` in `OnSceneGUI()` allows us to display the spline in the scene. 27
3.11 The white line displays the starting position of the cable 28
3.12 `Start()` in the Elastic Spring System model initializes the particles as Child Game Objects with Rigidbodys. 29

3.13 In here, `FixedUpdate()` calculates and applies the forces for each particle. 30
3.14 `UpdateCurrentForces()` calculates the in-between forces for single and double distance. 30
3.15 `ApplyCurrentForces()` applies the forces to the Rigidbodys of the particles calculated in `UpdateCurrentForces()`. 31
3.16 The green dots represent the cable's particles - the distance constraint is enforced too softly, therefore it appears rubbery. 31
3.17 A high spring constant on the other hand breaks the system. Note how the particles are placed seemingly randomly in the room. 32
3.18 `UpdateCurrentForces()` and `CalcDistance()` - In the stiff model, the force between two particles is implicitly given by calculating the necessary correction to satisfy the distance constraint. 33
3.19 `ApplyCurrentForces()` - In the stiff model, force application happens implicitly through repositioning. 33
3.20 The green dots represent the cable's particles - In the stiff model, the cable begins stretching, then falls through. 33
3.21 `Simulate()` - Called in `FixedUpdate()`. 34
3.22 `Integrate()` - The heart of Jakobsen's method. 35
3.23 `SatisfyConstraints()` - For every tuple, the constraints are satisfied. . 35
3.24 `DistanceConstraint()` ensures the distance between a particle pair is kept. 36
3.25 `JointCollisionNative()` makes use of Unity's built-in `Physics.OverlapCapsule()` function. 36
3.26 After getting the overlapping colliders, the positions are corrected depending on the penetration in `JointCollisionNative()`. 37
3.27 An implementation for continuous collision checking. 38
3.28 The cable model suffices, but it slides like a puck on an air hockey table. 39
3.29 Since friction happens while colliding, it makes sense to apply it in `JointCollisionNative()`. 39
3.30 `FrictionForNative()` calculates and applies friction depending on the penetration depth. 40
3.31 The cable coming to a stop. 40
3.32 With added location constraints, when pulling the cable strong enough to the right, it frees itself from the center, looking like it has been pulled from a socket. 41
3.33 `Nabla()` creates a function, calculating the gradient at `vs` of function f. 42

3.34 `CreateNablaAndSolver()` uses `Nabla()` to dynamically create the solver function for any constraint. It includes λ in the calculation, if `extended` is `true`. . . . 43
3.35 The single distance constraint for (X)PBD. . . . 44
3.36 The collision constraint for (X)PBD. . . . 44
3.37 The stiffness effects in Jakobsen's model. . . . 45
3.38 The stiffness effects in PBD. . . . 46
3.39 The stiffness effects in XPBD. . . . 46

Colophon

This thesis was typeset with $\LaTeX\,2_\varepsilon$. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.