



IT UNIVERSITY OF COPENHAGEN

ROBOTICS, EVOLUTION, AND ART LAB

**A DATA-ORIENTED APPROACH TO SOLVING
THE FORWARD DYNAMICS PROBLEM**

MASTER'S THESIS

AUTHOR

BORIS KARAVASILEV

SUPERVISOR

ANDRES FAIÑA, SEÁN PARKINSON

Copenhagen 2023

Abstract

Within this thesis an implementation of Featherstone's algorithm, also known as the articulated body algorithm, was developed. The implementation is data-oriented and first-of-its-kind based on the conducted research. Its computational performance appears to be competitive with PhysX 4.1, one of the physics engines integrated into the leading development platforms for real-time 3D experiences, Unity. The collected data suggests that the data-oriented implementation is ten times faster than the simulation step of PhysX within Unity. Among the possibilities for future work is improving the correctness of the implementation. The tests showed that when simulating a double pendulums with a time step of 1/60 of a second the mechanical system gains energy over time instead of maintaining constant energy. Data about performance and accuracy was collected in a series of eight tests, which may hold valuable insights to the possible source of the implementation error.

Keywords

Forward Dynamics, Featherstone's Algorithm, Articulated Body Algorithm, Data-Oriented Design, DOTS, Unity

Reference

KARAVASILEV, Boris. *A data-oriented approach to solving the forward dynamics problem*. Copenhagen, 2023. Master's thesis. IT University of Copenhagen, Robotics, Evolution, and Art Lab. Supervisor Andres Faiña, Seán Parkinson

A data-oriented approach to solving the forward dynamics problem

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Andres Faiña and Seán Parkinson. I have listed all the literary sources, publications, and other sources, which were used during the preparation of this thesis.

.....
Boris Karavasilev
September 5, 2023

Acknowledgements

I would like to thank both of my supervisors Andres and Seán for their guidance, constructive feedback, and words of encouragement when I needed them. I am also very grateful for the many hours of guidance by Seán Parkinson, Daniel Holz and other members of the physics team at Unity Technologies who enabled me to learn about rigid body simulations, data-oriented design and software development in general through this thesis. My gratitude also belongs to all the other employees of Unity that took the time to share knowledge with me and supported me on my journey. Last but not least, I would like to express my gratitude to my dear partner Teresa Bundgård as she encouraged me to stay focused and optimistic when I needed it the most.

Contents

1	Introduction	3
1.1	Problem Statement	4
1.2	The Goal of This Thesis	4
1.3	Structure	5
2	Rigid Body Simulation	6
2.1	Taxonomy	6
2.1.1	Kinematic vs. Dynamic Methods	7
2.1.2	Direct vs. Iterative Methods	7
2.1.3	Maximal Coordinate vs. Reduced Coordinate Formulation	8
2.2	Collision Detection	9
2.2.1	Broad-Phase	10
2.2.2	Narrow-Phase	11
2.2.3	Contact Determination	11
3	Featherstone’s Algorithm	12
3.1	Used Terminology	13
3.2	Table of Symbols	15
3.3	Link and Joint Indexing	16
3.3.1	Kinematic Chains	16
3.3.2	Kinematic Trees	17
3.4	Pass 1	18
3.5	Pass 2	23
3.6	Pass 3	26
4	Data-Oriented Design	27
4.1	Motivation	27
4.2	Principles of Data-Oriented Design	30
4.2.1	Data Storage Patterns Matter	30
4.2.2	Data Access Patterns Matter	31
4.2.3	Vectorization (SIMD)	31
4.2.4	Padding	32
4.2.5	Prefetching	33
4.2.6	Solve Problems You Have	33
4.2.7	Reality is the Problem	33
4.3	The ECS Framework	33
4.3.1	Big Array-based	34
4.3.2	Sparse Sets-based	34

4.3.3	Archetype-based	35
4.4	Unity's Data Oriented Technology Stack	36
4.4.1	Unity's ECS Implementation	36
5	Implementation	37
5.1	Used Technology	37
5.1.1	DOTS	37
5.1.2	Numerics library	38
5.2	Structure	39
5.2.1	Entities	39
5.2.2	Components	39
5.2.3	Systems	39
5.3	Used Conventions	40
5.4	Debugging	40
6	Testing	42
6.1	Accuracy and Energy Conservation	42
6.1.1	Single Pendulum 0°	43
6.1.2	Single Pendulum 45°	45
6.1.3	Single Pendulum 135°	46
6.1.4	Double Pendulum 0°	48
6.1.5	Double Pendulum 45°	50
6.1.6	Double Pendulum 135°	52
6.1.7	Double Pendulum 135° (Smaller Time Step)	54
6.2	Execution time	56
7	Discussion and Future Work	59
8	Conclusion	60
	Bibliography	61

Chapter 1

Introduction

Simulation is an essential tool in many fields such as engineering, robotics, and video game development. Dynamic simulation is a sub-field of simulation that involves the process of modelling and analysing the behaviour of a system over time, taking into account its changing variables and interactions. Examples of dynamic simulation include simulating the movement of vehicles in traffic, predicting the behaviour of a pendulum, or modelling the motion of fluids in a container. A sub-field of dynamic simulation focused on modelling and analysing the motion of rigid bodies (bodies that do not deform or change shape) is called rigid body dynamics. An important challenge in rigid body dynamics is solving the forward dynamics problem, which involves determining the resulting motion of a system given a set of applied forces or torques. This thesis is focused on solving the forward dynamics problem of rigid bodies connected with joints forming so-called *articulated bodies* and more specifically kinematic chains like a pendulum or a robotic arm.

Forward dynamics is a key problem in rigid body simulations. The literature describes a large variety of formulations of dynamic systems i.e. ways of formulating the equations of motion for a dynamic system. Despite the high number of different available formulations, they all generally fall into two categories depending on how they model constraints.

The first category is *reduced-coordinate* formulations also called *minimal-coordinate* or *generalised coordinate* formulations. These take a system with m degrees of freedom and a set of constraints that removes c of those degrees of freedom and parameterise the remaining degrees of freedom using a reduced set of n coordinates ($n = m - c$). Due to the way reduced-coordinate formulations model the system, the constraints are always inherently satisfied. For example, if we model a bead hanging on a string as a simple pendulum with a reduced-coordinate formulation the state of the mechanical system would be defined by only one coordinate e.g. the angular displacement of the string from its equilibrium position. The most significant advantages of reduced-coordinate formulations are: that they often lead to more concise equations of motion, the reduced number of coordinates leads to lower computational complexity, and because only the essential degrees of freedom are captured, the simulations are more stable and robust. Reduced-coordinate formulations also have the following disadvantages: by using fewer coordinates, some details of the system's motion may be lost, and incorporating constraints into the reduced coordinates can be complex, especially when dealing with intricate or nonlinear constraints.

The second category is *maximal-coordinate* formulations which describe a system with its original m degrees of freedom. In systems modelled this way, the constraints must be explicitly maintained by introducing additional forces into the system. For example, if we would model the same example of a bead on a string as in the previous example

with a maximal-coordinate formulation, we might treat the bead as a rigid body with 6 degrees of freedom to which we would have to apply appropriate forces to maintain the constraints which a string would impose upon it (i.e. maintain a constant length of the string and prevent the bead from falling to the ground). Maximal-coordinate representation provides a comprehensive and detailed description of the system's motion. It can capture the positions, orientations, and interactions of all individual components, offering a high level of detail. Maximal coordinate representation can handle various types of constraints more easily. On the other hand, describing a system using maximal coordinates introduces a larger number of variables, making the equations of motion more complex and increasing the computational burden. Also, the increased number of variables in maximal coordinate representation requires more memory to store and process the system's state.

There is a popular method using a reduced-coordinate formulation for computing the forward dynamics of kinematic chains and trees called *Featherstone's algorithm* also known as the *Articulated body algorithm*. This thesis focuses on implementing this algorithm with a data-oriented paradigm as it could yield better performance than the existing implementations using an object-oriented paradigm. The developed implementation will be tested and compared against PhysX 4.1 which is integrated into Unity, the real-time 3D development platform that will be used as an environment for the implementation and testing of the developed solution.

Data-oriented design (DOD) is an alternative approach to object-oriented design (OOD) that focuses on optimising the storage and processing of data. DOD can typically offer higher performance by storing data of the same type contiguously in memory, and afterwards processing this data in bulk to exploit CPU caching and prefetching.

1.1 Problem Statement

Featherstone's algorithm, also known as the articulated body method, is a well-established approach to solving the forward dynamics problem. Currently, to our best knowledge, there does not exist a purely data-oriented implementation of Featherstone's algorithm, meaning that there might be room for implementing a better-performing implementation than the ones currently available. A secondary problem is that the existing resources covering Featherstone's algorithm lack some details that are necessary for its implementation.

1.2 The Goal of This Thesis

This thesis aims to implement Featherstone's algorithm in a purely data-oriented way and evaluate the differences in performance and accuracy when compared to an existing implementation. Furthermore, this thesis aspires to provide an explanation of the algorithm with the details necessary for its implementation by a graduate student of computer science. The implementation will be done in Unity using C# and Unity's data-oriented technology stack (DOTS). The algorithm and its implementation will be described within this thesis and the implementation will be attached as part of the submission.

1.3 Structure

Chapter 2 summarises the current state of rigid body simulation and explains the role of Featherstone’s algorithm within this field. Chapter 3 breaks down and explains Featherstone’s algorithm. Chapter 4 covers the principles of data-oriented design and contrasts it with object-oriented design. Chapter 5 describes the developed data-oriented implementation of Featherstone’s algorithm. Chapter 6 presents the results of the comparison between the data-oriented implementation and an existing solution. Chapter 7 discusses the results of the performed tests and lists possibilities for future work. Chapter 8 concludes this thesis by reflecting on its results and contributions.

Chapter 2

Rigid Body Simulation

Rigid body simulation is a fundamental technique used in computer graphics, robotics, and engineering to model and analyse the motion and interactions of solid objects. In engineering literature, it is often referred to as multibody dynamics. This technique plays an essential role in a wide range of applications, from creating realistic animations in movies and providing interactive animations in video games to designing and controlling robotic systems. A lot of literature is available on this topic across different fields and there is no one way of classifying the existing methods, there are even some overlapping names of different techniques which can cause confusion. Therefore, this chapter gives an overview of the popular simulation paradigms and ways of classifying the existing methods. Special attention is paid to forward dynamics methods because Featherstone's method which is the focus of this thesis falls within this category. This chapter aims to put Featherstone's algorithm in the context of other known methods by highlighting its advantages and limitations. An excellent resource with more in-depth material is the book „Physics-Based Animation“ by K. Erleben et al. [8].

2.1 Taxonomy

There are different ways of classifying rigid body simulation methods depending on what properties we examine. Since this paper studies methods computable on a computer which always has a limited amount of resources, each method has to choose a balance between accuracy, performance and stability. Accuracy means how close is the behaviour of the simulated system compared to a real system. Performance refers to the computational cost of running the simulation. The stability of a simulation means how easy is it to make the simulation behave in a non-realistic way e.g. by using bodies with a very big mass difference. Some methods such as *analytical methods* are able to compute very accurate results but at the cost of not running in real-time or not being applicable in every situation. For example in movie production a longer processing time for the final visual effects is acceptable. On the other hand, in computer games, the player usually expects an interactive experience running at 60 frames per second. For such interactive scenarios, iterative methods such as the projected Gauss-Seidel method are common. To be more precise, it is common to mix different simulation paradigms in modern simulation tools to cover a broader spectrum of user needs and work around the limitations of the individual methods. The following sections describe the common ways of classifying simulation methods based on different

properties. For a more comprehensive state-of-the-art report please see the publication by J. Bender et al. [5] which is the latest report of this kind to our best knowledge.

2.1.1 Kinematic vs. Dynamic Methods

On the highest level, rigid body simulations can be split into two big groups, kinematic and dynamic methods. Both of which can be further divided into forward and inverse methods [8]. Kinematic methods compute the positions and movement of bodies without taking forces into account, they use the knowledge of a system's geometry.

Forward kinematics, computes motion based on a starting point and parameters of the system. An example of forward kinematics is computing the motion of a robotic arm with two joints, a shoulder joint and an elbow joint (each with 1 degree of freedom). The starting point would be the starting configuration of the arm and the algorithm would compute the motion of the arm based on the angular displacement of the joints.

Inverse kinematics computes the opposite. Considering the same example an inverse kinematics algorithm would compute the necessary angular displacement of each joint to make the robotic arm reach a desired point in space. These algorithms are often used in games e.g. to procedurally animate a character's hands and torso to make them follow an object like a gun or a sword

Rigid body dynamics algorithms are generally more complicated except for trivial cases because they are physically based and have to take the various forces and torques acting upon bodies into consideration.

Forward dynamics computes the motion resulting from applying specified forces to a system in a starting position. In our robotic arm example, this means that we would compute the movement of the arm from its starting configuration by exerting specified forces on the arm through e.g. a gravitational force, torques of the motors in the arm's joints etc.

Inverse dynamics is the last category within this type of classification of rigid body simulation methods. It computes the forces that must be imposed on the bodies in the system to reach a desired configuration. In the example of the robotic arm, this would mean calculating the torques that the motors in the joints must exert in order to move the robotic arm into a target position.

Featherstone's algorithm is a forward dynamics algorithm as it computes the motion of bodies connected with joints based on the forces acting upon them.

2.1.2 Direct vs. Iterative Methods

Another major way rigid body simulation methods can be classified is depending on what numerical method they use. The early simulation methods often used *direct methods* as they are accurate, but their computational cost does not scale well with the number of simulated bodies. With the increasing demand for interactive simulations *iterative methods* became popular and are often found in contemporary game engines.

Among the commonly used iterative methods are Gauss-Seidel type methods which have a linear convergence rate. An example of this type of method is the Project Gauss-Seidel (PGS) method. Another type of iterative methods are the Newton-type algorithms which can provide up to a quadratic convergence rate alleviating the visual artefacts caused by the linear convergence rate of Gauss-Seidel type methods. A disadvantage of the Newton-type iterative methods is that its per iteration computational cost is higher than the one of Gauss-Seidel iterative methods.

Even though iterative methods are computationally less demanding and scale well with an increasing number of bodies they can be unstable when dealing with extreme scenarios like high mass ratios (big differences in mass of the simulated bodies). In such specialised cases, or when a very high accuracy is needed direct methods are preferred.

In the reviewed literature Featherstone’s algorithm is not typically classified within these categories. Even though it does iteratively approximate the motion of an articulated body through numeric integration, but in literature it is typically characterised by its other qualities such as using a reduced-coordinate formulation and being recursive [5].

2.1.3 Maximal Coordinate vs. Reduced Coordinate Formulation

When it comes to classifying methods for simulating Articulated bodies i.e. structures of rigid bodies connected by joints the existing methods can be divided into two groups. The first group is called *maximal coordinate* and the second *reduced coordinate* formulations. The reduced coordinate formulations are also referred to as *generalised coordinate* or *minimal coordinate* formulations depending on the author. The main distinction between the two types of methods is the formulations of the equations of motion. Maximal coordinate formulations simulate rigid bodies with all of their 6 degrees of freedom. Reduced coordinate formulations exploit the knowledge of constraints in the modelled system and reduce the number of variables in the equations of motion. This reduction in the degrees of freedom decreases the computational complexity of the simulation.

Examples of maximal coordinate formulation methods are the *penalty force* method, *Lagrange multiplier* method and the *impulse-based* method. The penalty force method uses the addition of external repulsive forces to the rigid bodies to reduce the violation of constraints such as non-penetration constraints. This method is fast and easy to implement, but does allow for visual artefacts such as joint drift or interpenetration of objects into each other. A more accurate method is the *Lagrange multiplier* method which prevents the violation of constraints. A well-known Lagrange multiplier method in computer graphics was described by David Baraff [4]. This method can simulate articulated bodies without loops in linear time same as Featherstone’s algorithm. The Lagrange multiplier method is able to prevent the violation of constraints due to external forces. Unfortunately, it is not able to prevent the violation of constraints due to other reasons such as numerical errors. Therefore, numerical errors build up over time and lead to e.g. the breaking of the joints in the system which is not realistic and is a major difference from the Featherstone’s algorithm. The last above mentioned method is the impulse-based method which is similar to the Lagrange multiplier method but it enforces constraints by computing corrective impulses by predicting the future state of the simulation not only the current state [5].

The most well-known reduced coordinate method for simulating tree like articulated bodies without loops is the *Articulated Body Algorithm* also known as Featherstone’s algorithm. Instead of keeping track of all 6 degrees of freedom per rigid body in the simulation it reduces the necessary parameters to only one per joint in the articulated body. Therefore, the exact configuration of the articulated body e.g. a robotic arm can be precisely describe by using only the angular displacement of each joint (assuming that all joints are revolute i.e. hinge joints). This algorithm is computable in linear time with respect to the number of bodies in the articulated body. It also does not suffer from joint drift which makes it a perfect fit for robotics simulations which demand demand higher accuracy or even the simulation of ragdolls in games who’s limbs can move with high speeds (usually causing

visual artefacts if simulated with other methods). One of the best available explanations of this algorithm without excessive mathematical abstraction is Mirtich’s PhD thesis [1].

2.2 Collision Detection

This section briefly introduces the concept of collision detection and describes its role in a rigid body simulation. Despite the fact that collision detection and collision response were not implemented for Featherstone’s algorithm in this thesis it is introduced here for completeness and to serve as a starting point for future work. The specific details of implementing collision response through an impulse-based method for Featherstone’s algorithm are described by Mirtich in chapter 5 of his PhD thesis[1].

An important part of rigid body simulations is the interaction of rigid bodies when they come into contact with each other. Collision detection enables the simulation of these interactions by identifying the contact points between bodies in each step of the simulation. To better understand how collision detection interacts with the rest of the simulation see the diagram of a typical simulation loop in figure 2.1. The first step in the simulation loop is computing the new positions using the chosen simulation method or initialising the positions of the bodies to predefined starting values. Next, the collisions between bodies are identified and this data is passed to a collision response algorithm. The collision response algorithm typically computes some additional external forces that are considered in the equations of motion when computing the new positions for the next simulation step.

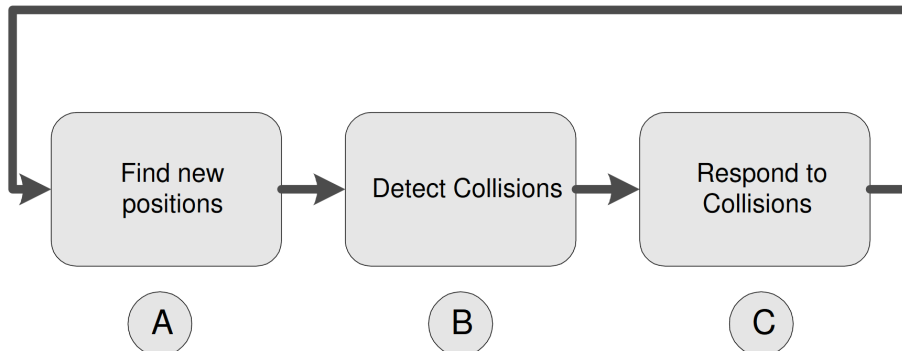


Figure 2.1: A typical simulation loop [8].

Collision detection often becomes the bottleneck of a simulation especially when simulating a large amount of bodies with complex geometry. The problem of collision detection appears to be a $O(n^2)$ problem, meaning that every triangle has to be checked against every other triangle in the scene. This would be incredibly computationally expensive and one of the first ones to address this issue was Hubbard [10] who introduced the concept of a *broad-phase* and *narrow-phase* collision detection. In the broad phase, the pairs of bodies that could be colliding are identified and the rest of the bodies are „pruned“ i.e. not considered in the following phases of collision detection. The list of possibly colliding pairs of bodies is used in the narrow phase to compute if the pair of bodies is separated, touching or penetrating. After the narrow phase follows the contact determination phase which computes the exact contact points which serve as an input for the collision response algorithm. The order of the three phases is visualised in figure 2.2.

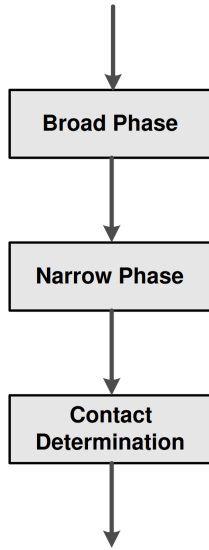


Figure 2.2: The phases of collision detection [8].

2.2.1 Broad-Phase

The goal of this phase is to determine which pairs of objects are likely to collide and exclude the rest from further collision detection computations. This phase dramatically reduces the computational cost of collision detection for scenes with many bodies. The broad phase uses the following four key principles to achieve this. The approximation principle is based on replacing each object with a simpler shape that contains it such as a sphere, cylinder, oriented bounding box or most commonly an axis-aligned bounding box (AABB). The locality principle is built on the idea of not testing pairs of objects for collisions if they are sufficiently far away from each other. Methods such as coordinate sorting and gridding are used to determine which objects are sufficiently far away from each other. Another principle used in the broad phase is the coherence principle which evaluates how coherent or smoothly changing the simulation is. In the case of highly coherent simulations, it is possible to e.g. reuse previously computed values because the simulation state has likely not changed much since the last simulation step. The last principle is the kinematics principle. This principle uses the data about the motion of objects to predict their future collisions and prevent unnecessary future collision checks before the estimated time of impact.

The simplest example of a broad phase algorithm is the *exhaustive search* algorithm which performs a brute-force pairwise check between all possible pairs of objects and checks for collisions. Therefore, it has a running time of $O(n^2)$. A much better algorithm with a linear $O(n)$ expected running time is the *sweep and prune* algorithm, also called *coordinate sorting*. It works with AABBs and achieves a linear running time if the endpoints of AABBs are sorted in increasing order.

As putting a single bounding box around each object can hide too many of the details away, spatial data structures such as *spatial subdivision* and *bounding volume hierarchies* (BVH) can be introduced. They break each object down into multiple objects of a slightly higher detail at every level of the data structure [8].

2.2.2 Narrow-Phase

During the narrow phase, the list of likely colliding pairs of objects is investigated in further detail. Each pair of objects is evaluated if the objects are separated, touching or penetrating. Most narrow-phase algorithms return much more detailed information such as separation distance, penetration depth, closest points etc. This information is then used for an exact contact determination in the next phase. There are many different narrow-phase algorithms divided into several main groups. *Gilbert–Johnson–Keerthi* (GJK) is a famous simplex-based type of algorithm to mention an example of one [8].

2.2.3 Contact Determination

Contact point is a touching point between the surfaces of two objects. A *contact plane* is a linear approximation of the two surfaces at the point of contact and its normal is called the *contact normal*. At the point of contact between two smooth surfaces, there are two coinciding tangent planes, but in computer simulations of polygonal models determining a single point of contact is not that straightforward. The contact point is usually computed from two features, one from each polyhedron as the closest point between them. A pair of two features used for determining a contact point is called a *principal contact*. There are three types of features: a vertex, an edge and a face. The combinations of these features form six different principal contacts. Each combination of features has a dedicated method of computing the closest point between these features. The interested reader can read about the specific ways of computing the contact point in Erleben's book *Physics-Based Animation* [8]. A special case of a contact point is touching contact where the closest point between the two bodies should be exactly zero. It is known that due to the numerical imprecision of floating point numbers, the distance will never be exactly zero. Therefore, to counter this problem a *collision envelope* is used in practice. This is a slightly larger version of the object at the same position as the original object. After subtracting the original object from it, we are left with a field which can be used for detecting touching contact with other objects as seen in figure 2.3.

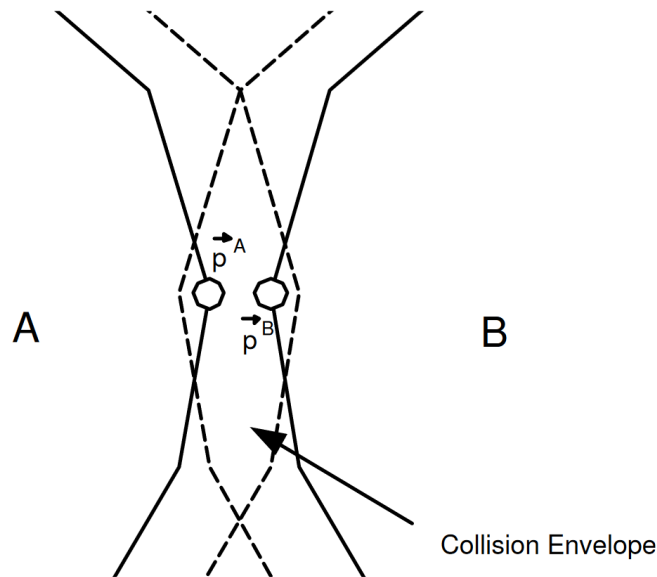


Figure 2.3: The collision envelope used for determination of touching contact [8].

Chapter 3

Featherstone's Algorithm

Featherstone's algorithm is a specialised rigid body dynamics algorithm for computing the forward dynamics of a kinematic tree. Meaning, that the user specifies the torques of the motors in the joints and the starting position of e.g. a robotic arm and the algorithm simulates its movement. Featherstone's algorithm is interesting because it has linear time complexity with respect to the number of bodies in the kinematic tree. Due to this fact, its performance scales well with an increasing number of rigid bodies in the chain. On top of this, its usage of a reduced coordinate system guarantees zero joint error. As a consequence of zero joint error, the simulation is more accurate as the attachment points of the joint on both rigid bodies are always aligned as they would be on a real system. This chapter is mostly based on Roy Featherstone's book „Rigid Body Dynamics Algorithms“ [3] and Brian Vincent Mirtich's PhD thesis called „Impulse-based Dynamic Simulation of Rigid Body Systems“ [1]. Mirtich describes Featherstone's algorithm in a more approachable way than Featherstone's book and provides pseudo-code throughout his thesis. Along with Mirtich's thesis an implementation of a simulator *Impulse* has been developed, which was not publicly available at the time of writing of this master's thesis.

This chapter aims to describe Featherstone's algorithm at the level of a MSc. in computer science student. Within this chapter, the focus is on explaining the underlying principles on simple examples as well as providing all details necessary for the implementation of Featherstone's algorithm. The algorithm computes each simulation step in three passes. It can be used to simulate kinematic chains or kinematic trees. In section 3.3 the link and joint indexing conventions used within this thesis are introduced. The following sections 3.4, 3.5, 3.6 describe the three passes of the algorithm for a kinematic chain. The necessary modifications to the algorithm to expand it to kinematic trees and a floating base can be found in Mirtich's thesis [1].

3.1 Used Terminology

Different publications use slightly different terminology therefore this section aims to clarify the terms used within this paper.

Link is a rigid body that represents a segment of the kinematic chain. Individual links are connected together by joints. In some literature, links are also referred to as bodies. Within this thesis the *current link* refers to link i and the *previous link* refers to link $i - 1$.

Joint connects two links together and constrains their movement. This thesis only considers the usage of *prismatic* joints (also called *revolute* joints) that have one degree of freedom (dof).

Kinematic chain is an arrangement of links (rigid bodies) connected to each other by joints where each link is connected to exactly two other links except for the ends of the chain. The kinematic chains discussed within this thesis have a grounded base link that is static and serves as the inertial reference frame. The number of degrees of freedom of such a chain is equal to its number of joints.

Inboard joint or link refers to the one closer to the base of the kinematic chain.

Outboard joint or link refers to the one closer to the end of the kinematic chain.

Kinematic tree has a more complex hierarchy than a kinematic chain as each link can be attached to multiple other links by joints. The kinematic trees considered within this thesis are ones with a static base link.

Articulated body in a kinematic chain refers to a link called the handle and its sub-chain treated as a single rigid body. An articulated body in a kinematic tree refers to a link (handle) and its sub-tree treated as a single rigid body.

Dynamic state of a kinematic chain or a kinematic tree in this thesis refers to the scalar joint positions q_i and scalar joint velocities \dot{q}_i of the joints within the kinematic structure. This definition is adopted from Mirtich's thesis [1].

Spatial isolated zero-acceleration (z.a.) force also called the bias force in Featherstone's book [3] is the force which must be exerted by the inboard joint of a link to prevent the link from accelerating. The adjective *spatial* indicates that the force vector is six-dimensional and encapsulates both the force (linear component) and the torque (angular component).

Spatial articulated zero-acceleration (z.a.) force is the force which must be exerted by the inboard joint of a link (handle) to prevent it and its sub-chain from accelerating. The adjective *articulated* indicates that the entire sub-chain beginning at the handle is being considered.

Spatial isolated inertia $\hat{\mathbf{I}}_i$ of link i is a six-dimensional (spatial) matrix that contains the mass matrix and the inertia tensor of the link.

$$\hat{\mathbf{I}}_i = \begin{bmatrix} 0 & M_i \\ \hat{\mathbf{I}}_i & 0 \end{bmatrix} \quad (3.1)$$

Spatial articulated inertia $\hat{\mathbf{I}}_i^A$ is a six-dimensional matrix representing the spatial inertia of a handle and its sub-chain in a kinematic chain treated as a single rigid body.

Inertial reference frame, also known as an inertial frame of reference and often referred to as simply a *frame* is a coordinate system within which the laws of physics hold true. An inertial referenced frame is an idealised concept within which we assume Newton's laws of motion to hold ideally. Every link i has a frame \mathcal{F}_i attached to it.

3.2 Table of Symbols

The table below lists symbols used within this chapter with their short descriptions and whether the specific quantity is a scalar, vector or matrix. I provide this table as I wish I had this when I started implementing Featherstone's algorithm as it would make it easier for me to understand the pseudo-code in Mirtich's thesis and decide on a datatype for the listed quantities.

Symbol	Description	Type
i	link and joint index i	$i \in \mathbb{Z}, i \geq 0$
m	scalar mass of link i	$m \in \mathbb{R}$
$q_i, \dot{q}_i, \ddot{q}_i$	scalar joint position, velocity, acceleration of joint i	$q_i, \dot{q}_i, \ddot{q}_i \in \mathbb{R}$
Q_i	scalar joint actuator force/torque of link i	$Q_i \in \mathbb{R}$
\mathbf{v}_i	linear velocity of link i	3x1 vector
\mathbf{a}_i	linear acceleration of link i	3x1 vector
$\boldsymbol{\omega}_i$	angular velocity of link i	3x1 vector
$\boldsymbol{\alpha}_i$	angular acceleration of link i	3x1 vector
\mathbf{g}	gravitational acceleration	3x1 vector
\mathbf{u}_i	axis vector of joint i (unit vector in the direction of the axis)	3x1 vector
$\boldsymbol{\nu}_i$	vector velocity of joint i ($\boldsymbol{\nu}_i = \dot{q}_i \mathbf{u}_i$)	3x1 vector
\mathbf{r}	radius vector	3x1 vector
\mathbf{d}_i	vector from the axis of joint i to the origin of \mathcal{F}_i	3x1 vector
\mathcal{F}_i	inertial reference frame of link i	3x3 matrix
\mathbf{R}	rotation matrix from \mathcal{F}_{i-1} to \mathcal{F}_i	3x3 matrix
\mathbf{I}_i	inertia tensor of link i	3x3 matrix
\mathbf{M}_i	matricized mass of link i (has mass on the diagonal)	3x3 matrix
$\hat{\mathbf{c}}_i$	spatial Coriolis vector	6x1 vector
$\hat{\mathbf{Z}}_i^A$	spatial articulated zero-acceleration force of link i	6x1 vector
$\hat{\mathbf{I}}_i^A$	spatial articulated inertia of link i	6x1 vector
$\hat{\mathbf{s}}_i$	spatial joint axis of link i	6x1 vector
$\hat{\mathbf{a}}_i$	spatial acceleration of link i	6x1 vector
${}_{\mathcal{G}}\hat{\mathbf{X}}_{\mathcal{F}}$	spatial transformation from frame \mathcal{F} to \mathcal{G}	6x6 matrix

Table 3.1: List of symbols used in this chapter.

3.3 Link and Joint Indexing

Featherstone's algorithm can compute forward dynamics for kinematic chains and kinematics trees without loops. This thesis adopts the indexing convention of Mirtich's PhD thesis where the index of each joint and link is an integer from 0 to n . The grounded root link that acts as the reference inertial frame is assigned the index 0. The indexing of the rest of the links and the joints is defined separately for kinematic chains and kinematic trees.

3.3.1 Kinematic Chains

The root of the kinematic chain is assigned the index 0 and every subsequent link has an index one higher than the previous link in the chain. Every joint in a kinematic chain has the same index as its outboard link as seen the figure 3.1.

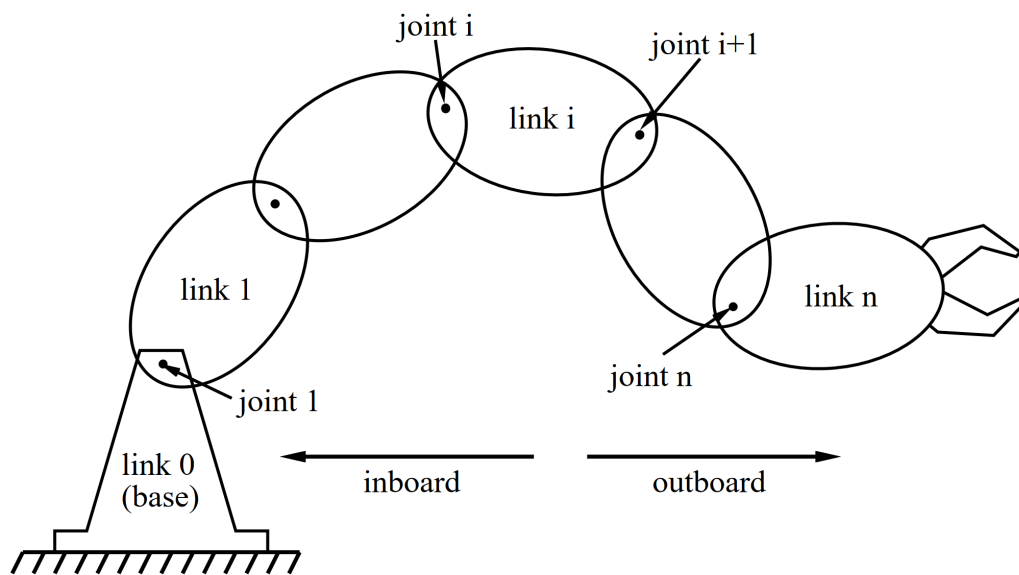


Figure 3.1: Link and joint indexing convention for kinematic chains [1]

3.3.2 Kinematic Trees

The indexing of joints for kinematic trees is the same as for kinematic chains where each joint has the same index as its outboard link. The links are indexed by using the algorithm from figure 3.2.

```
int numberLinks(tree, idx)

    r ← root link of tree
    r.idx ← idx
    idx ← idx + 1
    for each child c of r
        idx = numberLinks(c, idx)
    return idx
```

Figure 3.2: Kinematic tree link indexing algorithm [1].

The indexed tree in figure 3.3 can be obtained by running the algorithm 3.2 with the first moving link as the value of the „tree“ argument and the index 1 as the value of the „idx“ argument. The indexing algorithm is recursive and it explores the tree the same way as a DFS (depth-first search) algorithm would. The algorithm returns an index that is one higher than the highest index of a link in the tree.

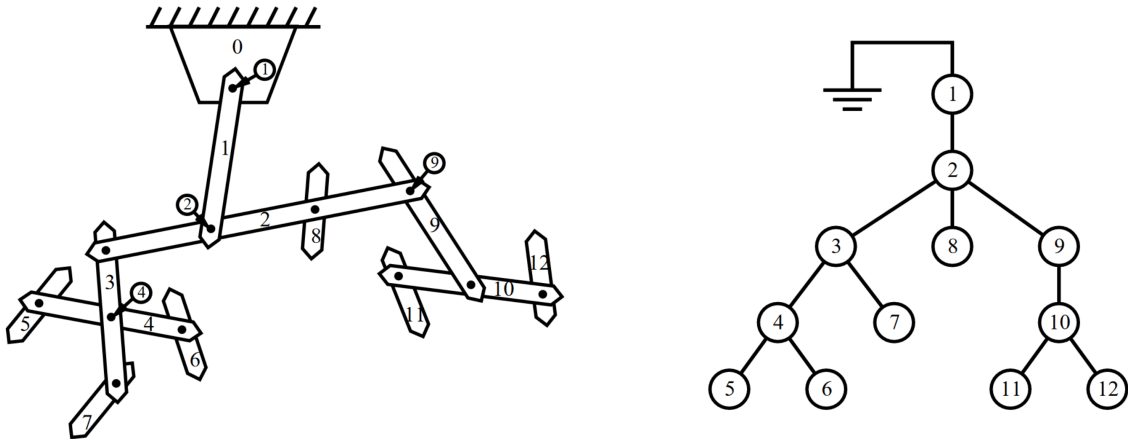


Figure 3.3: Link and joint indexing convention for kinematic trees [1].

3.4 Pass 1

The first pass of the algorithm is an outward pass from the base link to the tip of the chain that calculates the linear velocity v_i , angular velocity ω_i , the articulated zero acceleration force \hat{Z}_i^A , the spatial articulated inertia \hat{I}_i^A and the spatial Coriolis force \hat{c}_i of each link. Visualisation of a traversal of a kinematic chain during the first pass is shown in Figure 3.4.

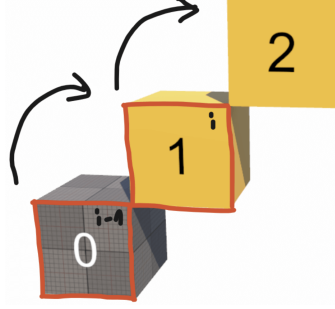


Figure 3.4: Example of the first step of the traversal of a kinematic chain during Pass 1.

Figure 3.5 contains the pseudo-code for computing the first pass. It results from the combination of Figures 4.4 and 4.7 in Mirtich's thesis. These figures have been combined to organise the algorithm into three passes just as Featherstone does in his book [3].

```

 $\omega_0, \mathbf{v}_0, \alpha_0, \mathbf{a}_0 \leftarrow \mathbf{0}$ 

for  $i = 1$  to  $n$ 
     $\mathbf{R} \leftarrow$  rotation matrix from  $\mathcal{F}_{i-1}$  to  $\mathcal{F}_i$ 
     $\mathbf{r} \leftarrow$  radius vector from  $\mathcal{F}_{i-1}$  to  $\mathcal{F}_i$  (in  $\mathcal{F}_i$  coordinates)
     $\omega_i \leftarrow \mathbf{R}\omega_{i-1}$ 
     $\mathbf{v}_i \leftarrow \mathbf{R}\mathbf{v}_{i-1} + \omega_i \times \mathbf{r}$ 
    if joint  $i$  is prismatic
         $\mathbf{v}_i \leftarrow \mathbf{v}_i + \dot{q}_i \mathbf{u}_i$ 
    else /* joint  $i$  is revolute */
         $\omega_i \leftarrow \omega_i + \dot{q}_i \mathbf{u}_i$ 
         $\mathbf{v}_i \leftarrow \mathbf{v}_i + \dot{q}_i (\mathbf{u}_i \times \mathbf{d}_i)$ 

     $\hat{Z}_i^A \leftarrow \begin{bmatrix} -m_i \mathbf{g} \\ \omega_i \times \mathbf{I}_i \omega_i \end{bmatrix}$ 
     $\hat{I}_i^A \leftarrow \begin{bmatrix} 0 & M_i \\ \mathbf{I}_i & 0 \end{bmatrix}$ 

    if joint  $i$  is prismatic  $\hat{c}_i \leftarrow \begin{bmatrix} \mathbf{0} \\ \omega_{i-1} \times (\omega_{i-1} \times \mathbf{r}_i) + 2\omega_{i-1} \times \mathbf{v}_i \end{bmatrix}$ 
    else  $\hat{c}_i \leftarrow \begin{bmatrix} \omega_{i-1} \times \mathbf{v}_i \\ \omega_{i-1} \times (\omega_{i-1} \times \mathbf{r}_i) + 2\omega_{i-1} \times (\mathbf{v}_i \times \mathbf{d}_i) + \mathbf{v}_i \times (\mathbf{v}_i \times \mathbf{d}_i) \end{bmatrix}$ 

```

Figure 3.5: Pseudo-code for computing Pass 1 [1]).

The input to this pass is the dynamic state that fully defines the state of the kinematic chain. The dynamic state consists of the scalar joint position q_i (measured as angular displacement in radians) and the scalar joint velocity \dot{q}_i (measured as angular velocity in radians per second) of each joint. Given that the kinematic chain of length n has $n - 1$ joints. The dynamic state of the system can be fully described with $n - 1$ pairs of scalar values because each joint has only one degree of freedom.

Before we get into the breakdown of what the pseudo-code exactly does let us introduce what a rotation matrix does. A rotation matrix in our case is a 3×3 matrix that represents the orientation of a body in 3D space. The columns of a rotation matrix describe where each of the vectors of a standard basis lands after applying the represented rotation. A standard basis of our 3D vector space is the collection of three unit vectors in the directions of the principal axes of our Euclidean coordinate system. Let's look at the example rotation matrix in Figure 3.6.

$$R = \begin{bmatrix} \hat{i} & \hat{j} & \hat{k} \\ -0.63, & -0.77, & 0 \\ 0.77, & -0.63, & 0 \\ 0, & 0, & 1 \end{bmatrix} \begin{matrix} x \\ y \\ z \end{matrix}$$

Figure 3.6: An example of a rotation matrix. Each column represents where the corresponding basis vector lands after applying the rotation.

The above example rotation matrix represents the rotation seen in Figure 3.7.

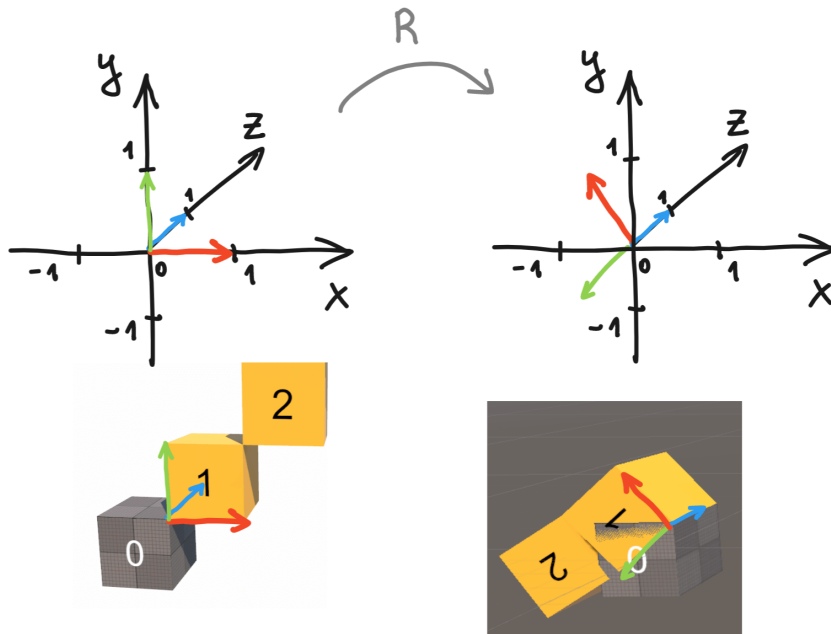


Figure 3.7: An intuitive way of thinking about the effects of a rotation matrix.

Now that we have an intuitive understanding of what a rotation matrix does let us dive into the first line of the for loop in the pseudo-code. The first pass iterates over the links from 1 to n . At first the rotation matrix \mathbf{R} representing the rotation from frame \mathcal{F}_{i-1} attached to the previous link to frame \mathcal{F}_i of the current link has to be calculated. The rotation of each frame can be represented by a 3x3 rotation matrix which transforms vectors from the space of link i to world space. Therefore the rotation matrix from frame \mathcal{F}_{i-1} to frame \mathcal{F}_i can be computed in the following way:

$$\mathbf{R} = \mathbf{R}_i^{-1} \times \mathbf{R}_{i-1} \quad (3.2)$$

Illustrated on our example of a kinematic chain with three links aligned with the axes of the world the result can simply be an identity matrix during the first simulation step as seen in Figure 3.8.

$$\mathbf{R} = \mathbf{R}_i^{-1} \mathbf{R}_{i-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R}_{i-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R}_i^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 3.8: Example of computing \mathbf{R} for the first simulation step of our example kinematic chain shown in Figure 3.4.

Notice what happens when a vector is multiplied with the rotation matrix \mathbf{R} from the right when looking at Figure 3.8. First, the matrix \mathbf{R}_{i-1} transforms the vector from space of link $i - 1$ into world space and by multiplying the result with \mathbf{R}_i^{-1} the vector is brought from world-space into the space of link i .

Next, the radius vector \mathbf{r} spanning from the origin of \mathcal{F}_{i-1} to the origin of \mathcal{F}_i is calculated as described by equation 3.3. The radius vector is expressed in the coordinates of \mathcal{F}_i . Using the position of the previous link p_{i-1} , the position of the current link p_i and the inverse of the rotation matrix of the current link \mathbf{R}_i^{-1} the radius vector is calculated like so:

$$\mathbf{r} = \mathbf{R}_i^{-1} \times (\mathbf{p}_i - \mathbf{p}_{i-1}) \quad (3.3)$$

The difference between the positions of the two links is multiplied by the inverse of the rotation matrix of the current link to bring it from world space to the space of link i . The result of the radius vector in our example during the first simulation step and the first iteration of the for loop is shown in Figure 3.9.

$$\mathbf{r} = \mathbf{R}_i^{-1} (\mathbf{x}_i - \mathbf{x}_{i-1}) = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

Figure 3.9: The result of the radius vector during the first simulation step of our example.

The radius vector is required to be represented in the inertial reference frame \mathcal{F}_i of the current link i in order for the algorithm to compute correct results. The importance of this is shown in figure 3.10. It is visible how in our example the vector is the same in world space and the space of link i during the first step of the simulation but in later steps of the simulation this does not hold anymore.

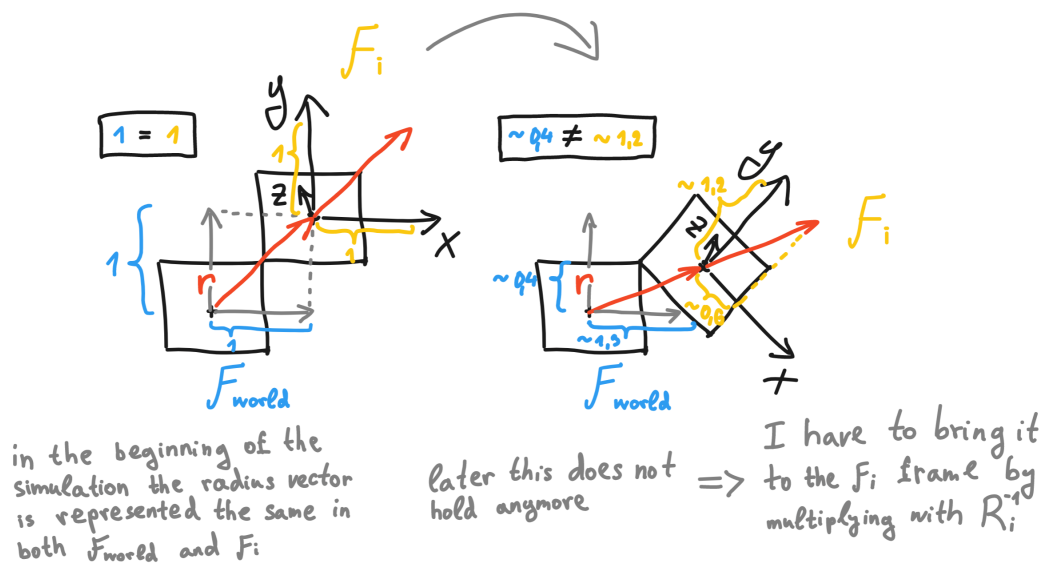


Figure 3.10: Illustration of the reason why the radius vector computation contains a multiplication by R_i^{-1} .

The angular velocity ω_i of the current link is equal to the angular velocity of the previous link rotated from the frame of the previous link to the frame of the current link.

The linear velocity of the current link v_i is the sum of the linear velocity of the previous link rotated into the current frame and the linear velocity resultant from computing the cross product of the angular velocity of the previous link rotated into the current frame and the radius vector r . The rotated angular velocity of the previous link is now the angular velocity of the current link ω_i . This is not obvious at first glance. The computation of the linear velocity resultant from the angular velocity of the previous link is better understood from the diagram in Figure 3.11. The same formula as the one for computing the linear velocity of a rolling tire of a car is used.

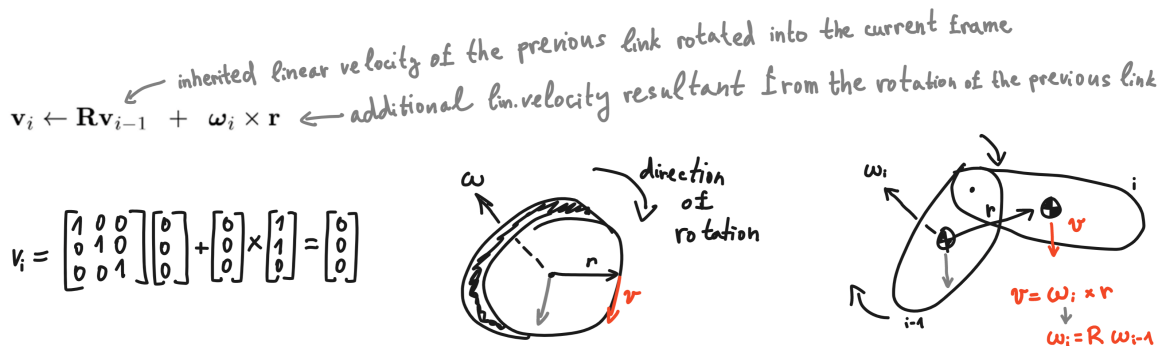


Figure 3.11: Illustration of how the linear velocity of the current link i is computed.

Now the angular and the linear velocity have been transferred from the previous to the current link, but unless the joint has been static the velocity of the joint must also be taken into consideration and counted into the final velocity of the current joint. In the case of a prismatic joint (also called a *slider* joint) the velocity of the joint is simply added onto the linear velocity of the current link in the direction of the axis (direction of sliding) of the joint. In the case of a revolute joint (also called a *hinge* joint), the joint velocity contributes to both the angular and the linear velocity of the current link. The unit vector u_i in the direction of the rotation axis is multiplied by the joint scalar velocity, therefore, it becomes a vector of the angular velocity contributed to the total angular velocity of the current link by joint i .

The velocity contribution of joint i to the velocity of the current link is calculated as the cross product between the axis vector u_i and the vector d_i spanning from joint i to the origin of \mathcal{F}_i expressed in the coordinates of \mathcal{F}_i , the cross product is afterwards scaled by the scalar joint velocity of joint i as depicted in Figure 3.12.

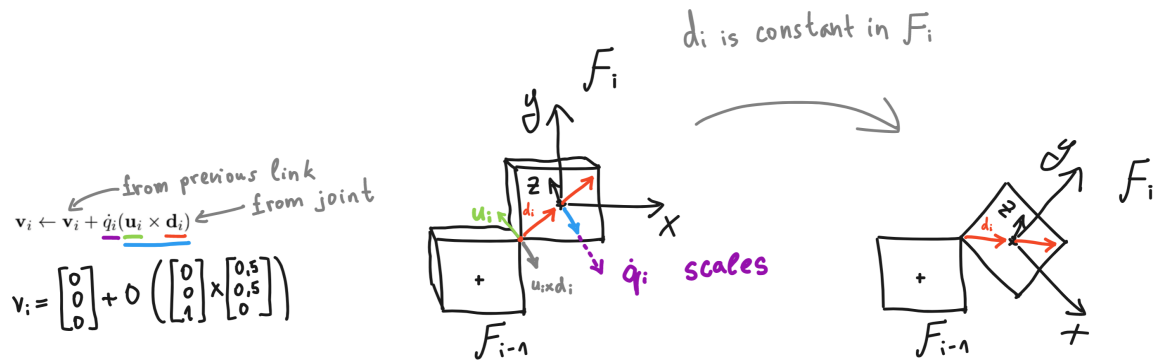


Figure 3.12: Depiction of the contribution of the joint scalar velocity to the final linear velocity of the current link i .

Both the spatial articulated zero-acceleration force \hat{Z}_i^A as well as the spatial articulated inertia \hat{I}_i^A of the current link is initialised to the quantity valid for the current link in isolation despite the adjective „articulated“ in its name. After an addition to this initial value in the second pass, the quantity becomes truly articulated i.e. expressing the value for the current link and its sub-chain. The 3x1 linear component (upper half) of the \hat{Z}_i^A z.a. force 6x1 vector is simply the opposite of the gravitational force acting upon the link to prevent it from linear acceleration. The 3x1 angular component (lower half) of the z.a. force 6x1 vector neutralises any angular acceleration. The lower left block of the \hat{I}_i^A spatial inertia 6x6 matrix is simply the link's inertia tensor and the upper right block of the matrix is the matricized mass of link i (a diagonal matrix containing the link's mass scalar along the main diagonal).

3.5 Pass 2

The second pass is an inward pass from the end of the kinematic chain to its root as seen in Figure 3.13.

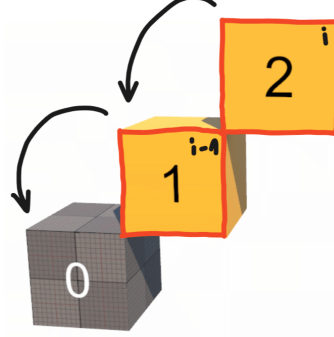


Figure 3.13: Example of the first step of the traversal of a kinematic chain during Pass 2.

During this pass the spatial articulated inertias and zero-acceleration forces of the links in the chain are computed as seen in the pseudo-code in figure 3.14.

$$\begin{array}{l}
 \text{for } i = n \text{ downto } 2 \\
 \hat{\mathbf{I}}_{i-1}^A \leftarrow \hat{\mathbf{I}}_{i-1}^A + {}_{i-1}\hat{\mathbf{X}}_i \left[\hat{\mathbf{I}}_i^A - \frac{\hat{\mathbf{I}}_i^A \hat{\mathbf{s}}_i \hat{\mathbf{s}}_i' \hat{\mathbf{I}}_i^A}{\hat{\mathbf{s}}_i' \hat{\mathbf{I}}_i^A \hat{\mathbf{s}}_i} \right] {}_{i-1}\hat{\mathbf{X}}_{i-1} \\
 \hat{\mathbf{Z}}_{i-1}^A \leftarrow \hat{\mathbf{Z}}_{i-1}^A + {}_{i-1}\hat{\mathbf{X}}_i \left[\hat{\mathbf{Z}}_i^A + \hat{\mathbf{I}}_i^A \hat{\mathbf{c}}_i + \frac{\hat{\mathbf{I}}_i^A \hat{\mathbf{s}}_i [Q_i - \hat{\mathbf{s}}_i' (\hat{\mathbf{Z}}_i^A + \hat{\mathbf{I}}_i^A \hat{\mathbf{c}}_i)]}{\hat{\mathbf{s}}_i' \hat{\mathbf{I}}_i^A \hat{\mathbf{s}}_i} \right]
 \end{array}$$

Figure 3.14: Pseudo-code for computing pass 2 (from figure 4.8 in Mirtich's thesis [1]).

To compute the first equation of the second pass we first need to compute the spatial transformations ${}_{\mathcal{G}}\hat{\mathbf{X}}_{\mathcal{F}}$, the articulated body inertia $\hat{\mathbf{I}}_i^A$ of the current link and the spatial joint axis $\hat{\mathbf{s}}_i$ of joint i . First, see the definition of a spatial transformation shown in Figure 3.15.

Definition 6 Let \mathcal{F} and \mathcal{G} be two frames, let \mathbf{r} be the offset vector from the origin of \mathcal{F} to the origin of \mathcal{G} (expressed in \mathcal{G} 's coordinates), and let \mathbf{R} be the 3×3 rotation matrix transforming (non-spatial) vectors from \mathcal{F} to \mathcal{G} . Then the 6×6 spatial transformation matrix from \mathcal{F} to \mathcal{G} is given by

$${}_{\mathcal{G}}\hat{\mathbf{X}}_{\mathcal{F}} = \begin{bmatrix} \mathbf{1} & \mathbf{0} \\ -\tilde{\mathbf{r}} & \mathbf{1} \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{0} \\ \mathbf{0} & \mathbf{R} \end{bmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{0} \\ -\tilde{\mathbf{r}}\mathbf{R} & \mathbf{R} \end{bmatrix}.$$

With this notation, the spatial velocity transformation shown above can be written

$$\hat{\mathbf{v}}_{\mathcal{G}} = {}_{\mathcal{G}}\hat{\mathbf{X}}_{\mathcal{F}} \hat{\mathbf{v}}_{\mathcal{F}}.$$

Figure 3.15: Definition of a spatial transformation from Mirtich's thesis [1].

In our example, a concrete computed spatial transformation matrix from the current to the previous frame is computed in Figure 3.16.

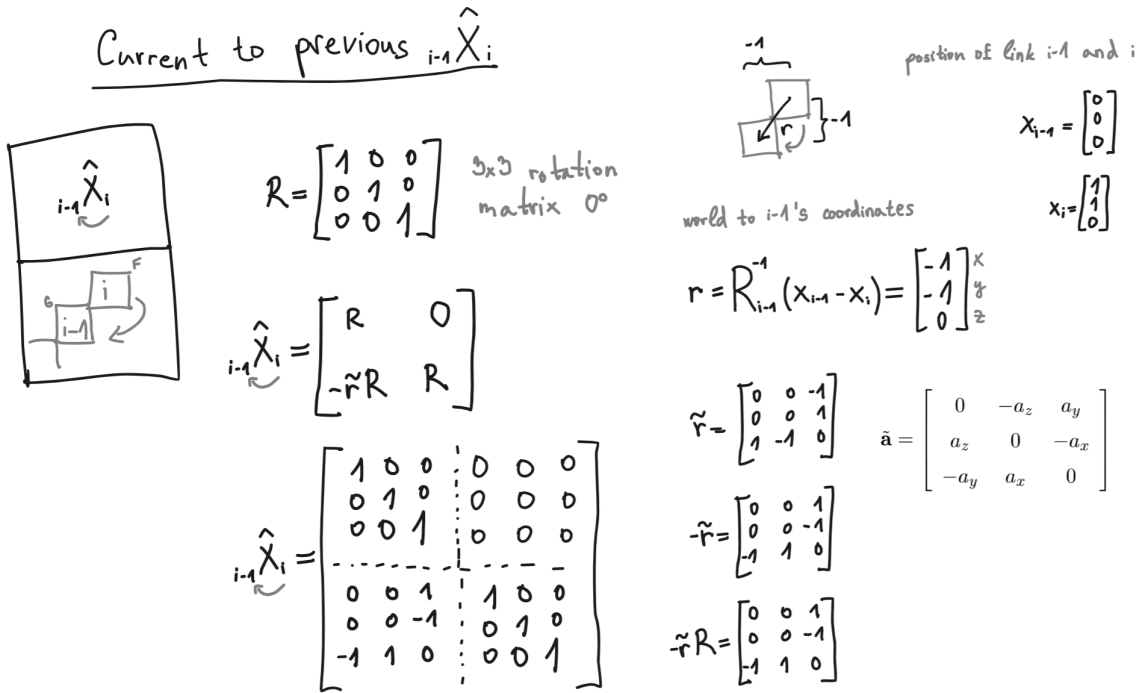


Figure 3.16: A computed spatial transformation from the current to the previous frame.

An example of the second required spatial transformation, one from the previous to the current frame is computed in Figure 3.17.

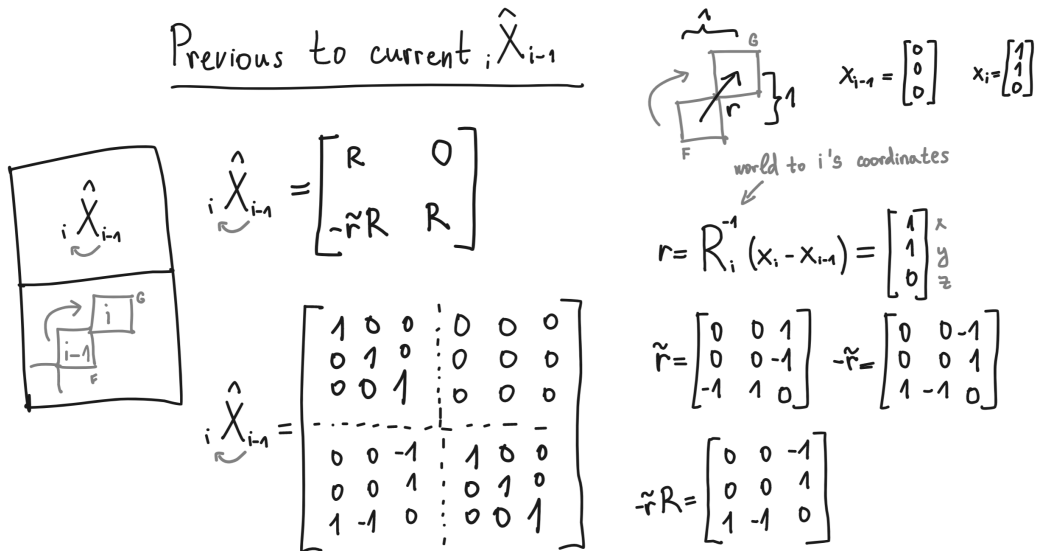


Figure 3.17: A computed spatial transformation from the previous to the current frame.

Next, to compute the spatial articulated inertia of the previous link we will need the spatial articulated inertia $\hat{\mathbf{I}}_i^A$ (it is actually still the isolated inertia of the current link computed in the first pass) which can be computed as seen in Figure 3.18.

Spatial articulated inertia of link i computed in Pass 1
Spatial joint axis of joint i

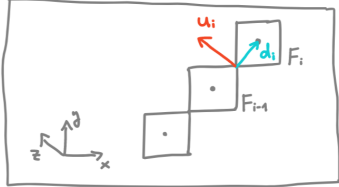
$$\hat{\mathbf{I}}_i^A = \begin{bmatrix} 0 & M_i \\ \hline I_i & 0 \end{bmatrix}$$

$$\hat{\mathbf{I}}_i^A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 0,16 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0,16 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0,16 & 0 & 0 & 0 \end{bmatrix}$$

$$\hat{\mathbf{S}}_i = \begin{bmatrix} u_i \\ u_i \times d_i \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ -0,5 \\ 0,5 \\ 0 \end{bmatrix}$$

$$u_i \times d_i = \begin{bmatrix} -0,5 \\ 0,5 \\ 0 \end{bmatrix}$$

$$u_i = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad d_i = \begin{bmatrix} 0,5 \\ 0,5 \\ 0 \end{bmatrix}$$

$$\hat{\mathbf{S}}_i = \begin{bmatrix} a \\ b \end{bmatrix}$$


$$\hat{\mathbf{S}}_i = \begin{bmatrix} b^T & a^T \end{bmatrix} = \begin{bmatrix} -0,5 & 0,5 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 3.18: A computed example of a spatial isolated inertia of the current link.

In order to compute the zero acceleration force of the previous link we will need the zero acceleration force computed in the first pass as seen in Figure 3.19.

Zero acceleration force $\hat{\mathbf{Z}}_i^A$

$$-m_i = -1 \quad \omega_i = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \text{ Initialized to zero}$$

$$g = \begin{bmatrix} 0 \\ -9,81 \\ 0 \end{bmatrix}$$

$$I_i = \begin{bmatrix} 0,16 & 0 & 0 \\ 0 & 0,16 & 0 \\ 0 & 0 & 0,16 \end{bmatrix}$$

For a cube of mass 1

$$\hat{\mathbf{Z}}_i = \begin{bmatrix} -m_i g \\ \omega_i \times I_i \omega_i \end{bmatrix} = \begin{bmatrix} -1 \begin{bmatrix} 0 \\ -9,81 \\ 0 \end{bmatrix} \\ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \times \begin{bmatrix} 0,16 & 0 & 0 \\ 0 & 0,16 & 0 \\ 0 & 0 & 0,16 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 0 \\ 9,81 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\hat{\mathbf{Z}}_i = \begin{bmatrix} -m_i g \\ \omega_i \times I_i \omega_i \end{bmatrix}$$

Figure 3.19: A computed example of a zero-acceleration force of the current link.

Now, having computed all of the above quantities we can use the results to compute the spatial articulated inertia and the spatial zero-acceleration force of the previous link in the chain. The dimensions of the matrices computed as intermediate results are shown in Figure 3.20.

$$\hat{\mathbf{I}}_{i-1}^A \leftarrow \hat{\mathbf{I}}_{i-1}^A + {}_{i-1}\hat{\mathbf{X}}_i \left[\hat{\mathbf{I}}_i^A - \frac{{}_i\hat{\mathbf{s}}_i^A \hat{\mathbf{s}}_i^A \hat{\mathbf{I}}_i^A}{{}_i\hat{\mathbf{s}}_i^A \hat{\mathbf{I}}_i^A \hat{\mathbf{s}}_i^A} \right] {}_i\hat{\mathbf{X}}_{i-1}$$

$\begin{matrix} 6 \times 6 & + & 6 \times 6 & \times & \underbrace{6 \times 6} & \times & 6 \times 6 \\ \downarrow & & \downarrow & & \downarrow & & \downarrow \\ \hat{\mathbf{I}}_{i-1}^A & & \hat{\mathbf{I}}_{i-1}^A & & \left[\hat{\mathbf{I}}_i^A - \frac{{}_i\hat{\mathbf{s}}_i^A \hat{\mathbf{s}}_i^A \hat{\mathbf{I}}_i^A}{{}_i\hat{\mathbf{s}}_i^A \hat{\mathbf{I}}_i^A \hat{\mathbf{s}}_i^A} \right] & & {}_i\hat{\mathbf{X}}_{i-1} \end{matrix}$

Figure 3.20: The dimensions of the matrices computed as intermediate results.

3.6 Pass 3

The third pass iterates over the links outwards, from the root to the end of the chain similar to the first pass as seen in Figure 3.21.

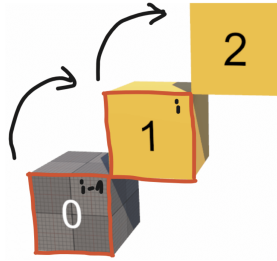


Figure 3.21: Example of the first step of the traversal of a kinematic chain during Pass 1.

The third pass computes the joint accelerations and the link spatial accelerations. The pseudo-code for performing this computation is depicted in figure 3.22.

$$\hat{\mathbf{a}}_0 \leftarrow \hat{\mathbf{0}}$$

for $i = 1$ to n

$$\ddot{q}_i = \frac{Q_i - \hat{\mathbf{s}}_i^A \hat{\mathbf{I}}_i^A \hat{\mathbf{X}}_{i-1} \hat{\mathbf{a}}_{i-1} - \hat{\mathbf{s}}_i^A (\hat{\mathbf{Z}}_i^A + \hat{\mathbf{I}}_i^A \hat{\mathbf{c}}_i)}{\hat{\mathbf{s}}_i^A \hat{\mathbf{I}}_i^A \hat{\mathbf{s}}_i}$$

$$\hat{\mathbf{a}}_i = {}_i\hat{\mathbf{X}}_{i-1} \hat{\mathbf{a}}_{i-1} + \hat{\mathbf{c}}_i + \ddot{q}_i \hat{\mathbf{s}}_i$$

Figure 3.22: Pseudo-code for computing pass 3 (from figure 4.8 in Mirtich's thesis [1]).

All the necessary quantities for computing the link and joint accelerations have already been computed in the previous passes and are only combined together in the third pass.

There is one extra step that needs to be done, and that is the numerical integration of the scalar joint accelerations of all the joints to obtain the scalar joint velocities. Thereafter, one more round of integration is done to obtain the scalar joint positions in radians which can then be directly used to compute the new link positions and orientations.

Mirtich used a Runge-Kutta numerical integration method with an adaptive step size in his simulator *Impulse* [1]. The data-oriented implementation developed in this thesis uses a semi-implicit Euler's method with a fixed step size because the reference implementation of PhysX used for comparison uses this method. The same integration method was used to ensure a fair comparison.

Chapter 4

Data-Oriented Design

Data-oriented design (DOD) can be viewed as a programming paradigm that focuses on efficient data transformations i.e. storing, accessing and processing data in ways that maximise performance. The resources on this topic are limited compared to other more widely adopted programming paradigms such as procedural programming, object-oriented design (OOD), functional programming, logic programming etc. Data-oriented design is often positioned in contrast to object-oriented design, which comes from the fact that DOD-focused solutions often look different from a typical OOD solution, but these two paradigms are not necessarily mutually exclusive. A program can be written in an object-oriented language but still be optimised for performance by keeping the transformations of data and the target hardware in mind, such a program can be considered a data-oriented as well as an object-oriented one. DOD is not an alternative to other programming paradigms, it is rather an overarching philosophy of programming. Even though, Fabian [9] points out that in some programming languages such as the declarative logic programming language *Prolog* that don't give the programmer control over „how“ things are done but enable only declaring „what“ should be done, it can be difficult or impossible to apply DOD.

The concept of data-oriented programming is not new, it has been around for decades in one form or another. Programming is in its essence about storing, loading and transforming data, even instructions are data occupying space in memory. The data-oriented design was officially given its name by Noel Llopis in his September 2009 article [11] and it was popularised by Mike Acton's talk at the CppCon 2014 conference [13].

This chapter begins by motivating the use of data-oriented design by explaining the discrepancy between the computation speed of modern CPU's and the speed of loading data from main memory. Thereafter follows an overview of the guiding principles of data-oriented design based on the available literature and other resources such as conference talks. The next section describes different implementations of the commonly used data-oriented *Entity Component System* (ECS) framework. The final section of this chapter introduces the Data-Oriented Technology Stack (DOTS) that is part of the Unity engine as it is used within this thesis for the implementation of Featherstone's algorithm.

4.1 Motivation

Over the last decades the performance of processors has increased by multiple orders of magnitude through improving parameters such as higher clock frequency (higher amount of processing cycles per second), higher transistor count and an increasing number of CPU

cores. On the other hand, the improvement of data access speeds on dynamic random access memories (DRAM) used for main memory in modern computers has been dramatically smaller as seen in figure 4.1. Therefore, in modern systems the performance bottleneck is usually the speed of access of data from main memory.

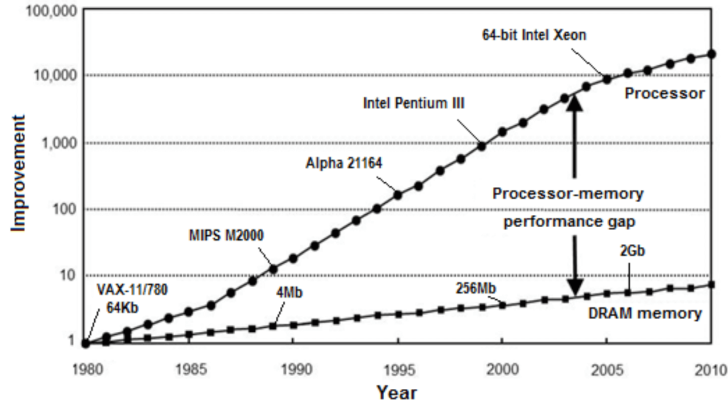


Figure 4.1: The increasing performance gap between processor speed and DRAM memory data access time, over the time period of three decades. The speeds of CPU's have been evaluated by using the standard SPECint benchmark programs [7].

Even though the technology for manufacturing fast static RAM (*SRAM*) is known to humanity, it is simply way too expensive to manufacture compared to the slower and much cheaper dynamic RAM (*DRAM*). Due to this stark difference in price, the *hierarchical memory model* was born. The hierarchical memory model is based on the assumption that memory is often not used completely randomly and there are certain data access patterns. Therefore the hierarchical model includes small amounts of fast *SRAM cache* memory used to store data that is used most frequently and the rest of the data is stored on the cheaper but slower *DRAM* (main memory). In fact, in contemporary systems, there are often multiple layers of cache memory called layer one (L1) cache (typically located directly on the chip of the CPU), layer two (L2) cache etc. As a rule of thumb, the closer to the CPU a memory lies within the hierarchical memory model the faster, smaller and more expensive it gets as seen in the figure 4.2.

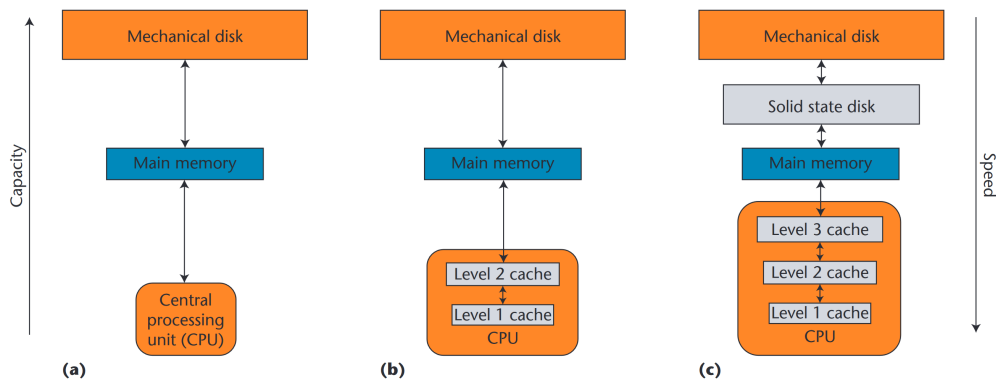


Figure 4.2: Visualisation of the relationship between capacity and speed (also often price per megabyte) in the hierarchical memory model. The sub-figures (a) to (c) show the evolution of memory architectures over time. [2].

The hierarchy of cache memories is designed to reduce the impact of the low data access speed from the main memory. The smallest possible chunk of memory that can be loaded into a cache is called a *cache line* and has typically the size of 32, 64 or 128 bytes. Each cache memory can only store a limited amount of cache lines determined by the cache size. For instance, a 64-kilobyte cache memory with 64-byte lines can store 1024 cache lines. When the CPU needs some data it first checks if the fastest L1 cache contains the cache line with the desired data. In case of an L1 cache miss the CPU checks the L2 cache, if it contains the desired cache line it is loaded into the L1 cache to increase the access speed for subsequent accesses of this data, if even the L2 cache does not contain the sought after data the CPU continues searching up the hierarchy. Whenever the CPU finds the block of memory with the data in question, it loads it into a cache memory one level closer to the CPU unless it is already in the closest L1 cache. As seen in figure 4.3, the difference in speed between fetching data from main memory vs. L1 cache can be up to two orders of magnitude!

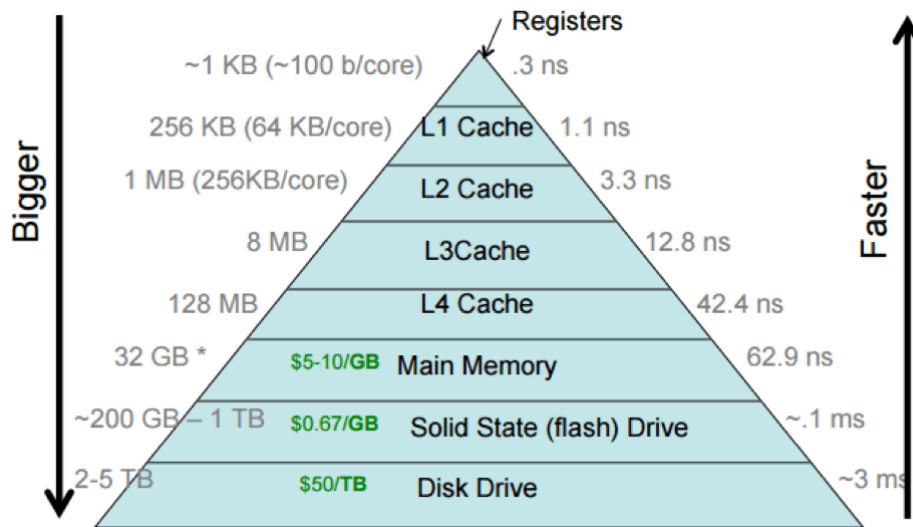


Figure 4.3: Comparison in data access latency between different memory types in the hierarchical memory model [15].

Data is not read in isolation, but instead, an entire cache line of memory is loaded into the hierarchy of fast cache memories. Therefore the most optimal way to write programs for modern hardware is to write such programs that utilise as much of the data loaded into the fast cache memory as possible and waste as little of the scarce cache memory as possible. The idea of storing related data close together in memory so that it can be read into cache and processed together efficiently is called the concept of *spatial locality* [12]. Another essential fact that programmers should keep in mind to write efficient code is the concept of *temporal locality*, meaning that data accessed with higher frequency has a higher chance of being still loaded into one of the caches resulting in significantly faster data access.

The knowledge of these and other important details about modern hardware enables DOD to yield much better-performing code than simply ignoring the hardware and only thinking about abstractions. This possibility of noticeably increasing the performance of software through the knowledge of hardware sparks interest in data-oriented design, especially among intermediate and experienced developers.

4.2 Principles of Data-Oriented Design

The main goal of DOD is to maximise the performance of software through the knowledge of hardware and actively designing software to efficiently store and transform data. DOD is an approach to programming that encompasses a variety of principles and ideas that are not new but are rather a reminder of first principles. This section is based on Mike Acton’s talk at CppCon 2014 [13] and Anne Van Ede’s master’s thesis [6]. The following sub-sections list some of the popular principles of DOD, but there are more optimisation techniques and ideas that can be found e.g. in the book „Data-oriented Design“ by Richard Fabian [9] or Mike Acton’s talk. There is also „Game engine architecture“ by Jason Gregory, a very well-written book that contains an approachable introduction to „Computer Hardware Fundamentals“ and „Memory Architectures“ in the correspondingly named chapters.

4.2.1 Data Storage Patterns Matter

It is important to choose an appropriate data structure to store data depending on the way the data will be accessed to ensure the optimal use of cache. To demonstrate what this means in practice imagine a city builder game where there are a lot of AI controlled trucks driving across the streets of the city. Assume that we decide to store each truck as a structure containing all its attributes and we store all trucks in an array as depicted in figure 4.4. This is similar to how objects in OOP are saved in memory. In games it is usually the case that it is necessary to evaluate the new state of all objects that are affected by some logic. This means that we need to iterate over our array of trucks e.g. compute their new positions. If our code needs all the data about each truck (the position, speed and the collision box) to compute the new positions of the trucks, this is would be an optimal storage pattern.

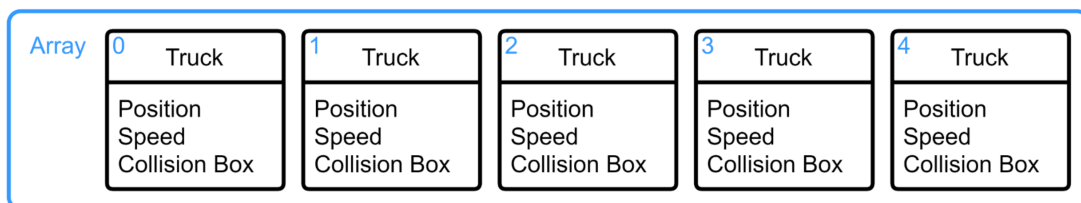


Figure 4.4: Example of an array of structures [6].

In practice, it is rarely the case that we use all the data related to an object when we perform specific calculations such as updating a position of a truck. In a more realistic example each truck structure would also probably contain items such as health, colour, capacity and other unnecessary data for the computation of a truck’s new position. In this case, the unnecessary data for a given computation will waste space in the cache line. A more appropriate storage pattern for this case would be storing each attribute of the trucks in a separate array as seen in the figure 4.5. Thanks to this separation, when we compute the new positions of the trucks the cache lines will be fully populated with only the attributes relevant for the computation resulting in a more efficient solution.

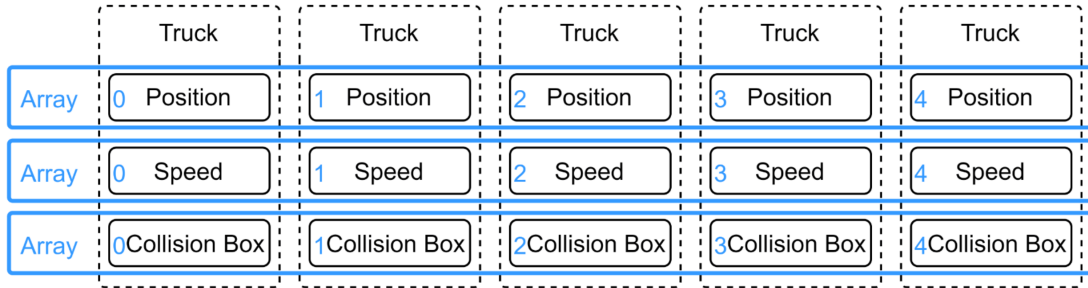


Figure 4.5: Example of a structure of arrays [6].

4.2.2 Data Access Patterns Matter

Even if related data is stored completely sequentially in a contiguous memory block exploiting the principle of spatial locality the way the data is accessed can hinder or boost performance. To illustrate this, imagine the following example. We have a grayscale image stored as a two-dimensional array of floating point numbers and we want to increase the brightness of the image by simply adding a constant number to each pixel. To achieve this we have to write a loop nested in another loop to iterate through all the pixels row-by-row or column-by-column and increment each pixel by the given amount. In most programming languages it is more efficient to iterate through the two-dimensional array row-by-row because that results in accessing data sequentially i.e. using all the data in each cache line. Using the column-by-column approach instead would result in jumping between non-neighbouring memory addresses, which would waste a lot of data in each cache line. Therefore, if we follow the philosophy of DOD we should consider these implications of how we write code and choose the row-by-row approach as it will likely perform better.

4.2.3 Vectorization (SIMD)

Vectorization is the process where multiple operations on scalars are replaced with a single operation on vectors (arrays) as seen in figure 4.6.

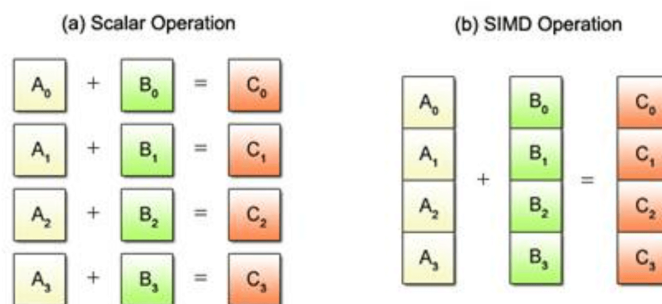


Figure 4.6: Scalar vs. SIMD operations [14].

If we store our data as a structure of arrays as described in the sub-section 4.2.1 then our data is perfectly prepared for vectorization. Thanks to the existence of *Single Instruction Multiple Data (SIMD)* instructions it is possible to process 4,8 or even 16 floating point numbers at once (depending on hardware). The code snippet 4.1 shows a simple example of performing four scalar operations separately.

```

1 int a[4] = { 1, 3, 5, 7 };
2 int b[4] = { 2, 4, 6, 8 };
3 int c[4];
4
5 c[0] = a[0] + b[0];    // 1 + 2
6 c[1] = a[1] + b[1];    // 3 + 4
7 c[2] = a[2] + b[2];    // 5 + 6
8 c[3] = a[3] + b[3];    // 7 + 8

```

Listing 4.1: Example of four separate scalar operations [14]

The following code snippet 4.2 shows the equivalent of the code in listing 4.1 optimised to use vectorization, meaning that instead of four separate operations the computation is performed at once by leveraging SIMD.

```

1 int a[4] __attribute__((aligned(16))) = { 1, 3, 5, 7 };
2 int b[4] __attribute__((aligned(16))) = { 2, 4, 6, 8 };
3 int c[4] __attribute__((aligned(16)));
4
5 __vector signed int *va = (__vector signed int *) a;
6 __vector signed int *vb = (__vector signed int *) b;
7 __vector signed int *vc = (__vector signed int *) c;
8
9 *vc = vec_add(*va, *vb);    // 1 + 2, 3 + 4, 5 + 6, 7 + 8

```

Listing 4.2: Example of vectorization using SIMD [14]

4.2.4 Padding

Programmers should be careful how they define new data structures and be aware of the fact that there might be padding added in-between and even inside of data structures. This is done mainly for two reasons. First, there might be added padding into a data structure after data types that are not aligned with the boundaries of the natural size of the system (e.g. 32-bit system with 4-byte boundaries), padding is added in many architectures, including x86 and ARM because accessing data that is not aligned on its natural boundary can result in slower or even incorrect memory accesses. So a simple trick that every programmer can use to potentially reduce the size of a data structure is to order its elements by size as seen in figure 4.7.

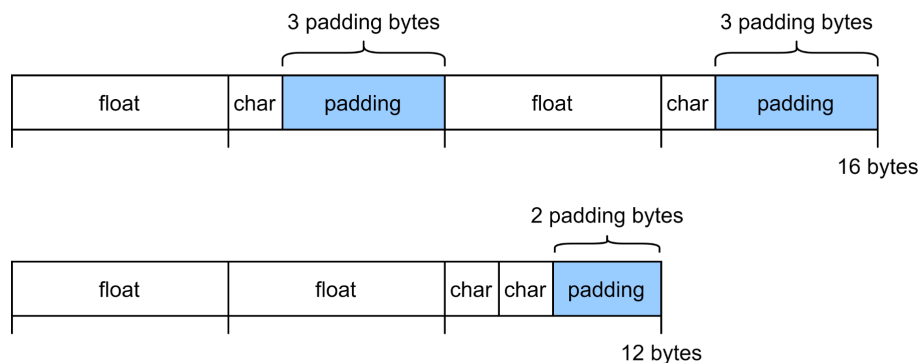


Figure 4.7: Example of rearranging a data structure to minimise its size due to unnecessary padding (in a 32-bit system with 4-byte alignment) [6].

Second, there might be added padding in-between data structures to prevent *data splitting*, which occurs when a single data structure is split between multiple cache lines. Data splitting can lead to significant performance costs due to different threads or different processors needing to synchronise if they try to modify a part of a data structure that is split between multiple cache lines [6].

4.2.5 Prefetching

Cache prefetching is a hardware feature designed to increase the performance of cache memory systems by speculatively predicting which data will be needed by the CPU before it is actually needed. The data that is likely to be needed soon by the CPU is loaded into the cache simultaneously while the CPU is still processing the current data already available in the cache. This reduces the amount of time when the CPU is inefficiently waiting for data to be loaded from the main memory. It is important to stick to predictable data access and data storage patterns to take full advantage of this feature. This can be achieved e.g. through storing homogeneous data sequentially in memory and then iterating over it [6].

4.2.6 Solve Problems You Have

Focus on solving the problems that you know you have to solve. Don't solve for problems that you don't have (e.g. the most general case and possible problems in the future) because it will likely cause more problems now and you will often waste energy on implementing an over-complicated solution that will not be even fully utilised. Instead of focusing on writing the most generic code possible, write simple and understandable code that can be easily replaced if necessary [13].

4.2.7 Reality is the Problem

There is nothing wrong with taking specific hardware and specific scenarios into account when writing code, it is not a hack to optimise an abstract problem to perform well on a real platform. The reality of running software on real hardware is the full definition of a problem. It is not possible to write optimised code without knowing which platform or finite set of platforms the software will be running on. When performance is the highest priority, the most beautiful code is likely not the right solution, but the code that runs fastest on the target platform is [13].

4.3 The ECS Framework

The Entity Component System (ECS) is a data-oriented framework that makes writing data-oriented code easier. This section introduces the most common implementations of this framework. In the context of DOD, it is important to know the specific implementation of ECS as each implementation has its trade-offs and performs best under different circumstances.

In ECS there are no objects as in OOP. In OOP objects are instances of a class that defines both the data and the logic associated with the objects of that class. The concept of ECS is based on the separation of data and logic. In ECS there are *entities* instead of objects. Each entity is associated with a set of *components* that store the entity's data. The logic is defined separately in *systems* that transform the data stored in the components to compute the new state of the entities in the system. As ECS is a data-oriented framework,

its implementation details are important. Three popular implementations of ECS are described in the following sub-sections. The main difference between them is how they keep track of which component belongs to which entity.

4.3.1 Big Array-based

The Big Array implementation of ECS stores each type of component such as position, name, collider etc. in a separate array. The components are linked to the entities that they belong to implicitly through the index in the components array. In other words, each component belonging to entity 0 is located in the array of the given component type at the same index 0. Entities that do not need a specific component type simply do not store any data at the corresponding index in the array of that component type.

This implementation is not really used in practice because it usually ends up wasting a lot of memory in the cache lines. This happens due to the fact that in real applications there is often a need for component types that are used only by a small percentage of entities. Consider the following example depicted in figure 4.8, we are making a game with soldiers and trucks. Every truck has a „Tire Friction“ component but soldiers do not need this component. When a system starts computing the new positions of the trucks based on their speed and tire friction it will need to load the array storing the tire friction components into the cache. A lot of the cache memory will be wasted because the array of tire friction components will contain unused allocated memory for each soldier entity. There will likely be much more soldiers than trucks in our game, so the majority of the indices in the array will be unused.

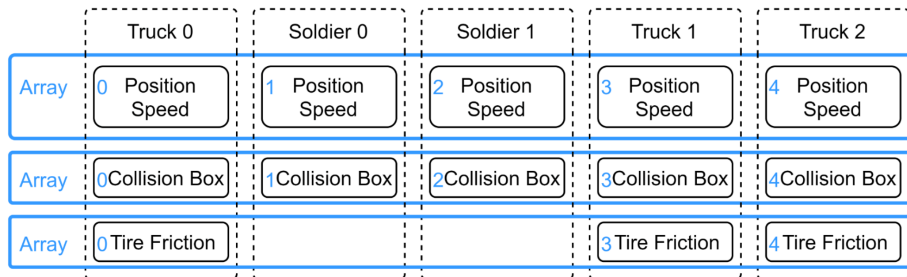


Figure 4.8: Example of components stored in a Big Array-based ECS implementation [6].

4.3.2 Sparse Sets-based

The Sparse Sets implementation uses two arrays to store each component type. The first array is a *component array* that contains tightly packed component data stored in a contiguous memory block and it has as many elements as there are entities with this type of component. The second array is a sparse *index array* that stores references to the component array for each entity. In the case where an entity does not have a particular component the slot in the index array is empty.

Let us imagine the same example as in the previous section where we are making a game with soldiers and trucks. The position-speed components are saved together contiguously in memory and we can access this component of each entity simply through the index directly from the components array. To access the tire friction component of each truck we first have

to find the actual index in the index array and then we can use it to access the component in the components array as seen in figure 4.9.

This implementation is a little less straightforward to implement than the Big Array approach but it utilises cache lines more efficiently. A big advantage of this implementation is that it enables us to perform *structural changes* to entities in constant time, meaning that the addition or removal of components to entities takes constant time. To add a component we simply add it to the end of the components array and reference it from the index array. To remove a component we delete its data from the component array and delete the reference in the index array. In case there is an empty slot in the components array after a component removal from an entity the next time we add a component of this type to an entity, we simply insert it in this slot to form a contiguous memory block.

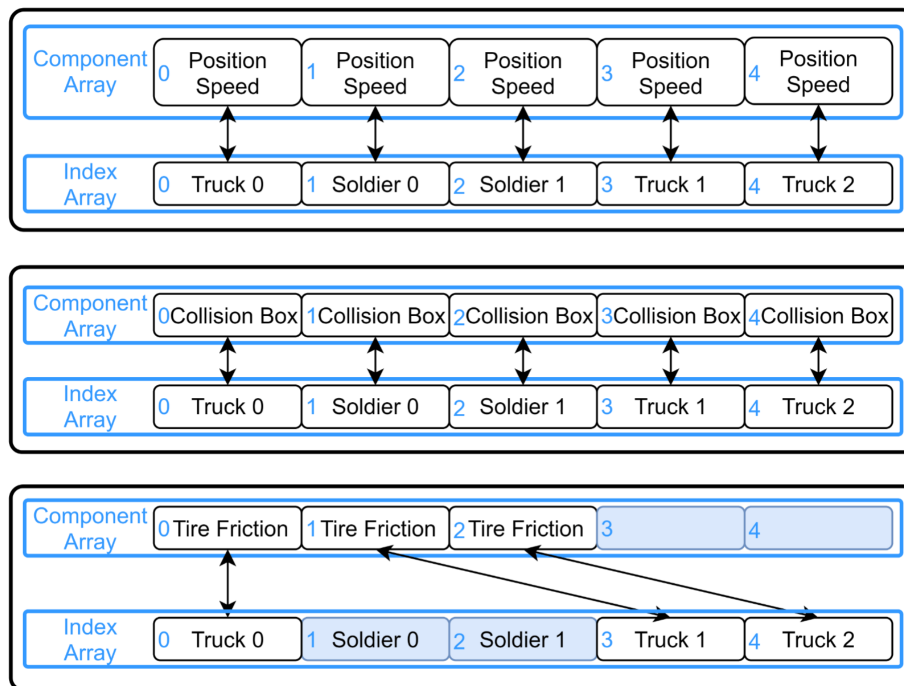


Figure 4.9: Example of components stored in a Sparse Sets-based ECS implementation [6].

4.3.3 Archetype-based

In an Archetype-based implementation of ECS the entities that have the same combination of component types are stored together in a contiguous block of memory and are said to be of the same *archetype*. Because systems usually process entities with the same set of component types together it is convenient and performant to group these entities together in memory. The downside of this implementation is the high cost of structural changes. If we consider making a game with trucks and soldiers once again if we remove the Tire Friction component from a truck entity, its archetype would change and it would be moved in memory over to the archetype currently containing soldiers as seen in Figure 4.10. Because the archetype of entities can change at any time we do not typically give names to specific archetypes or entities. When we want to perform some data transformation we simply query all the entities that match a given set of components in our system instead of using an archetype name such as „Truck“ or a „Soldier“.

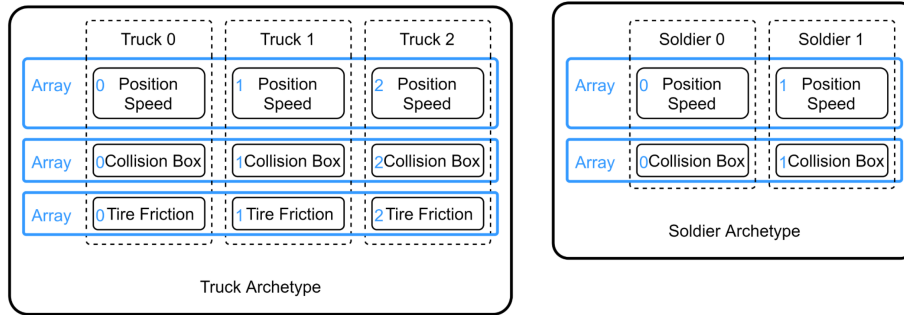


Figure 4.10: Example of components stored in an archetype-based ECS implementation [6].

4.4 Unity’s Data Oriented Technology Stack

Within the practical part of this thesis, the Unity Editor and Unity’s *Data Oriented Technology Stack (DOTS)* were used to develop a data-oriented implementation of Featherstone’s algorithm as described in chapter 5. This section briefly introduces Unity’s DOTS and specifically its implementation of an ECS framework.

Unity’s DOTS is a collection of tools enabling programmers to write high-performance data-oriented code easier while providing a safety net protecting them from the common pitfalls of concurrent programming¹. DOTS consists of three main parts. The first part is an Archetype-based ECS framework making it easier to start writing data-oriented code without having to worry about the smallest details. The second part is the *Burst* compiler that takes advantage of the sequential layout of related data thanks to the ECS framework and heavily optimises code through automatic vectorization, function inlining and memory access optimisation. The Burst compiler is also able to target a variety of platforms and removes the necessity of writing platform-specific code manually. The third and last major part of DOTS is the C# job system which makes it easier and safer to schedule code to run on multiple threads in parallel and also simplifies handling dependencies and synchronisation between jobs.

4.4.1 Unity’s ECS Implementation

In Unity’s implementation of ECS, each entity lives within a world and has its unique ID and version number. The entity’s ID is an index into an array of entity metadata and the version number indicates how many times the entity has been recycled after deletion. To be able to look up entities within a world, each world has an entity manager storing an array of entity metadata. Each element of this array contains a pointer to a *chunk*, which is a 16KB block of memory of a single archetype, and an index within the chunk. Each chunk contains an array of entity IDs and an array for each component type. To look up the components of a specific entity we would first visit the index of the entity’s ID in the metadata array, then follow the chunk pointer and read the data at the index within the chunk in each component array [16].

¹You can read more about Unity’s DOTS here: <https://unity.com/dots>

Chapter 5

Implementation

A major part of this master’s thesis is a data-oriented implementation of Featherstone’s algorithm. My implementation is developed in the C# programming language within the real-time 3D development tool *Unity*¹. More importantly, Unity’s Data-Oriented Technology Stack (DOTS)² including the *Entity Component System* (ECS) data-oriented framework was leveraged to enable building a performant solution. My implementation uses the data-oriented framework enabling easy structuring of the code into entities, components and systems. Unity’s numerics library is also utilised to avoid re-implementing standard matrix operations. The data-oriented implementation of Featherstone’s algorithm itself is my original contribution and is based on my hand-computed breakdown of the algorithm which I based on reputable literature [3] [1]. To the best of my knowledge, my implementation of Featherstone’s algorithm is the only purely data-oriented implementation of this algorithm at the time of writing this thesis and can hopefully serve as a stepping stone for future data-oriented implementations of this algorithm.

5.1 Used Technology

When developing a physics simulation it is very useful to be able to visualize the results of the algorithm to be able to evaluate its correctness and help with debugging. Since the scope of this thesis is already ambitious the real-time 3D development tool *Unity* was chosen to provide a convenient and easy way to render 3D objects to represent a kinematic chain. Besides the 3D rendering functionality, Unity also provides a data-oriented technology stack that enables easier and safer development of data-oriented code. Another great benefit of choosing Unity as a platform for my implementation is the fact that Unity provides an *Articulation Body* component which can be used to construct and precisely simulate kinematic chains. The *Articulation body* component is an integration of the *Articulations* feature in PhysX, an object-oriented implementation of Featherstone’s algorithm that I used for comparison to evaluate the correctness of my data-oriented implementation.

5.1.1 DOTS

The Data-Oriented Technology Stack (DOTS) provided by Unity is a collection of technologies and tools focused on enabling the development of performant data-oriented code. Its main component is the archetype-based Entity, Component, System (ECS) framework

¹<https://unity.com>

²<https://unity.com/dots>

which stores data in memory in a way that enables performant access and processing of this data. The basic building block of ECS is an entity. All entities can have different components storing a specific set of data attached to them. All entities with the same set of components form an archetype which is saved contiguously in memory. The contiguous storage of related entities enables performant loading and processing of their data by the logic defined in systems. Data storage and data access patterns are key to achieving high performance on modern computers which are limited by the memory bandwidth as explained in further detail in chapter 4. Maintaining optimised storage of entities in memory and their fast access is handled by Unity's ECS framework. All a user has to do is create entities, add components to them and query for them in systems to read and modify their data. Despite the ECS framework simplifying the data-oriented development process, the developer must adapt to this quite different way of programming compared to object-oriented programming. The developer must also understand how ECS works under the hood to leverage its full power and not result in even worse performance than with an OOP solution.

The second big component of DOTS is the Burst compiler which translates IL/.NET bytecode into optimised native CPU code by using the LLVM compiler. It can target many different platforms without requiring the programmer to write platform-specific code. Thanks to the restrictions and organisation of data in memory provided by the ECS framework the Burst compiler can increase the performance of a simulation by up to several orders of magnitude by simply flipping a switch in the settings. The Burst compiler is capable of automatically vectorizing code i.e. performing multiple computations in one instruction thanks to intrinsic SIMD instructions, inlining of functions (removing the call to a function and directly executing the function's body) and memory access optimizations.

The last major part of DOTS is the C# job system. Within systems in ECS one can write code directly in the system's `OnUpdate` method or in a job that is then run on a main thread or scheduled to run on one or multiple worker threads. Initially, I started implementing the logic for the first pass of the algorithm directly in the `OnUpdate` method of the `FeatherstoneSystem` as it made the access of data a little easier and I could focus primarily on the implementation of the algorithm itself. Later on, when I got more comfortable with DOTS I implemented the second and the third pass of the algorithm in their own jobs which makes it easy to schedule them to run in parallel on multiple worker threads. This means that multiple kinematic chains in a scene will be processed partially in parallel, which is faster than if they were all processed sequentially on the main thread.

5.1.2 Numerics library

The implementation of Featherstone's algorithm involves a lot of 3x3 and 6x6 matrix and vector operations. Luckily, I had the possibility to use an internally developed highly performant numerics library at Unity that includes common linear algebra operations. This library is compatible with DOTS and also leverages the Burst compiler to maximise performance. Because this is a low-level library optimised for performance it requires the developer to manually allocate a fixed amount of memory at the beginning of the simulation from which chunks of memory are later used to store the vectors and matrices that are being processed. This restriction forces the programmer to evaluate the maximum amount of memory that will be needed for the processing of the data structures at any given time in the simulation.

5.2 Structure

My implementation of Featherstone’s algorithm is split between two systems, one type of entities representing the links in a kinematic chain and several components for storing the necessary data. In total, the implementation is under about 1000 lines of code, but it took me about half a year and an extension to the original deadline to complete. This thesis took me so long to complete because I had to familiarise myself with the required physics concepts, get used to this type of scientific literature, and familiarise myself with the numerics library and the whole data-oriented technology stack that was being finished and fully released during the work on my thesis. When it came to the details of DOTS, I was also often dependent on help from the Unity physics team, which is dispersed across multiple locations with different time zones, adding to the development process’s complexity.

5.2.1 Entities

Each link in a kinematic chain is an entity in my simulation. Joint data is attached as a component on the inboard link and its index corresponds to the outboard link following Featherstone’s and Mirtich’s indexing convention. The last link in a chain does not hold joint data, therefore there are two archetypes in my simulation the links that do and the links that do not have the `RevoluteJoint` component.

5.2.2 Components

The two components that store most of the data are the `Link` component and the `RevoluteJoint` component. Most of the physical quantities that appear in the equations in Mirtich’s pseudo-code are stored in these two components. In order to store the 6-dimensional vectors and matrices they had to be broken down to `float3` and `float3x3` types as these are the highest dimension types supported by the `Unity.Mathematics` library. To perform operations such as vector-matrix multiplication in 6 dimensions, the matrix must be first constructed within the preallocated heap of memory and its four quadrants are initialised by loading the data from the 3x3 floats from the corresponding component. After performing the necessary computations, the result is persisted by loading the data from the temporary heap of memory back into a field of a component belonging to the corresponding link.

5.2.3 Systems

There are two systems in the simulation. The first system is the `ChainBuildingSystem` which is run once at the beginning of the simulation and initialises a `chainedEntitiesBuffer` on each stationary base link (one per kinematic chain in a scene). This circumvents the limitations of the ECS framework and provides me with a way to traverse the kinematic chain in both directions i.e. from base to tip and from tip to base as required by the algorithm. To enable the construction of this buffer the developer has to manually specify the successor of each link in the `RevoluteJointAuthoring` component while creating a scene for the simulation which resembles a singly linked list. The kinematic chain is traversed in the `ChainBuildingSystem` and a buffer with random access to entities in the chain is created and stored on the base link of each kinematic chain in the scene for use in the actual simulation.

The second system is called the `FeatherstoneSystem` and it contains the implementation of the algorithm itself. Before implementing this system I first created a detailed

breakdown of the algorithm presented in chapter 3 and hand-computed a single iteration of each pass of the algorithm to understand how it works and have some initial values I could use for debugging. The first pass of the algorithm is implemented in the systems `OnUpdate` method and it uses an idiomatic `foreach` to query for and iterate through the base links of all kinematic chains in the scene. Each kinematic chain is traversed using the `Successor` field in the `RevoluteJoint` component starting from the base links. The computations of the velocities, the Coriolis force and the initialisation of the articulated zero acceleration force along with the articulated inertia are done during the traversal. The second and third passes are implemented as jobs that are easily parallelizable and they compute the articulated zero acceleration force, articulated inertia and the acceleration of each link and joint.

5.3 Used Conventions

In order to be able to effectively develop, debug and review the code I decided to use certain naming conventions and utilise comments throughout the code to make it easier to map the code to the hand-computed calculations. Very often the computed quantities have an index written in subscript that is either i or $i - 1$ because it is not possible to use the minus sign in the name of C# variables I use the suffix `_current` instead of i and `_previous` instead of $i - 1$. An exception to this rule are variables that do not represent a mathematical expression and their names are not combined with other variable names such as `currentLink`. I generally tried to name the variables either with the name of the quantity that they represent such as `AngularAcceleration` or with a name that is close to how the computed expression would be written in L^AT_EX such as `s_i_prime_I_i_articulated` resembling $\dot{s}_i \hat{I}_i^A$. I have separated the code for each pass into sections using comments describing what each section is computing to make it easier to find in the pseudo code and the hand-computed breakdown of the algorithm.

5.4 Debugging

For a long time during the implementation, there was no visual result that could be verified. Therefore, the only way to continuously check the correctness of the code was to hand-compute the algorithm and then compare the intermediate results with the results on paper. Towards the end of the development process, it was finally possible to visually evaluate if the algorithm was implemented correctly. Of course, after the first run with the enabled visualization, it was not working as expected. After extensive code review and debugging the results looked almost correct, but it was still visible that there is energy being injected into the system and it was not behaving naturally. I accomplished to find the mistakes in the first iteration of each pass and within the first frame of the simulation, this is what I had hand-computed reference values for.

It would not be practical to hand-compute the values of further iterations for each link in the chain so I utilised the fact that Unity has an object-oriented implementation of Featherstone’s algorithm in the form of the `Articulation Body` component. With the help of Unity’s physics team, I built the Unity editor along with PhysX in debug mode which enabled me to step through the PhysX code while the Unity editor was running my simulation. A scene containing two equivalent kinematic chains composed of three links was made. One of them was simulated using my implementation and the other one using

the built-in `Articulation Body` components powered by PhysX. To locate further bugs in the code I thoroughly stepped through both my implementation and the implementation of PhysX to see where the intermediate results start diverging. I used these insights to eliminate the remaining bugs in the code and achieve equivalent results to PhysX but in with a data-oriented implementation. The following chapter ?? compares my implementation with the PhysX implementation in more detail and presents the results of the performed tests.

Chapter 6

Testing

A series of tests were carried out to evaluate the performance of the data-oriented implementation of Featherstone’s algorithm developed as a result of this thesis. The setup and the results of each test are described in this chapter. Each test measures and compares either the accuracy, energy conservation or performance of my implementation against the equivalent feature of the PhysX physics engine called „Articulations“. This feature was used through the „Articulated Body“ component in Unity which is a wrapper around „Articulations“ in PhysX. Specifically, PhysX 4.1 was chosen for comparison as it is integrated into Unity which makes the testing process easier. An important fact is that the „Articulations“ feature for the simulation of articulated bodies is powered by an object-oriented implementation of Featherstone’s algorithm which enables this comparison of my data-oriented implementation against an object-oriented one.

Both PhysX and my implementation use Euler’s integration method. In all the tests except one a time step of $1/60$ (0.01666) of a second was used. In one test a four times smaller time step $1/240$ (0.004166) of a second was used which led to a much more stable simulation of a double pendulum.

6.1 Accuracy and Energy Conservation

To measure the accuracy and energy conservation of the simulations performed with the data-oriented implementation of Featherstone’s algorithm two types of scenes were used. One type contained a kinematic chain set up as a single pendulum with a static base link and one dynamic link attached to the base by a revolute joint as seen on the left side of Figure 6.1. The second type of scene contained a double pendulum with a static base and two dynamic links as seen on the right side of Figure 6.1. The joints were simulated as ideal joints with no friction and the experiment was assumed to be in vacuum with a constant gravitational force.

Each test scene contained two copies of the same kinematic chain in a single or double pendulum configuration, one copy driven by an OOP and the other driven by a DOD implementation of the algorithm. These chains were often placed at the same overlapping location to enable an easier visual inspection of differences in their motion. An orthographic camera looking in the direction of the z-axis was used in all the test scenes for easier visual evaluation of the correctness of the simulations.

Data about the world space angular displacement of each link in the kinematic chain from its equilibrium position was collected at each step of the simulation. On top of this

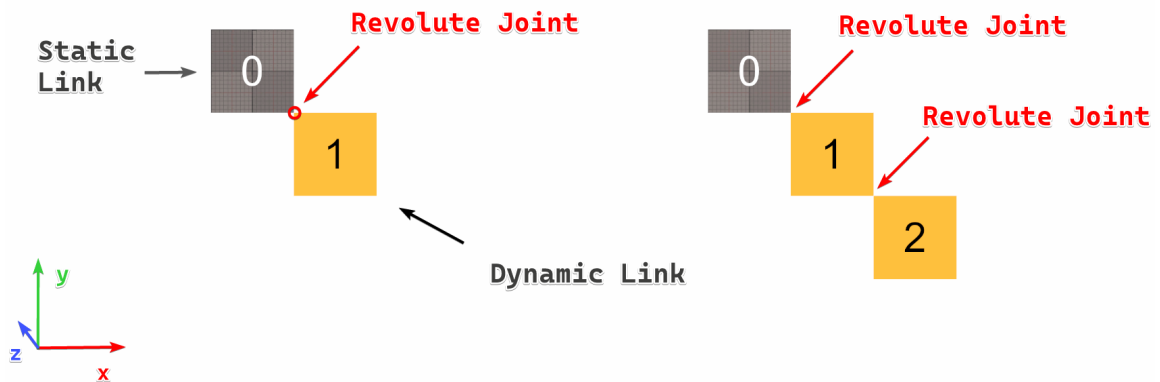


Figure 6.1: Screenshot of the single and double pendulum used for testing.

data, the kinetic, the potential and the total mechanical energy was also recorded after each step. To be able to compare data from simulations using different time steps the elapsed simulation time was recorded after each step of the simulation. The elapsed simulation time t was computed as follows using the current step number n and the time step Δt :

$$t = n\Delta t$$

In each of the following tests, a pendulum was simulated starting either from equilibrium and expected to remain there as seen on the left side of figure 6.2 or was displaced to a smaller angle of 45° or a bigger angle of 135° to test the qualities of the implemented solver. To exemplify this, the right side of Figure 6.2 depicts the trajectory of a single pendulum being dropped from an angular displacement of 45° .

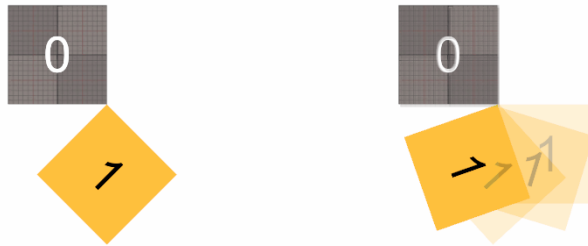


Figure 6.2: A single pendulum in equilibrium on the left and a visualisation of its motion after being displaced to 45° .

6.1.1 Single Pendulum 0°

In the first test, a single pendulum started in its equilibrium position and over the duration of the test was examined if it would stay there. In Figure 6.3 it is visible on the left chart that at first glance the pendulum exhibits no movement. After a more thorough examination of the zoomed chart on the right side it can be seen that the reference implementation of

PhysX produces a completely steady and correct result as opposed to my implementation which has a small amount of noise. This noise could be due to an implementation mistake or simply a trick that I have not used compared to PhysX.

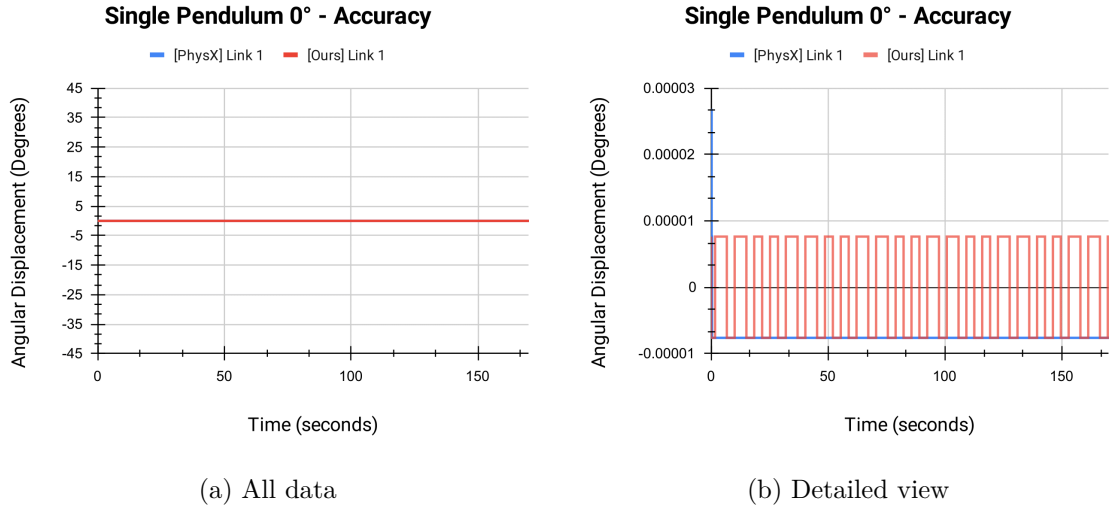


Figure 6.3: Plot of the angular displacement of the first link in the kinematic chain.

At first glance, both simulated systems seem to maintain a stable total amount of mechanical energy on the left chart in Figure 6.4. After examination of the right chart of the same figure, we can see that this is true but the constant amount of energy is slightly lower for my data-oriented implementation.

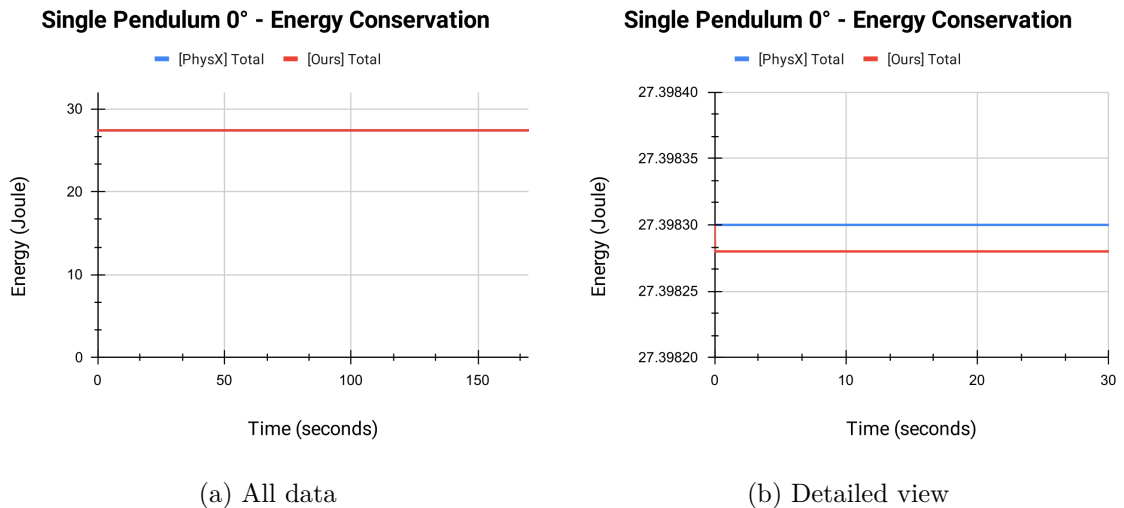


Figure 6.4: Plot of the total mechanical energy of the kinematic chain.

The kinetic energy of the system simulated with the data-oriented implementation is zero as expected and the potential energy is equal to the total mechanical energy of the system as seen in Figure 6.5. The results of this test were the same for the PhysX implementation but are not shown in Figure 6.5 to maintain clarity as all the values overlap.

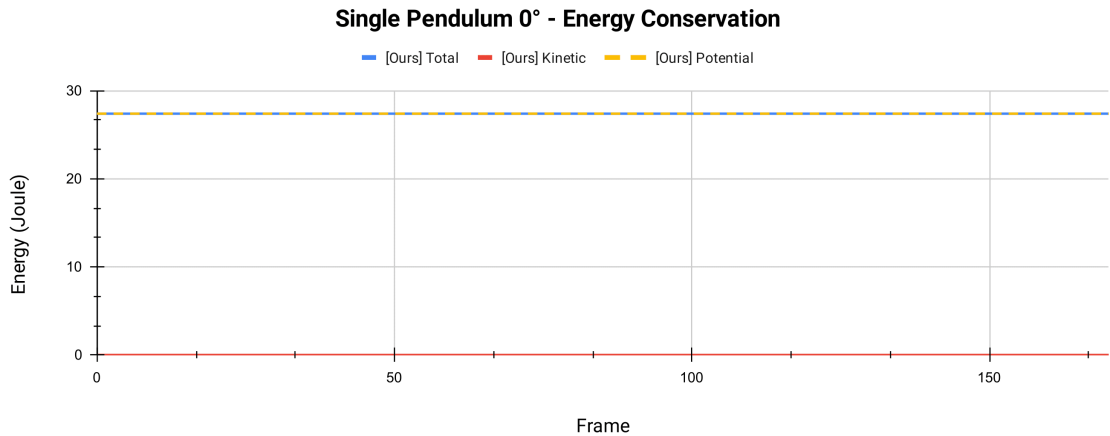


Figure 6.5: Plot of the total, kinetic and potential energy of the kinematic chain.

6.1.2 Single Pendulum 45°

The second test with a single pendulum had a small initial angular displacement of the dynamic link of 45° from its equilibrium position. In this test, both the PhysX and my simulation yielded the same angular displacement of the dynamic link over time as seen in Figure 6.6. The displacement perfectly overlaps at the start and the end of the simulation as seen on the left and right chart respectively.

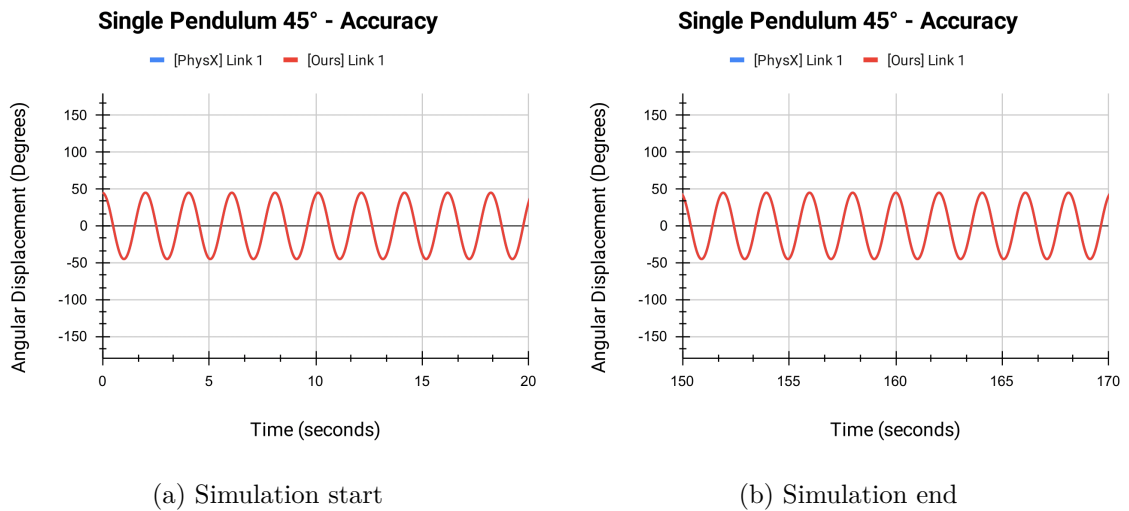


Figure 6.6: Plot of the angular displacement of the first link in the kinematic chain.

Both systems seem to maintain constant total mechanical energy when inspecting the left chart in Figure 6.7, but after having a closer look at the right chart it is visible that the energy of both systems oscillates around the initial value. Interestingly, the energy of the data-oriented implementation oscillates with approximately three times higher amplitude. This observation can indicate an error in the DOD implementation and an area that could be further examined during future work.

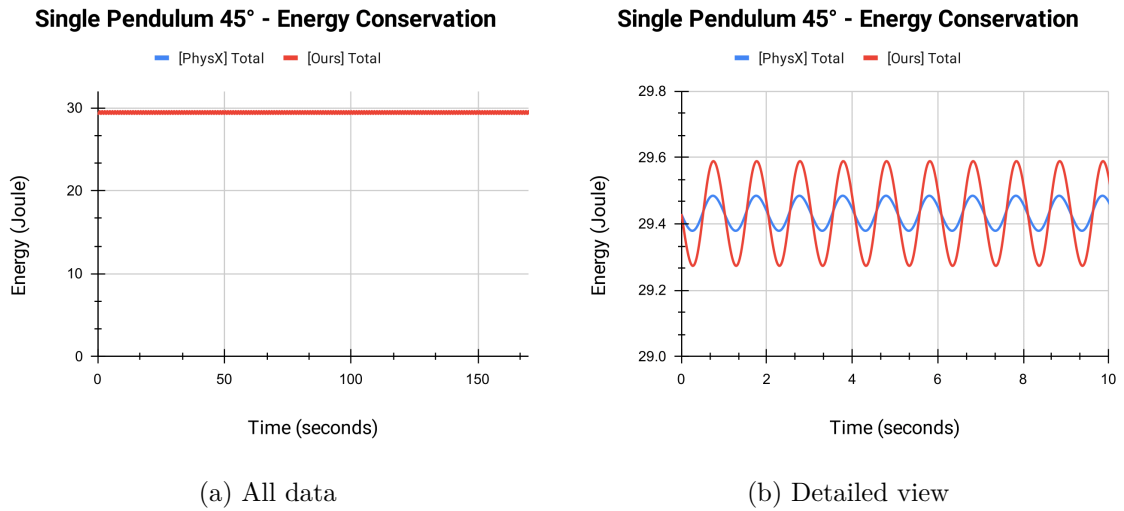


Figure 6.7: Plot of the total mechanical energy of the kinematic chain.

Over all the total mechanical energy, the kinetic energy and the potential energy of both systems over time is very similar but not identical as seen in Figure 6.8. These differences are not significant enough to result in visual artefacts visible by the naked eye during this test but are symptoms that are much more prominent in the tests featuring a double pendulum.

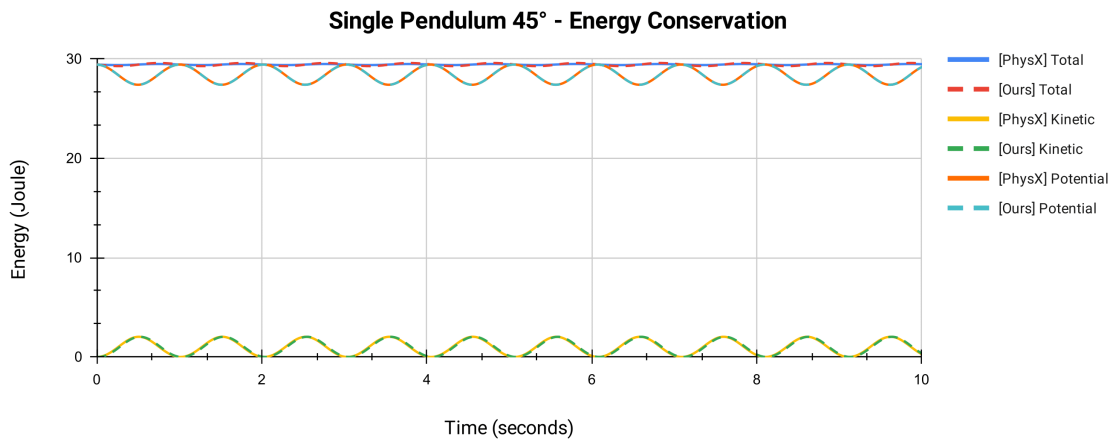


Figure 6.8: Plot of the total, kinetic and potential energy of the kinematic chain.

6.1.3 Single Pendulum 135°

The starting configuration of this test was a single pendulum with an initial angular displacement of its dynamic link of 135° from equilibrium to test how stable the simulation would be in more extreme conditions. Figure 6.9 shows that both my implementation and the implementation of PhysX produced very similar results throughout the entire lifetime of the simulation. Even though the results seem similar in the charts, they are not completely

identical. In fact, the average absolute difference between the displacement data from both simulations is 0.067333 degrees.

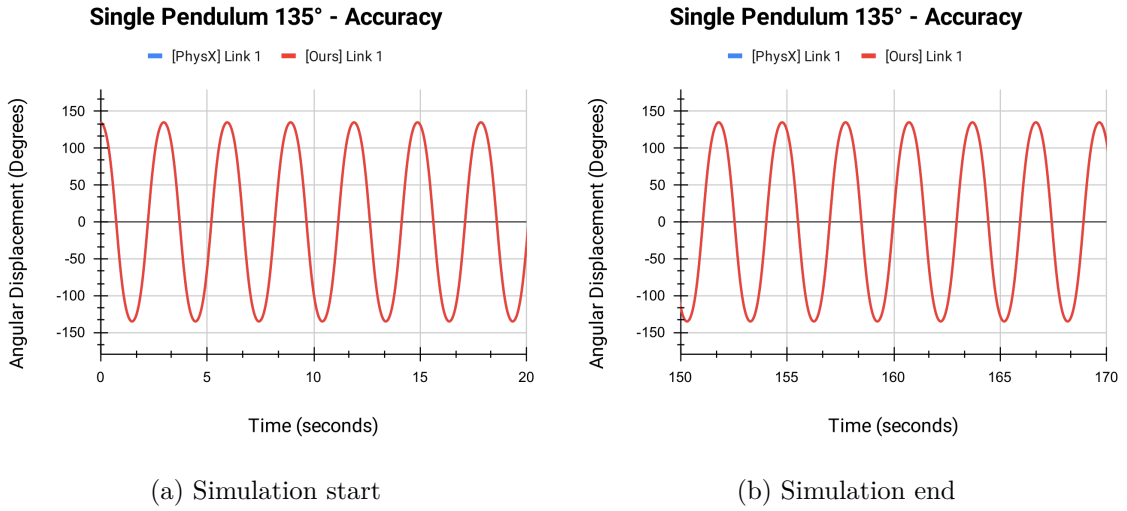


Figure 6.9: Plot of the angular displacement of the first link in the kinematic chain.

In this test with a higher initial angular displacement it is possible to see quite easily that the total mechanical energy of neither of the systems was constant as seen in Figure 6.10. Again, notice the higher amplitude of the energy of the data-oriented system. This is likely a symptom of an error in the implementation.

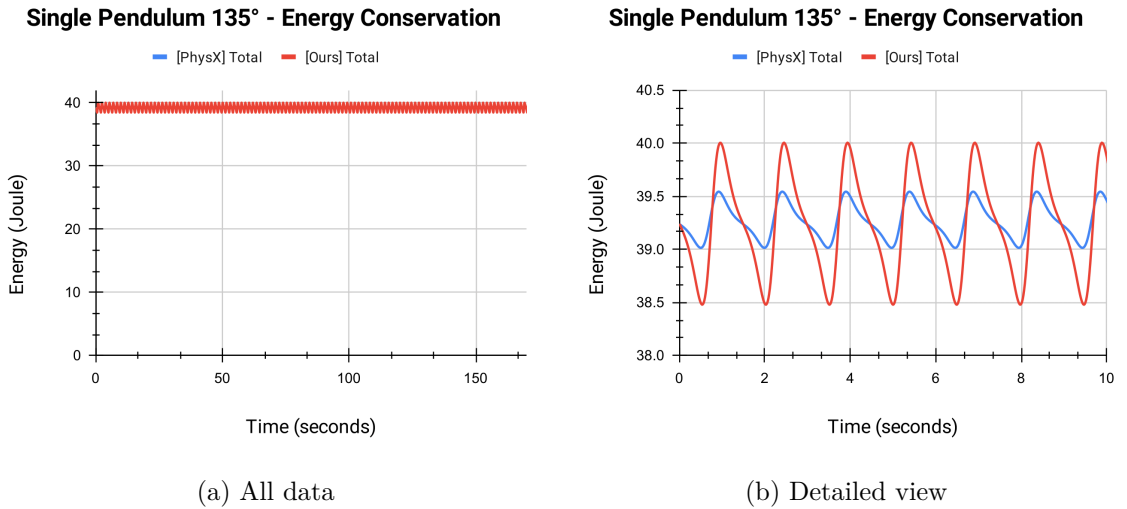


Figure 6.10: Plot of the total mechanical energy of the kinematic chain.

The absolute difference between the energies of both systems was not very big as seen in Figure 6.11.

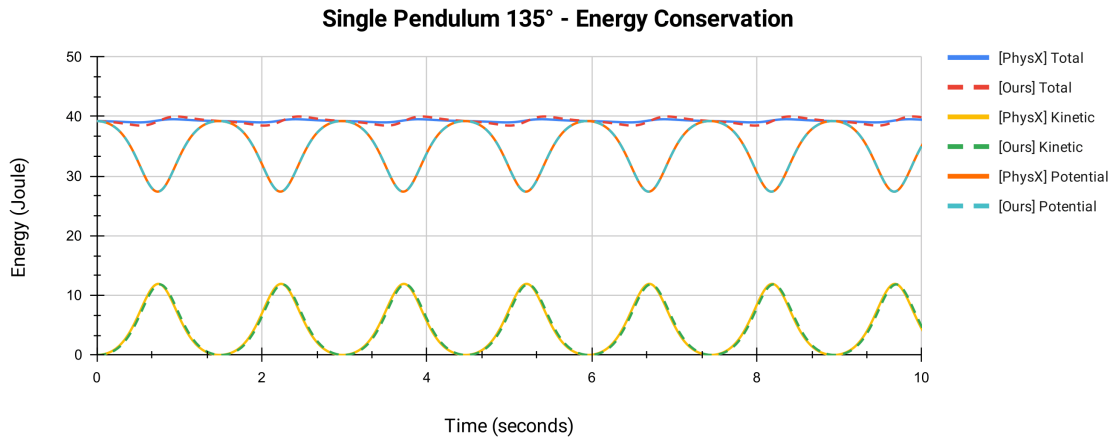


Figure 6.11: Plot of the total, kinetic and potential energy of the kinematic chain.

6.1.4 Double Pendulum 0°

All the single pendulum tests gave stable and quite accurate results when compared to the reference results of PhysX. This and the following tests use a double pendulum, meaning a mechanical configuration of two dynamic links chained together hanging of a static base as depicted earlier in Figure 6.1.

The first test with a double pendulum checks if a hanging pendulum remains in the state of equilibrium and examines if the mechanical energy in the system remains constant. From Figure 6.12 it is visible that the angular displacement of the first dynamic link in the chain is not constant and there is a small increasing amount of noise. This noise is more pronounced than the one in the single pendulum 0° test. The angular displacement noise being more significant in the double pendulum indicates that the error in the implementation is amplified by adding more links to the kinematic chain.

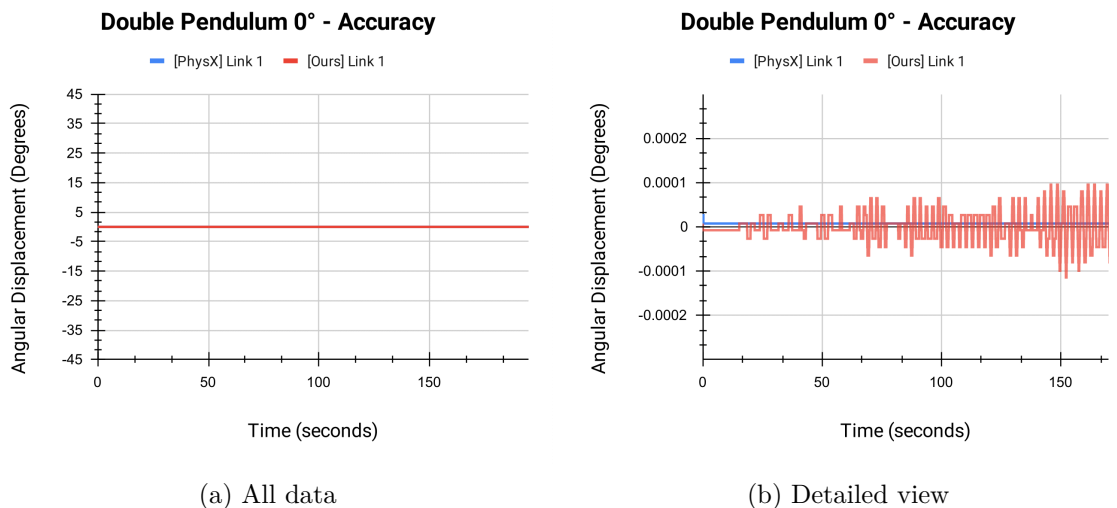


Figure 6.12: Plot of the angular displacement of the first link in the kinematic chain.

The angular displacement from equilibrium of the second dynamic link in the kinematic chain is even more significant than the one of the first link as seen in Figure 6.13. The highest absolute error is less than 0.00025 of a degree which does not result in any visual differences between the two chains, but could be a clue for future debugging.

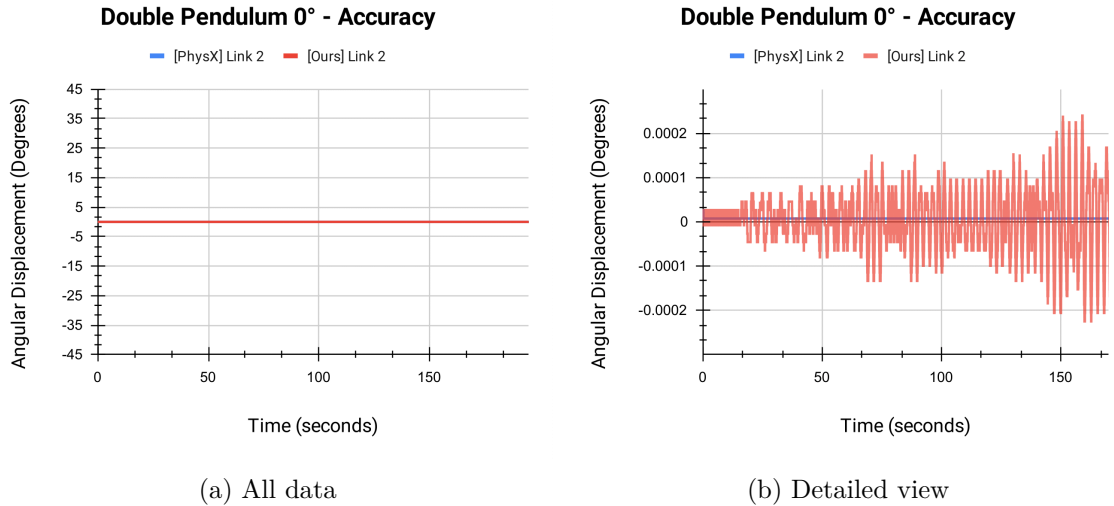


Figure 6.13: Plot of the angular displacement of the second link in the kinematic chain.

Both the PhysX and my implementation seems to keep constant mechanical energy over the time of the simulation as they are expected to. After a closer inspection of the level of constant energy in figure 6.14 the one of the data-oriented implementation is lower by about 0.0001 of a Joule which could be a consequence of the small difference in angular displacement.

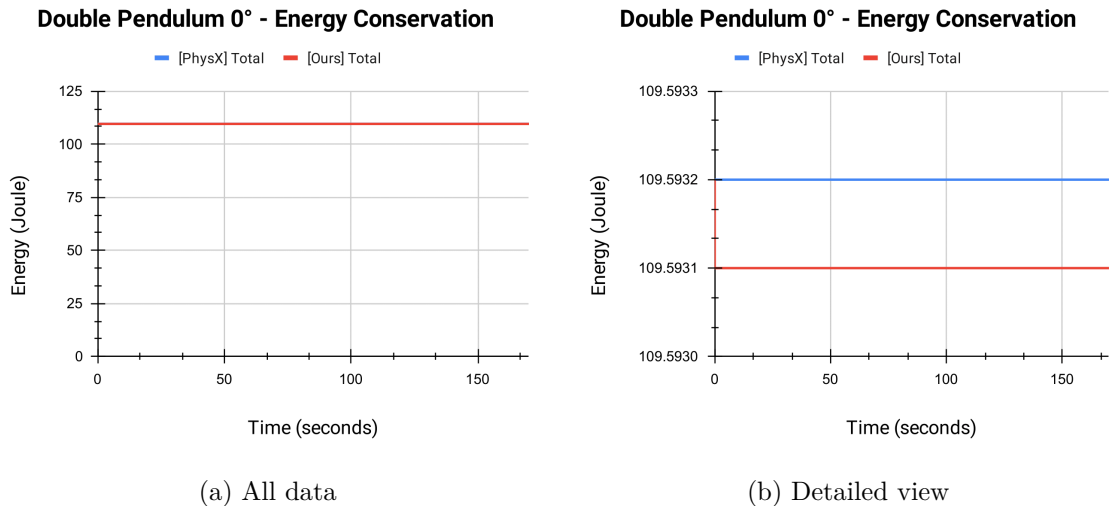


Figure 6.14: Plot of the total mechanical energy of the kinematic chain.

Both implementations produce very similar results and no major differences in energy conservation are visible as seen in Figure 6.15.

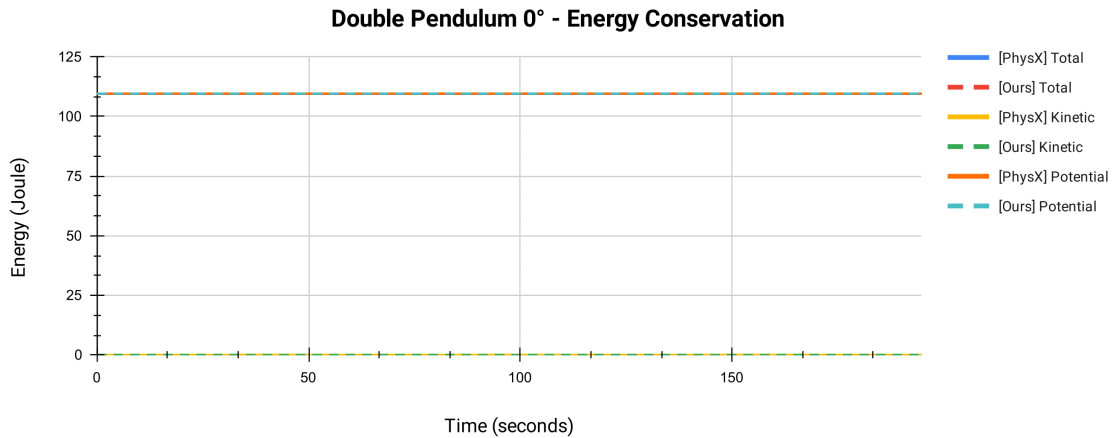


Figure 6.15: Plot of the total, kinetic and potential energy of the kinematic chain.

6.1.5 Double Pendulum 45°

In this test, a double pendulum was simulated with an initial angular displacement of 45° from equilibrium and after approximately 70 simulation seconds it ended up spinning out of control producing „NaN“ values and the energy of the system kept raising exponentially.

During the first 10 seconds of the simulation, the angular displacement of the first link was very similar between PhysX and my implementation as seen in Figure 6.16. Afterwards, the differences kept increasing more and more until the pendulum ended up spinning around.

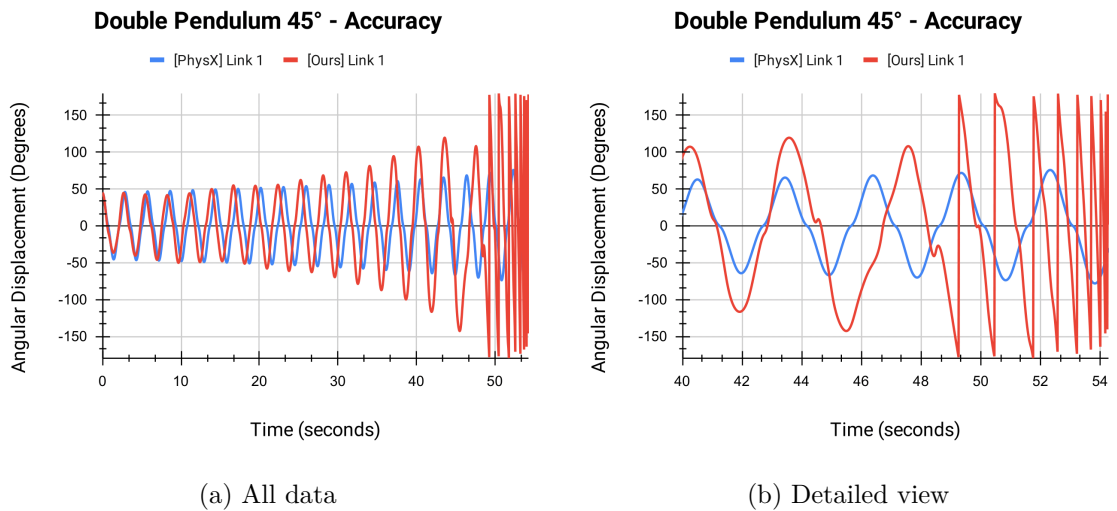


Figure 6.16: Plot of the angular displacement of the first link in the kinematic chain.

The differences between the angular displacement of the second link in the kinematic chain by each implementation were a little more significant as seen in Figure 6.17.

Interestingly, the total mechanical energy of both systems was rising over time, even the one of the system simulated by PhysX. It is known that PhysX performs best with a small amount of joint friction, but in all of these tests, zero friction was used. Nevertheless, the total mechanical energy of the system simulated with my implementation was rising at

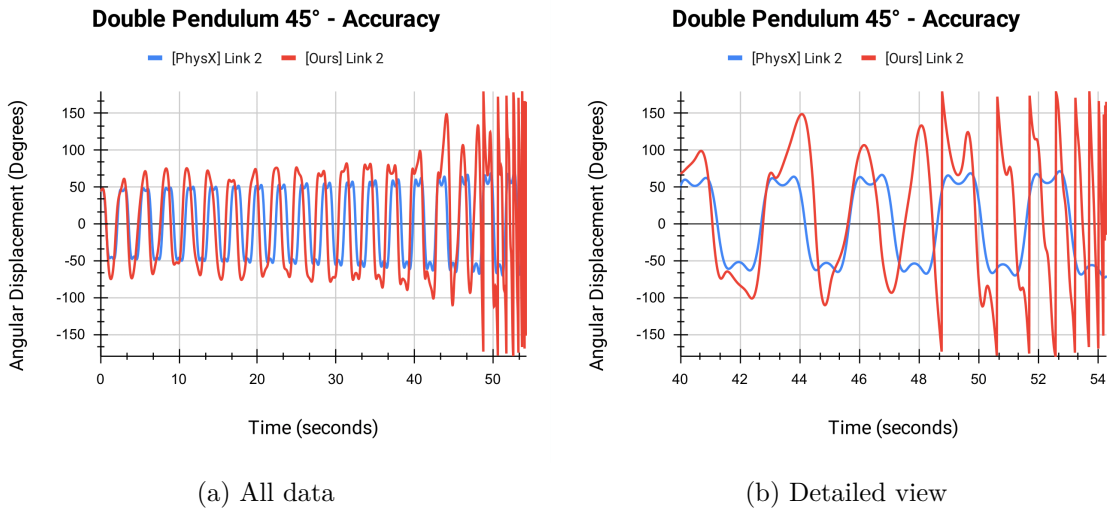


Figure 6.17: Plot of the angular displacement of the second link in the kinematic chain.

a higher rate than the one simulated with PhysX and started exponentially growing after approximately 50 seconds of simulation time as seen in Figure 6.18.

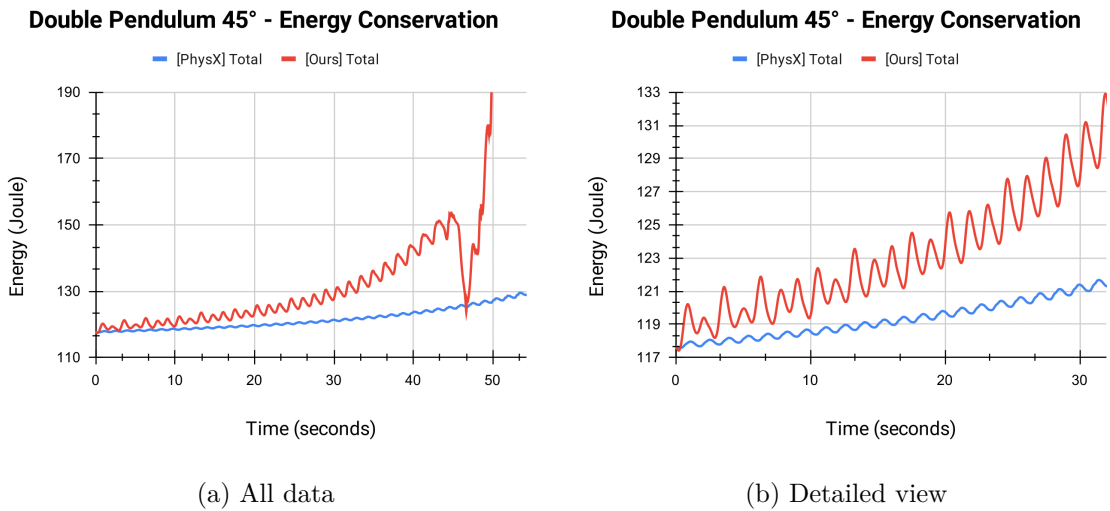


Figure 6.18: Plot of the total mechanical energy of the kinematic chain.

In Figure 6.19 it is visible that the kinetic energy of my implementation keeps rising where the potential energy reaches a top limit as expected, due to the limited distance that the links can reach away from the first joint.

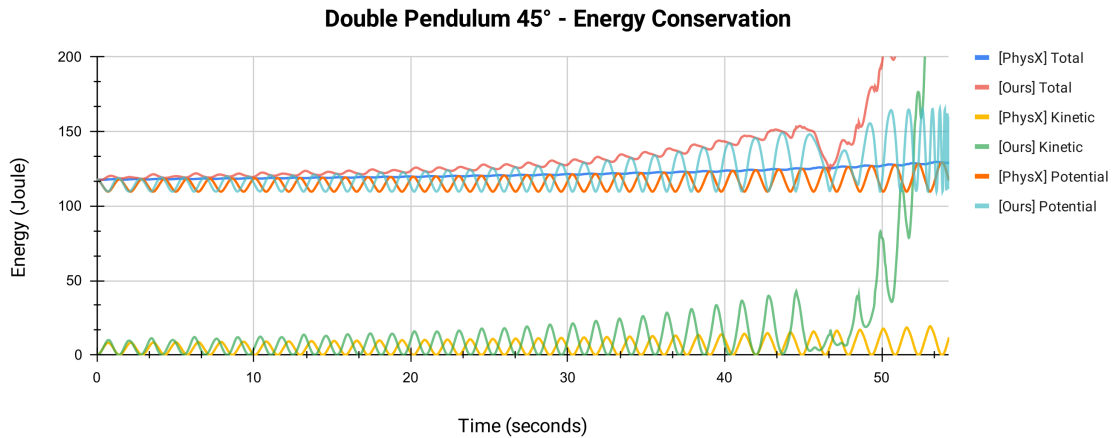


Figure 6.19: Plot of the total, kinetic and potential energy of the kinematic chain.

6.1.6 Double Pendulum 135°

In the most extreme test of the double pendulum, it was displaced 135° from its equilibrium state and the pendulum simulated with my implementation became unstable much faster than in the previous test where the pendulum started displaced at a 45° angle. In Figure 6.20 we can see that the first link starts spinning around after 8 simulated seconds whereas in the previous test, it took about 50 seconds which is more than 6 times longer.

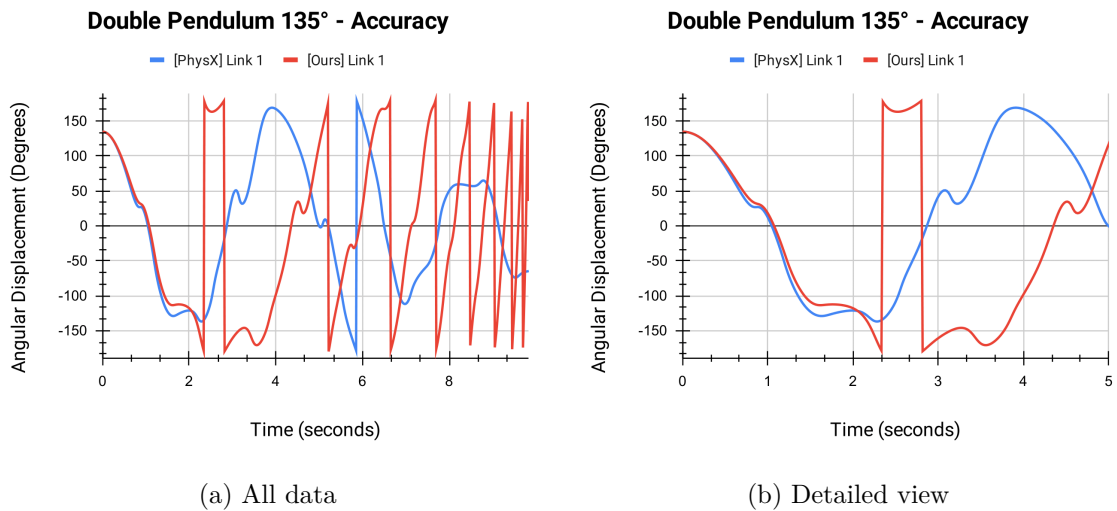


Figure 6.20: Plot of the angular displacement of the first link in the kinematic chain.

We can see much more movement of the second link through the whole simulation it also starts spinning around its joint much sooner as seen in figure 6.21.

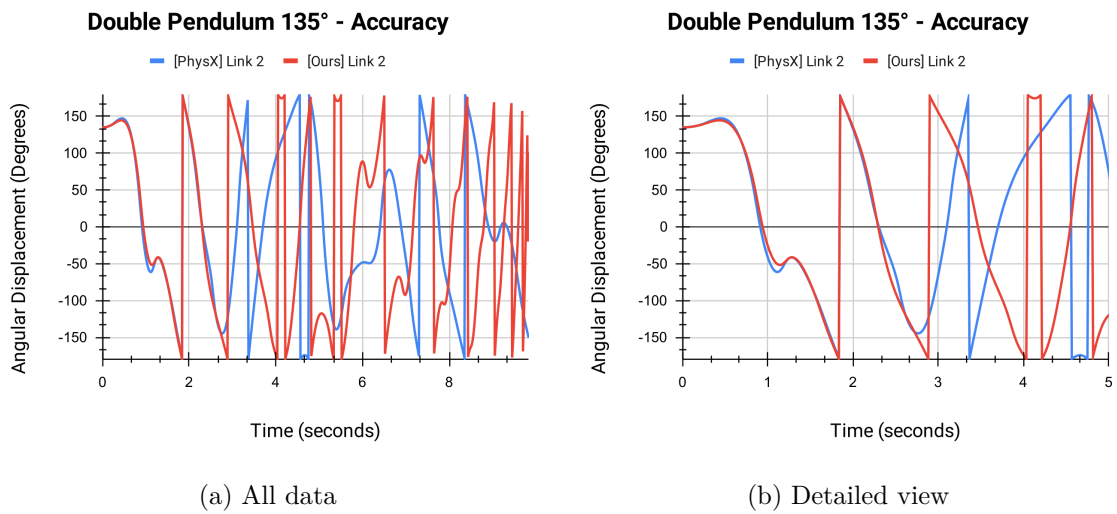


Figure 6.21: Plot of the angular displacement of the second link in the kinematic chain.

In the charts of the mechanical energy of the system we can see that the energy starts increasing exponentially already after 8 to 9 seconds of the simulation as seen in figure 6.22.

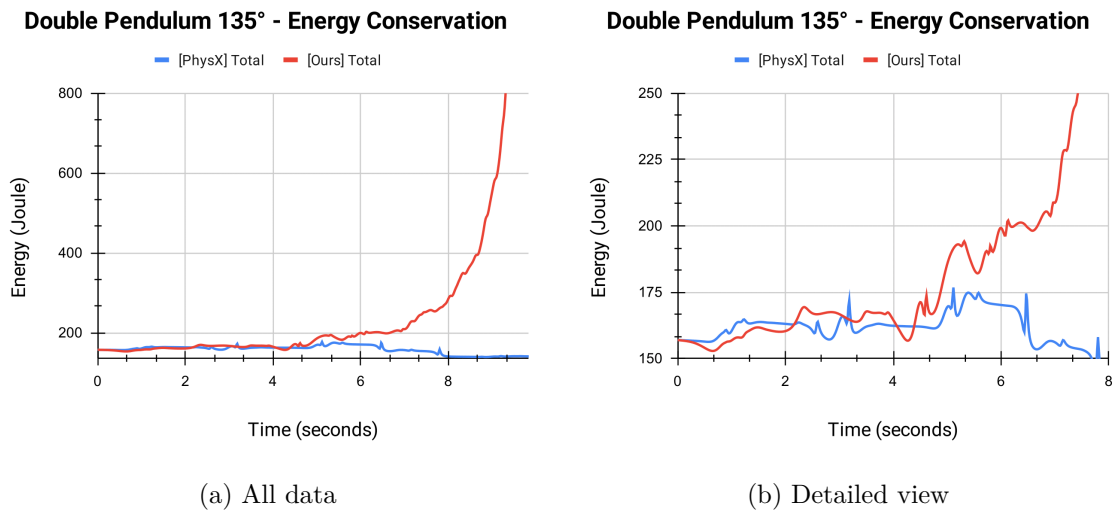


Figure 6.22: Plot of the total mechanical energy of the kinematic chain.

The kinetic and potential energy behaves similarly to the previous test, the kinetic energy keeps growing and the kinetic increases the oscillation frequency over time as seen in Figure 6.23.

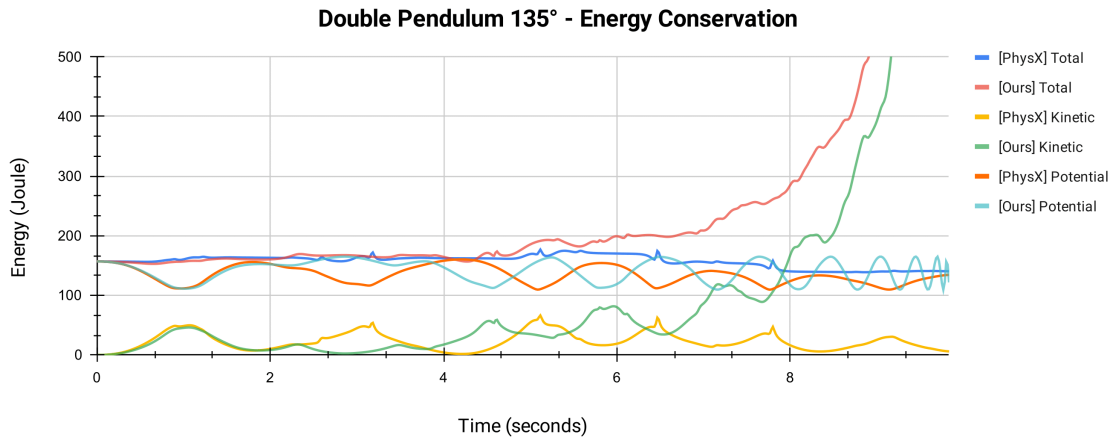


Figure 6.23: Plot of the total, kinetic and potential energy of the kinematic chain.

6.1.7 Double Pendulum 135° (Smaller Time Step)

In the previous and most extreme test of the double pendulum, the mechanical system became unstable after 8 seconds of simulation. In this test, the same starting configuration was used but the simulation step was lowered from 1/60 of a second to 1/240 of a second which led to great improvements in stability and accuracy of the simulation. In fact, the mechanical system remained stable over the entire lifetime of the simulation which was over 50 seconds of simulation time as seen in Figure 6.24. From the figure it is clearly visible that the accuracy has largely improved as the angular displacement of the first link simulated with my implementation closely follows the one of the first link simulated with PhysX.

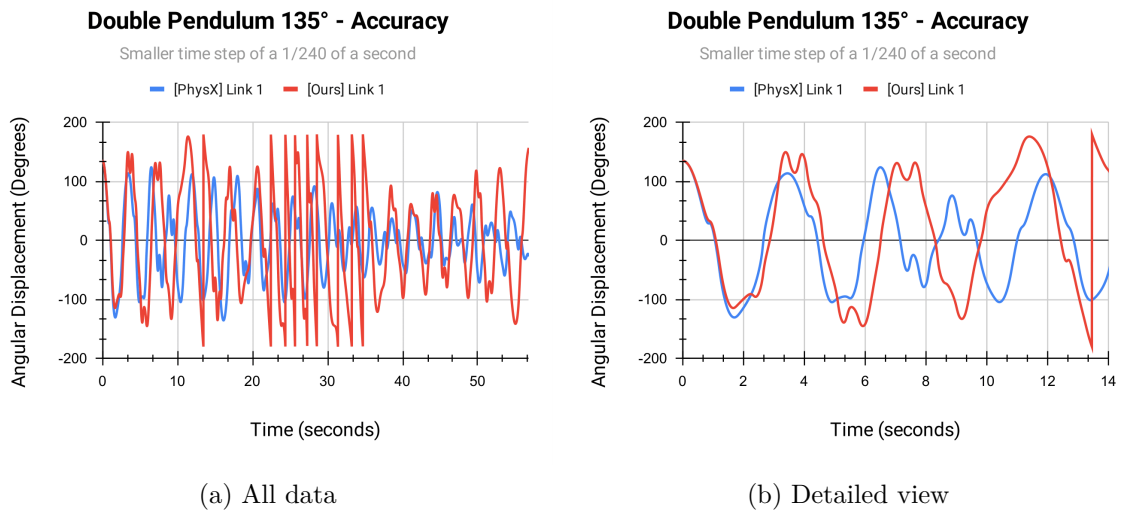


Figure 6.24: Plot of the angular displacement of the first link in the kinematic chain.

The second link in both kinematic chains did spin around sometimes as visible in Figure 6.25 which is expected of a double pendulum dropped from this height. Importantly the system remained stable.

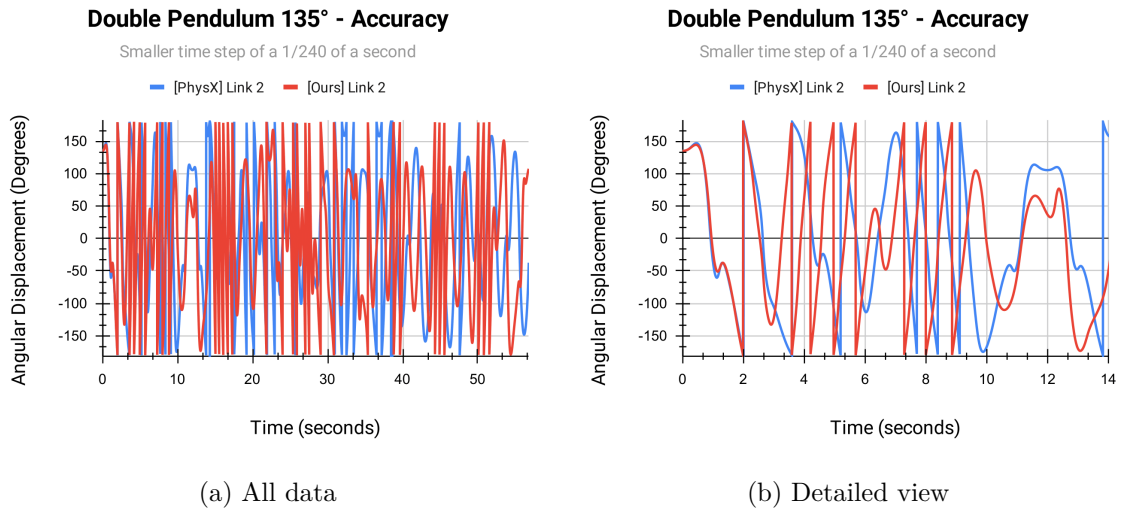


Figure 6.25: Plot of the angular displacement of the second link in the kinematic chain.

The total mechanical energy of both systems is far from constant where my implementation produced more turbulent oscillations in the energy graph. A big positive change is that in almost a six times longer simulation the energy of the data-oriented implementation did not grow uncontrollably and stayed within a much lower range of values as seen in Figure 6.26. This was expected as the smaller simulation step results in less numerical error and more precise simulation. It is to be investigated in future work why the energy in the system simulated by PhysX remains more stable and even decreases over time.

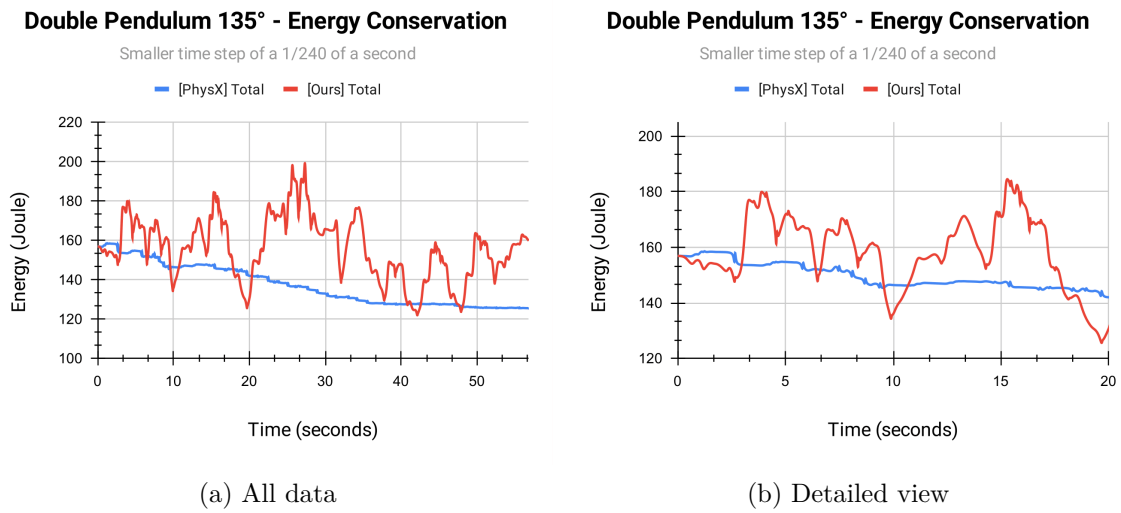


Figure 6.26: Plot of the total mechanical energy of the kinematic chain.

The kinetic energy of the data-oriented system is much closer to the one of the system simulated with PhysX as seen in Figure 6.27 if compared to the previous test with a bigger time step. It would be interesting to examine how much better the results can get with an even smaller time step. This could indicate if the error of my implementation is related to integration or lies elsewhere.

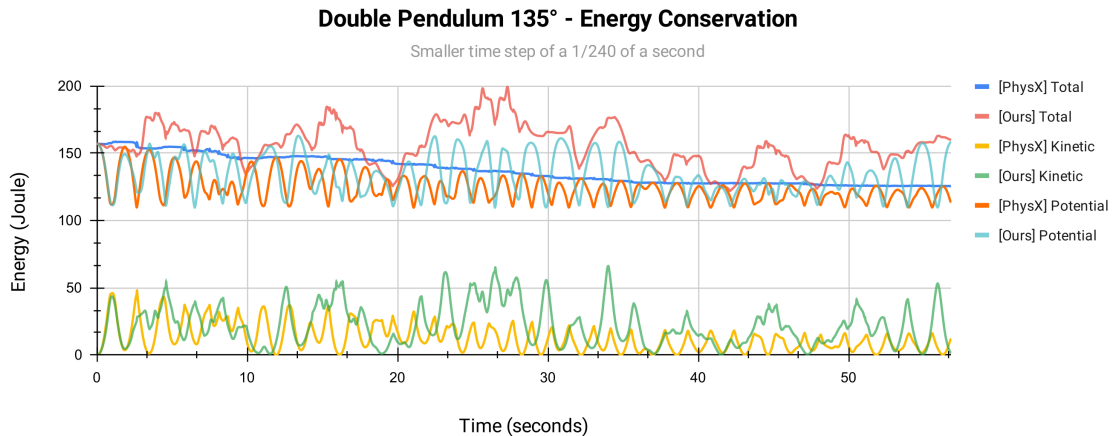


Figure 6.27: Plot of the total, kinetic and potential energy of the kinematic chain.

6.2 Execution time

Another interesting aspect of the quality of the data-oriented implementation is its performance in comparison to PhysX which is implemented in an object-oriented way. Due to a lack of time, the performance tests were not as comprehensive as the accuracy tests as it was deemed more important to collect more data about the correctness and accuracy of the implementation that could serve as clues for future debugging.

The performance was measured by recording the difference between the time since the startup of the simulation before and after computing the PhysX simulation step and before and after the data-oriented simulation step. It is not an entirely fair comparison as the PhysX implementation does check for collisions and has overall more overhead due to its more complete feature set. The Unity physics simulation mode was switched to `SimulationMode.Script` which enabled a manual control of the physics simulation and contributed to recording as correct testing data as possible.

The final performance test was carried out in the scene with the double pendulum displaced to 135° and using the smaller time step to prevent the simulation from ending prematurely. This scene was chosen as it features two dynamics links that require the utilisation of the full algorithm for recursively traversing the chain.

In terms of the testing details, the internal Unity editor 2023.3.0a1 was used as it is very close to the latest version of the editor and features the newest features and optimisations. The editor was switched into „Release“ mode from the default „Debug“ mode which can hinder performance. The performance was measured both with having Burst compilation enabled and disabled. In the case of enabled Burst compilation, safety checks that are useful for debugging but hinder performance were turned off. As there is a certain „warm-up“ time for both PhysX and the data-oriented implementation the first 100 and the last 100 data points were discarded to eliminate outliers.

The performance test was conducted on a Dell XPS 15 9510 laptop while being connected to a power supply and an external 4k monitor. All unnecessary applications were closed and both the test with enabled and disabled Burst compilation were performed right after each other to ensure that the computer was in as similar state as possible during both of the tests. A more detailed technical specification of the used laptop is described in the table 6.1.

Laptop model	Dell XPS 15 9510 (Windows 10 Pro)
GPU	NVIDIA GeForce RTX 3050 Ti Laptop GPU
CPU	11th Gen Intel(R) Core(TM) i9-11900H @ 2.50GHz - 8 Cores
RAM	2 x 32 GB DDR4, 3200 MHz

Table 6.1: Testing machine technical parameters.

In Figure 6.28 it is visible how with both Burst compilation being disabled and enabled the data-oriented implementation outperformed PhysX’s object-oriented implementation. As stated earlier this is not a perfect comparison but was made as fair as possible.

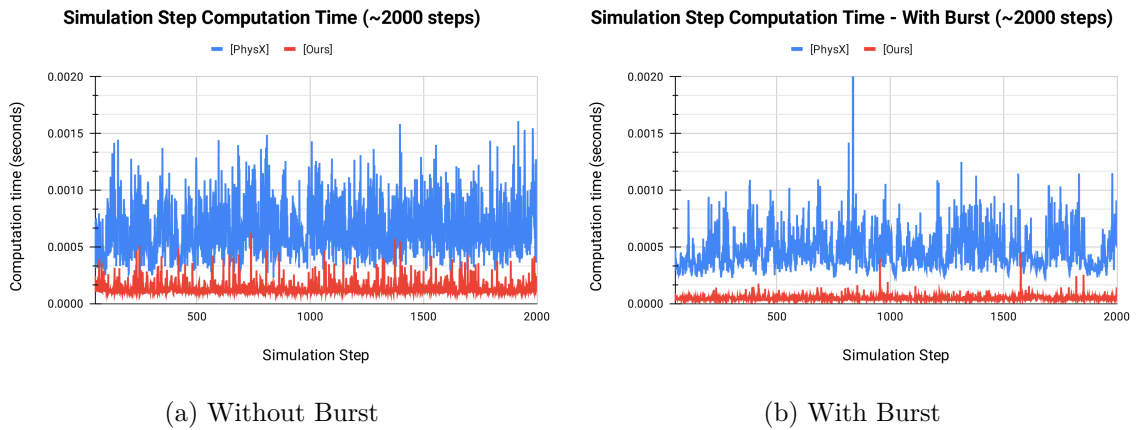


Figure 6.28: The recorded run times of each simulation step for a double pendulum.

In summary, the developed data-oriented implementation was a little over 5 times faster than PhysX without Burst compilation being turned on. After enabling Burst compilation, the data-oriented implementation was over 10 times faster than PhysX’s object-oriented implementation as seen in Figure 6.29. This indicates that the data-oriented implementation has the potential to become a competitive solution after more debugging and optimisation.

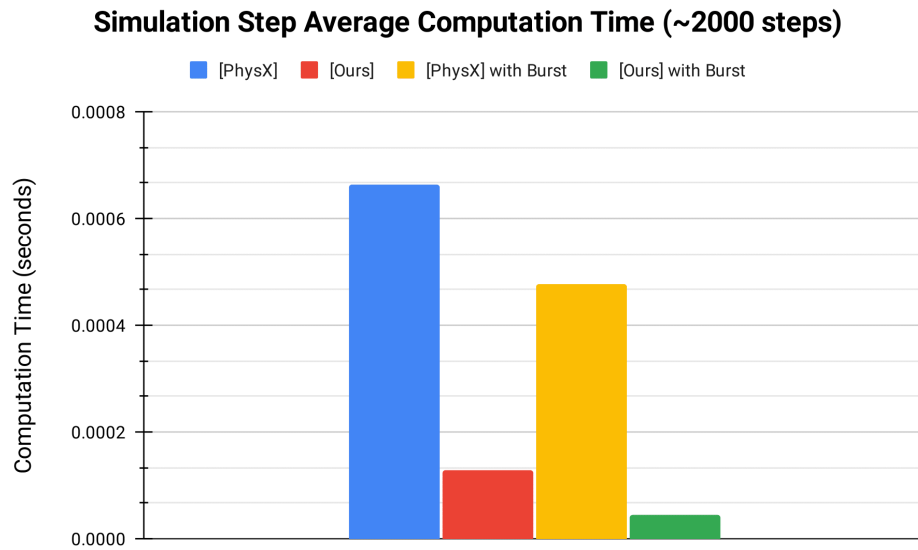


Figure 6.29: Comparison of the average time to compute a simulation step by PhysX’s object-oriented and my data-oriented implementation of Featherstone’s algorithm*.

*In the case of PhysX, Featherstone’s algorithm was not measured in complete isolation due to technical limitations and a lack of time. However, the testing scene contained only the two kinematic chains necessary for the test to keep the comparison as fair as possible by preventing PhysX from simulating the motion or collisions of any unrelated bodies.

On the other hand, it was out of the scope of this thesis to do almost any optimization of the data-oriented implementation and the focus was mainly on getting a first working prototype. Therefore, there may be a lot of room for improvement.

Chapter 7

Discussion and Future Work

The main focus of this thesis was to assess the feasibility, performance and accuracy of implementing Featherstone’s algorithm using a data-oriented design facilitated by the ECS framework within Unity’s Data-Oriented Technology Stack. The performance testing revealed a clear advantage of the data-driven approach over the object-oriented alternative. As expected, the data-oriented implementation outperformed the object-oriented counterpart, showing performance gains of up to tenfold. This performance improvement is likely due to the exploitation of efficient data storage and access patterns maximising cache efficiency along with vectorization of instructions.

However, it is evident that there is a disparity in the accuracy of the data-oriented implementation and the implementation of PhysX which becomes visible in the simulation of a double pendulum. Even after extensive debugging and comparisons of the data-oriented implementation of the pseudo-code in literature and the open-source PhysX code, it remains to be discovered why the data-oriented implementation produces different results from PhysX and more importantly results that are not realistic nor believable, at least at the standard time step used by PhysX in Unity (1/60 of a second). The collected data may give multiple clues as to where the issue may be.

Besides improving the accuracy there are many natural extensions that could be made to the data-oriented implementation. Little time has been devoted to optimising the code so there are likely many performance gains which can be tapped into. In terms of features and functionality, the algorithm can be extended from kinematic chains to kinematic trees and even ones with a floating base as described by Mirtich [1]. Another important feature that could be added for interactive simulations is contact handling and collision detection. Since Unity’s DOTS is used, concurrency can be easily and safely exploited for simulating multiple kinematic trees in a scene in parallel by leveraging the job system.

In terms of future testing opportunities, it would be valuable to measure the performance and accuracy of the implementation on motorised articulated bodies such as robotic manipulators. The current implementation does support motorised joints, but exerting non-zero torques by the joint motors has not been formally tested.

Another interesting test would be to measure how the performance scales with the length of the kinematic chain and with the number of kinematic chains within one scene. Especially, there could be a big difference in performance scalability if the simulation jobs were run in parallel on different threads as by exploiting the power of Unity’s job system.

Chapter 8

Conclusion

Within this thesis, a first-of-its-kind purely data-oriented implementation of Featherstone’s algorithm was developed in C# leveraging Unity’s Data-Oriented Technology Stack. The performance and accuracy of the data-oriented implementation were compared to PhysX 4.1, a popular physics engine integrated into Unity. This was evaluated in a series of seven accuracy tests and one performance test. Accuracy was favoured in the testing process as it was deemed a higher priority to check the correctness of the implementation before engaging in further optimisation and performance tests. Testing showed that by simulating single pendulums with different initial angular displacements the accuracy was almost identical to PhysX. However, when double pendulums were tested with a time-step of $1/60$ of a second the mechanical system gained energy over time instead of maintaining constant energy. The accuracy of the double pendulum tests was visibly lower and the resulting motion was different from PhysX. A lower time step of $1/240$ of a second significantly improved the accuracy and stability of the simulated double pendulum. The lower accuracy of the double pendulum tests is likely due to a small error in the implementation. It was expected that a data-oriented implementation could perform better than an object-oriented one. However, the measured performance of the data-oriented implementation surpassed expectations and proved to be ten times faster than PhysX’s simulation step. This indicates that this work is worth continuing as a competitive solution could be on the horizon.

Besides developing a data-oriented implementation of Featherstone’s algorithm this thesis provides extra details about the algorithm to help successors understand and implement this algorithm. Furthermore, the data-oriented paradigm was introduced along with various resources for further reading. Future work could focus on locating the error in the implementation using the collected data to address the accuracy issues. The implementation can also be extended from kinematic chains to kinematic trees and even ones with a floating base quite easily. Incorporating a response to collisions is also an interesting area for future exploration. Future work could also focus on testing how the performance scales with a higher number of links and kinematic chains in a scene. Tests involving non-zero joint actuator torques would be interesting to see as these are currently supported but were not formally tested.

Bibliography

- [1] MIRTICH, B. V. *Impulse-based Dynamic Simulation of Rigid Body Systems*. 1996. Dissertation. University of California, Berkeley.
- [2] ALTED, F. Why modern CPUs are starving and what can be done about it. *Computing in Science & Engineering*. 2010, vol. 12, no. 2, p. 68–71. Publisher: IEEE.
- [3] FEATHERSTONE, R. *Rigid Body Dynamics Algorithms*. New York: Springer, 2008. ISBN 978-0-387-74314-1 978-0-387-74315-8.
- [4] BARAFF, D. Linear-time dynamics using lagrange multipliers. In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 1996, p. 137–146.
- [5] BENDER, J., ERLEBEN, K., TRINKLE, J. and COUMANS, E. Interactive Simulation of Rigid Body Dynamics in Computer Graphics. In: CANI, M.-P. and GANOVELLI, F., ed. *33rd Annual Conference of the European Association for Computer Graphics, Eurographics 2012 - State of the Art Reports, Cagliari, Sardinia, Italy, May 13-18, 2012*. Eurographics Association, 2012, p. 95–134. DOI: 10.2312/conf/EG2012/stars/095-134.
- [6] EDE, A. van. *ENCODE: From Object-Oriented to Data-Oriented, an Automatic ECS Conversion Designer and Education Tool*. 2021. Dissertation. Utrecht University.
- [7] EFNUSHEVA, D., CHOLAKOSKA, A. and TENTOV, A. A Survey of Different Approaches for Overcoming the Processor-Memory Bottleneck. *International Journal of Computer Science and Information Technology*. 2017, vol. 9, no. 2, p. 151–163.
- [8] ERLEBEN, K., SPORRING, J., HENRIKSEN, K. and DOHLMANN, H. *Physics-Based Animation*. Charles River Media Hingham, 2005.
- [9] FABIAN, R. *Data-oriented design*. 2018. ISBN 9781916478701.
- [10] HUBBARD, P. M. Interactive collision detection. In: *Proceedings of 1993 IEEE Research Properties in Virtual Reality Symposium*. IEEE, 1993, p. 24–31.
- [11] LLOPIS, N. Data-oriented design (or why you might be shooting yourself in the foot with OOP). *Game Developer Magazine*. 2009, vol. 16, no. 8.
- [12] NYSTROM, R. *Game programming patterns*. Genever Benning, 2014. Available at: <http://gameprogrammingpatterns.com/contents.html>.
- [13] ACTON, M. *CppCon 2014: Mike Acton „Data-Oriented Design and C++“*. 2014. Available at: <https://www.youtube.com/watch?v=rX0ItVEVjHc>.

- [14] *Basics of SIMD Programming*. Available at:
<http://ftp.cvut.cz/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingTutorial/BasicsOfSIMDProgramming.html>.
- [15] *Caching*. Available at:
<https://cs.brown.edu/courses/csci1310/2020/assign/labs/lab4.html?spm=a2c65.11461447.0.0.47497f654zfsJD>.
- [16] *Unity Entities Package Documentation*. Available at:
<https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/index.html>.