# POLITECNICO DI TORINO

Master of Science in Cinema and Multimedia Engineering


Master Thesis

# Ragdoll Matching: a non-learned physics-based approach to Humanoid Animation applied to VR Avatars

Advisors

Prof. Nuria PELECHANO

Prof. Andrea BOTTINO

Dr. Jose Luis PONTON

Prof. Francesco STRADA

Candidate

Mattia CACCIATORE

March 2024

**Abstract**

The proliferation of Virtual Reality technologies has intensified the quest for enhanced immersion in virtual experiences. In this context, the simulation of the user's body in the form of a credibly animated Avatar, together with the care in ensuring expected and plausible interactions with the virtual environment, are both crucial factors that contribute to preserving a solid sense of presence in the user experience.

This study presents a potential approach to achieve a real-time physics-driven humanoid character animation, implemented in the Unity game engine. We designed and developed a solution that permits to control a physics Ragdoll by providing as reference a target rig animated with a kinematic-based technique. By doing so, the animated character is able to perform desired movements while being subjected to the physics engine, allowing collisions and interactions with the virtual environment. Motion Matching was identified as a well suited animation technique, as it satisfies the need for realistic humanoid movements even when applied to VR avatars. The focus during the design process was to maximize versatility and interoperability of the tools developed. The control policy that pilots the physics Ragdoll is based on simple automatic control techniques: compared with other more recent solutions that rely on machine or deep learning techniques, this approach eliminates the need for a training phase for the physics simulation.

Our strategy prioritizes modularity and versatility over perfection of results in a well-known and controlled context, while also maintaining the possibility for independent future improvements to each of the simulation modules.

# Ringraziamenti

Giunto alla fine di un percorso durato circa 5 anni e mezzo, sento in dovere di spendere delle parole di gratitudine nei confronti di persone speciali, che hanno permesso che questo periodo della mia vita prendesse forma nel migliore dei modi.

Al Terzo Piano del Collegio Einaudi, sezione Crocetta. Un calderone di menti tanto brillanti quanto vivaci. Tutti insieme abbiamo condiviso lo stesso percorso, ma ognuno con le caratteristiche della propria persona ha contribuito a renderlo più interessante, stimolante, e anche più leggero. Grazie per avermi permesso di vivere giorni mai vuoti (sia nel bene, che nel male), di non avermi fatto sentire mai la solitudine anche a 1000km da casa. Grazie a chi si è reso disponibile, a chi mi ha permesso di imparare, di avere un punto di riferimento. E grazie a chiunque in questi anni abbia dato anche a me la possibilità di essere un riferimento. Spero di esser riuscito a contribuire alla vostra esperienza almeno come voi lo avete fatto nella mia.

Ad Andrea, Valerio, Marco, Giuliana, Federica, Claudia, Le Vecchiette di Paese. Ero una semplice matricolina, eppure dopo neanche qualche mese che ci siamo conosciuti, mi avete dato la possibilità di prendere parte a questo stretto legame che dura ancora oggi. Ricordo il giorno in cui decideste di aggiungermi al gruppo telegram: il fatto che in tanti anni di persone in collegio, di cui alcuni vissuti anche a distanza, io sia stato l'unico a prendere parte alla cerchia così stretta (Giuli perdonami ma in questo caso ti definirei non classificabile :P), ha sempre significato molto per me. E non per una questione di elitismo, sia chiaro, ma perché forse avete visto qualcosa che vi ha spinto a voler stringere un legame con me. Grazie per essere stati voi i miei punti di riferimento, grazie per aver condiviso gioie e dolori di anni di studio, grazie per essere ancora qui anche dopo la fine del percorso, grazie per accettarmi anche quando sparisco, faccio ritardo o sono poco partecipativo. Anche in questo caso, spero in questi anni di avervi lasciato anche io almeno un pezzettino di quello che mi avete lasciato voi.

Ad Elia, l'ultima scoperta del Terzo Piano, l'erede della mia stanza. E' stato difficile non notare la tua personalità, ma soprattutto quanto in realtà avessi da dire dentro. Grazie per avermi dato la possibilità di ascoltarti, di rivedere piccole parti di me ad anni di distanza. Grazie per avermi dimostrato che quelli del nord hanno in fondo anche qualcosa del sud. Permettimi di lasciarti qui due parole per te: nessuno è speciale, c'è solo chi ha la fortuna di avere la strada più liscia di altri. Inutile dire frasi fatte tipo "se ce l'ha fatta uno ***** come me possono farcela tutti". Piuttosto ti dico, cerca sempre di alimentare il tuo motore con ciò che ti fa stare bene, e percorri una strada che ti renda felice.

Agli amici di Brindisi. Anche se sono stati anni in cui abbiamo visto cambiare tante cose, le serate sull'Isola a 1000km da casa sono rimaste una costante sempre presente. Grazie per esser stati un porto sicuro per compagnia, chiacchiera e gioco.

A mia cugina Marta e Alessandro. Grazie per aver portato in questi ultimi anni un pezzetto di casa qui su a Torino, e grazie per la disponibilità e l'aiuto che mi avete concesso nei momenti di bisogno.

Ad Alessia, la persona che dopo oltre 8 anni è ancora qui al mio fianco. Gli anni di liceo, uno spostamento a 1000km da casa, la nuova vita in una grande città, gli scogli del percorso universitario, le camminate nella sera per rincasare, la p andemia, i mesi a distanza, le evoluzioni più disparate dei rapporti tra i nostri amici, i momenti bui, i traguardi, la nuova casa. Tutto insieme. Credo sia impossibile per me immaginare chi sarei in questo momento senza averti avuto vicino. Probabilmente una persona peggiore. Penso che tu sia una delle poche persone che può avere idea dell'impegno che sto provando a metterci nello scrivere queste parole. Grazie per continuare a insegnarmi a gioire delle cose semplici, a provare entusiasmo per le cose belle, a capire cosa significa tenere alle persone, a guardare le proprie debolezze. Credo di avere ancora tanto da imparare, non so nemmeno se e come ci riuscirò, ma spero che continuare a osservarti mi potrà aiutare. Per quello che sei e per quello che hai vissuto e vivi, io ti auguro tutto l'amore e la felicità di questo mondo, e di volare in alto con i tuoi sogni genuini. E soprattutto spero di continuare a contribuire come posso alla felicità della tua vita standoti accanto. In questo pezzo di carta oggi ci sei anche tu, quindi ancora una volta, grazie.

A Mamma e Papà. Io non riesco ancora oggi ad abituarmi all'idea di quanto io sia fortunato ad avere due genitori come voi. Se oggi sono qui, con questo percorso alle spalle, con questo lavoro, con delle ambizioni, è soltanto merito di tutto quello che avete fatto per me, da sempre. Grazie per aver creduto in qualsiasi mio interesse, nonostante potesse essere anche distante dai vostri o difficile da comprendere. Grazie per avermi dato la possibilità di esplorare questi interessi, così

che le mie passioni potessero prendere forma. Grazie per avermi lasciato libero di percorrere la mia strada. Grazie per essermi sempre stati vicino a spianare quella strada, in qualsiasi modo vi fosse possibile, a qualsiasi costo e sacrificio. E grazie perchè continuate tutt'ora a fare tutto ciò che ho citato. Vi posso assicurare che in questi anni ho avuto modo di constatare quanto tutto ciò sia ben lungi dall'esser considerabile scontato o ovvio. Agli occhi delle persone che mi circondano siete un esempio di cui vado così fiero da arrivare talvolta a sentirmi quasi in difetto al pensiero che altri potrebbero non avere avuto la mia stessa fortuna. Mi auguro solo di riuscire a far germogliare sempre tutto ciò che seminate per me. Che questo pezzo di carta possa regalarvi uno di quei germogli.

A tutta la mia famiglia. Grazie per avermi fatto sempre percepire un clima di supporto e fiducia, nonostante le distanze e la mia disattenzione nel farmi sentire.

# Table of Contents

# List of Tables

# List of Figures

# Acronyms

| | |
|---|---|
| UPC | Universitat Politècnica de Catalunya |
| VR | Virtual Reality |
| HMD | Head-mounted display |
| CGI | Computer-generated imagery |
| FSM | Finite state machine |
| AI | Artificial intelligence |
| Deep RL | Deep Reinforcement Learning |
| XR | Mixed Reality |
| PID | Proportional-Integral-Derivative |

# Chapter 1

# Introduction

## 1.1 Problem overview

With the proliferation of VR (Virtual Reality) technologies in the consumer market, the quest for enhanced immersion has become paramount.

One of the key challenges is the effective simulation of the user's entire body as a VR avatar. The evolution of the technology behind VR headsets is following a path that keeps the user's comfort at the center of the experience design. In the past, solutions proposed by pioneering devices in this field, such as the first HTC Vive, aimed to provide a potentially uncompromised experience but required the establishment of a complex and dedicated ecosystem (a powerful PC to run the software, an HMD (Head-mounted display), two controllers, a wired connection, external cameras, optional sensors for full-body tracking, etc.). Today, while elite headset models continue to cover a market segment, the ability to develop stand-alone devices has reinstated the importance of thinking about accessible, ready-to-use, and easily transportable products.

Nowadays, a VR headset aims to consist of only three elements: an HMD and two controllers for both hands. This setup allows tracking the position and orientation of three parts of the user's body: the head and both hands. With this information, it is possible to effectively map the movement of the upper body in the user's virtual avatar. When the goal is to simulate the movement of the entire user's body, the main challenge arises in faithfully reconstructing the movement of the lower body, which lacks of dedicated tracking devices.

When it comes to maintain a credible VR experience without undermining the suspension of disbelief, another crucial factor lies in the user's ability to interact

realistically with the virtual environment. In this regard, the challenge consists in ensuring that each user action corresponds to a reaction within the simulated system, preventing undesirable occurrences such as interpenetrations between the mesh of user's avatar and those of the virtual environment. This seamless interaction not only enhances the sense of immersion but also contributes to the overall authenticity of the VR experience.

## 1.2 Objective

This project aims to propose a possible approach to achieve realistic human body movement simulation, while ensuring that the user will be able to physically interact with the virtual environment. This solution is specifically designed to promote versatility and interoperability, in order to be adopted by developers and integrated in different possible scenarios.

The basic concept guiding our approach involves selecting a real-time animation technique that doesn't rely on physics, to accurately replicate human body movements on a rigged avatar. This kinematic-based animation serves as target for controlling a "virtual puppet", which runs on the physics engine. Unlike other recent related works, we aim to avoid adopting machine/deep learning techniques for the physics control policy. Instead, we tested the use of simple automatic control techniques: this approach eliminates the need for a training phase for the physics simulation, making the tool more versatile and ready-to-use as it remains independent from any specific training dataset.

In this specific case, the designated animation technique chosen for achieving accurate body movement is Motion Matching. We aim to test this implementation on a VR application, in order to observe the resulting plausibility of body movements, as well as and the effectiveness of contact interactions with the environment.

## 1.3 Document structure

This thesis is organized into several chapters, with the main topic of describing the solution proposed. The central content of the document is contained in Chapter 3 and Chapter 4, which respectively discuss the design and the implementation of the tools developed.

The following list summarizes the contents of each different chapter:

- **1 Introduction** outlines the problem statement and research objectives.
- **2 Background** reviews related work in real-time animation, data-driven

techniques, physics-aware avatars in VR, and AI approaches to motion simulation. Preliminary definitions related to the Unity Engine framework, Ragdoll physics, and PID control are also discussed.

- **3 Active Ragdoll Module: System Design** delves into the design of the proposed solution called Active Ragdoll Module, detailing elements such as the Ragdoll Maker and Ragdoll Controller.
- **4 Implementation** covers the technical details of implementing the Active Ragdoll Module, including code architecture and the developed Unity components.
- **5 Combining Active Ragdoll with VR Motion Matching** explores the integration of the Active Ragdoll Module with VR Motion Matching, with a brief experiment on XR Hand Tracking.
- **6 Discussion and Results** analyzes the performance and outcomes of the Active Ragdoll Module, along with its application to VR Motion Matching.
- **7 Conclusions** summarizes the results of the research and suggests potential avenues for future work.

# Chapter 2

# Background

## 2.1 Related work

This section will provide the reader some paragraphs regarding background themes and information derived from the literature, establishing a context for the work presented in this thesis.

### 2.1.1 Real-time Animation

When developing real-time graphics applications (e.g., video games, VR applications, simulators), it is important to underline the distinction between traditional Computer Animation and Real-time Animation techniques.

Creating a product of CGI (Computer-generated imagery) such as an animation film or video, involves a process aimed at faithfully reproducing the designed concept and aesthetics, whether it should look like an inspired cartoon or like a photorealistic simulation. When the animation preview matches the desired result, the whole sequence needs to be rendered in order to obtain the final asset: each frame of the sequence takes an amount of time to be rendered, depending on the complexity of what the frame contains (e.g., number of polygons, complex materials, physical accurate simulations).

In contrast, the development of a real-time graphics application introduces a critical constraint. As the system output should be updated at runtime to dynamically react to the user input, a single frame needs to be rendered within a fraction of second. To meet a target frame rate of 60fps, one frame must be rendered in at most 16,7 milliseconds. This requirement underscores that it is crucial to analyze the performance impact of the techniques employed in the simulation.

Consequently, while traditional CGI allows to pre-render an extremely fine-tuned motion capture animation for a specific sequence, or to bake an accurate physics-based simulation of a deformable body (assuming the availability of computational power), a real-time scenario necessitates to find the best compromise between result effectiveness and performance impact of chosen techniques, including animation methods.

### 2.1.2 Data-driven techniques: Motion Matching

One of the most common approaches when dealing with real-time animation is to think the animation process as a FSM (Finite state machine). Consider a character in a third-person video game: based on user inputs and game logic parameters, it must perform certain movements and actions (e.g., standing still, walking, running, jumping). An animation approach based on a FSM involves modeling a graph where nodes represent the character's logical states (Figure 2.1). Each node is associated with a properly configured animation, played only when the character is in that state. The edges of the FSM graph indicate transitions from the current state to subsequent states. Often, each edge is associated with a condition controlled by the game logic: if met, the transition to the connected state is triggered.

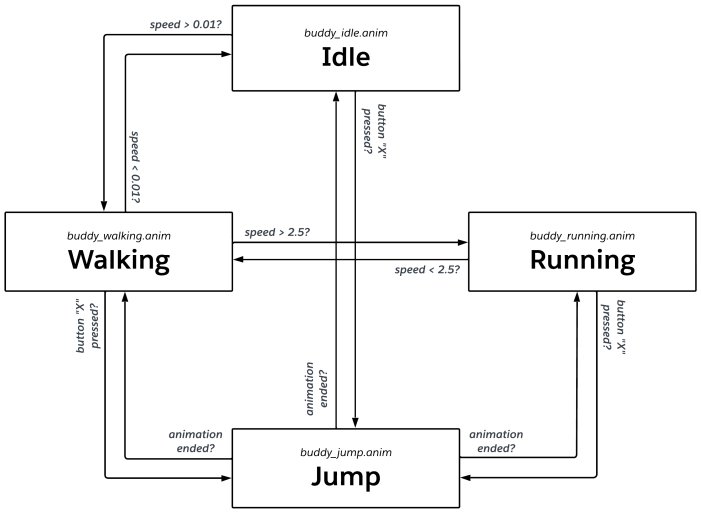



**Figure 2.1:** Example concept of an animation FSM graph

While this approach is straightforward and effective for various types of gameplay and different ranges of stylized aesthetics, the complexity of such a graph quickly escalates when the goal is to achieve realistic and natural rendering of complex movements.

One possible alternative solution was discussed in 2016 by Clavet [1]. In response to this issue, he proposed a totally different, data-driven approach that is not dependent on the character's logical state: Motion Matching. This technique utilizes a set of pose data, such as those obtained from motion capture sessions, to identify the most suitable pose for the next frame. To accomplish this, a selected set of bones and/or skeleton properties (e.g., feet positions, velocities, local and future orientation) are combined to create a feature vector, which is assessed as a measure for pose matching. On each frame, an algorithm searches through the pose data to find the pose that minimizes the error between its feature vector and the one predicted to be optimal for the next frame.

Motion Matching is a powerful animation technique because it decouples the animation process from the game/application logic. The quality of the result depends solely on the features chosen to be considered and the quality and quantity of the dataset provided.

In 2022, Ponton, Pelechano et al. [2] from UPC (Universitat Politècnica de Catalunya) experimented with the application of Motion Matching for animating full-body VR avatars. Their goal was to reconstruct a convincing human motion simulation using only the three trackers available on most consumer VR headsets (two controllers and the HMD).

Under the supervision of Ponton and Pelechano, as will be detailed later, the work proposed in this thesis will precisely employ VR Motion Matching as a reference animation technique for the physics simulation of our VR avatar.

### 2.1.3 Embodying Physics-aware Avatars in VR

Following the definitions elaborated by Caroux [3] in the context of video games, one of the key aspects in creating satisfying and pleasing VR experiences is the need to ensure users a strong and lasting sense of engagement. Two fundamental aspects contribute to this result:

- User immersion, defined as "a psychological state characterized by perceiving that one is involved, included, and interacting with an environment that provides a continuous stream of stimuli and experiences" (Stanney and Salvendy [4]).
- User presence, defined as "the subjective experience of being in one place or environment even when one is physically located in another" (Stanney and Salvendy [4]). This latter aspect, according to K.M. Lee [5], can be further subdivided into:
  - Spatial presence, linked to virtual objects and environments.

- Social presence, linked to virtual "social actors."
- Self-presence, linked to the virtual self representation.

Our work aims to act on the user's VR avatar representation and on its relationship with the virtual environment, exploring new development tools and solutions with the goal of empowering both user self-presence and spatial presence.

A recent paper [6] investigated the effect on users of using physics-based VR avatars. The study's focus was on determining which user movement remapping approach yields a greater sense of embodiment.

On one hand, ensuring a one-to-one mapping of user real movements onto their avatar leads to possible unnatural inconsistencies when interacting with the virtual environment. When a virtual object intersects with an avatar's movement, if that movement is still possible for the user because there are no real obstacles in the space that prevents it, then a one-to-one mapping approach would still aim to preserve the user's real movement. This creates an unnatural condition in the virtual environment, such as object penetration with the avatar's body.

On the other hand, subjecting the avatar 3D model to the action of the physics engine allows for collision detection with other virtual objects, increasing the possibilities for interaction with the virtual environment. However, this results in breaking the one-to-one mapping of user movements: the avatar is indeed constrained by potential obstacles or interactions with virtual objects that do not necessarily have a counterpart in the real space where the user is located. This leads to a possible discrepancy between the user's real pose and the VR avatar's pose.

The test results from the study demonstrate that users perceive a greater sense of presence in embodying physics-aware avatars, provided that the discrepancy between the user's real pose and their avatar's pose is contained within certain limits, in accordance with the phenomenon of the "self-avatar follower effect" [7].

In line with the context described, our objective supports the research and development in the direction of refining more advanced techniques for the simulation of physics-based VR avatars.

### 2.1.4 AI-based approach to motion simulation

Ensuring immersive VR experiences comes with the need to reproduce the user's movements with the data provided by the limited number of sensors available on the HMD. Furthermore, consistently with the conclusions of Section 2.1.3, we want this reproduced motion to be compliant with physics laws.

With the advancement of modern AI (Artificial intelligence) techniques, one of the most common approaches to tackling such a problem involves the use of a motion control policy driven by Deep RL (Deep Reinforcement Learning), which is trained to minimize the position discrepancy between the user and the simulated avatar.

Peng et al. [8] presented DeepMimic, showing that various sets of motion data can be used to train a control policy capable of imitating those movements in a physics-driven context. Following a similar base approach, Ye et al. [9] with Neural3Points presented a learned data-driven physics based method for predicting user's full-body movement starting from the sparse sensors of a VR headset, and simulating an avatar that mimics the predicted pose in real-time. Later on, Lee et al. [10, 11] proceeded this research developing QuestEnvSim, which implements a Deep RL control policy that combines VR sparse sensors, physics simulation and environment observation to achieve an environment-aware full-body avatar capable of interacting with the surrounding objects.

All of these works and related ones bring up very convincing results, but they all share a common aspect that, from certain perspectives, could be seen as a limitation: they require a training phase for the physics simulation. This implies that the result significantly depends on the data consumed by the control policy to learn its behavior. Depending on the model architecture, a Deep RL control policy may perform extremely well in a vast variety of cases that share some similarities with the training data. However, an important question arises: how well will the model continue to perform when attempting to expand the training dataset to include a greater variety of situations? Furthermore, increasing the size of the dataset leads to presumably longer training phases, together with the need for more memory to store it.

Let's examine an approach similar to QuestEnvSim[11]: to enable interaction with the surrounding virtual environment (e.g., walls, obstacles), the training phase must be aware of data related to that environment. Moving the character to a completely different scene necessitates at least partially repeating the training phase with data collected from the new environment.

While most industries driving innovation in artificial intelligence are well-suited and familiar with this workflow, the creative nature of the game and VR industry demands approaches that prioritize flexibility, optimization, customization and ease of use to promote rapid and reliable prototyping and development.

In line with what has been said, for the purposes of this research, we have chosen to proceed with a non-learned approach. As will be seen in Chapter 3, the design of our proposal is very similar to that presented by Llobera and Charbonnier [12],

with the difference that the control policy employed does not use Deep RL but is built as a control system based on simple techniques derived from the theory of automatic control (2.2.3).

## 2.2 Preliminary definitions

### 2.2.1 Unity Engine framework

Unity Engine is a powerful and versatile game development platform renowned for its ease of use, flexibility, and robust capabilities. Developed by Unity Technologies, it provides developers with a comprehensive suite of tools and features for creating real-time interactive 2D, 3D, and XR (Mixed Reality) experiences across various platforms. Unity offers developer a well structured game development process by offering an intuitive interface, a vast asset store, and a wealth of documentation and tutorials. With its cross-platform compatibility, developers can deploy their creations seamlessly to multiple platforms, including PC, consoles, mobile devices, XR systems and web browsers. The Unity Engine's rich ecosystem and extensive community support make it a preferred choice for both indie developers and large game studios worldwide.

The core architecture of the engine is centered around the concept of GameObject. A GameObject is the fundamental building block of a virtual scene: it acts as a "container" of different components. A component is a functional piece that can be attached to a GameObject, and defines a specific behaviour or functionality. Every GameObject comes with a default component called `Transform`, which controls and stores information about the spatial properties of the object, determining its location, orientation and size relative to its parent GameObject and to the world origin.

Developers are able to define completely personalized behaviours by creating custom components implemented using C# code. A C# script containing a class which inherits from Unity's `MonoBehaviour` can be instantiated as a component attached to a GameObject. Every component is displayed in the editor throw its Inspector, which exposes serialized fields of the component's class that can be tweeked and edited directly from the editor.

Unity comes equipped with its own real-time physics engine, which is based on NVIDIA's PhysX. Developers have access to a collection of pre-made physics-based components to subject GameObjects to the physics simulation. The base component responsible for imparting physics properties to a GameObject is the `Rigidbody`. This component enables objects to react to external forces such as gravity, collisions, and user interactions. Its attributes, including mass, drag, and

constraints, can be adjusted dynamically to fine-tune the object's behavior and achieve the desired result at runtime.

With a brief introduction to some concepts regarding the Unity framework, this section provided readers with a better understanding in preparation for the discussion on the project implementation proposed for this thesis.

### 2.2.2 Ragdoll physics

Ragdoll physics [13, 14] represents a procedural approach to character motion simulation within virtual environments. Unlike traditional kinematic-based animation methods, Ragdoll-based animation introduces a level of realism by physically simulating the movement of a character's articulated skeletal structure in response to external forces.

At its core, a Ragdoll system relies on the physics engine to treat each bone in the character's rig as an independent rigid body, interconnected by joints that mimic the anatomical constraints of the human body. This approach favors more organic responses to environmental stimuli, such as collisions or gravitational forces.

This technique is frequently used as a substitute for conventional static death animations in video games and animated films, providing a more authentic representation of how the character model should react to collisions, gravity and external forces. It also proved to be a key tool in designing specific gameplay mechanics for certain video games (e.g., Gang Beasts[15], Human Fall Flat[16], Party Animals[17]) which have made Ragdoll Physics-based character movement the core of their game design.

### 2.2.3 PID control

PID (Proportional-Integral-Derivative) control is one of the most common approach when it comes to feedback control methods. Its definition has been extensively covered by literature related to the field of automatic controls [18, 19, 20].

The idea behind this control method lays on the feedback principle. Figure 2.2 shows the block diagram representation of a traditional feedback control system. The process represents the entity that has to be controlled, and *y(t)* is the process variable.

Controlling the system means ensuring that the process variable $y(t)$ follows a reference value $r(t)$. To achieve this purpose, a controller module is introduced upstream of the process: it takes the error between the reference and the process variable $e(t) = r(t) - y(t)$ as input, and generates a manipulated variable $u(t)$ based on a specific computational rule. The manipulated variable $u(t)$ is then

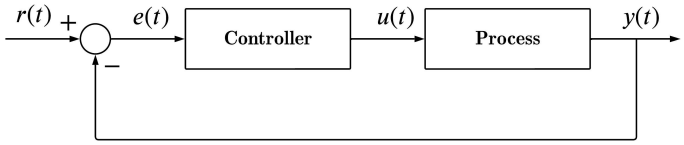



**Figure 2.2:** Traditional feedback control system block diagram

passed as input to the process, which will generate the next $y(t+1)$ value to feed back into the loop: it will be used to calculate the error $e(t+1)$ and produce the new manipulated variable $u(t+1)$.

PID control is a computational rule that can be implemented into a controller (called PID controller). It produces an output that can be described as three separate terms:

1. P (proportional) term: the "present" error. It is proportional to the error at the instant $t$.
2. I (integral) term: the accumulation of the "past" errors. It is proportional to the integral of the error up to the instant $t$.
3. D (derivative) term: the prediction of the "future" error. It is proportional to the derivative of the error at the instant $t$.

The complete analytic expression of the manipulated variable can be written as follows:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{de}{dt} \tag{2.1}$$

$K_p$, $K_i$ and $K_d$ are respectively called proportional, integral and derivative gain of the PID controller. These parameters must be properly tuned in order to achieve the desired behaviour of the system. Figure 2.3 shows an example of feedback control loop based on a PID controller.

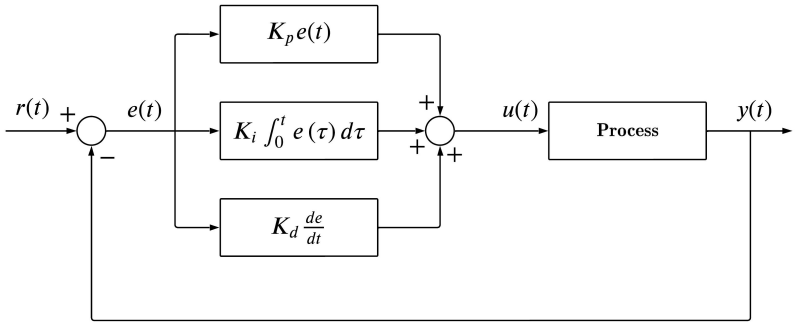



**Figure 2.3:** PID controller system block diagram

# Chapter 3

# Active Ragdoll Module: System Design

Now that a proper background knowledge has been provided to the readers, in the following sections there will be a more detailed description of the proposed solution.

As we briefly exposed in Section 1.2, the core objective is to achieve both plausible humanoid animations and environmental interactions in a VR real-time application. The strategy developed by this project aims to move the actual avatar simulation to the domain of the physics engine, regardless of the technique adopted to animate the avatar (in this case, it is Motion Matching). For demonstration purposes, let's consider a concept pipeline for the real-time animation of an avatar (Figure 3.1).

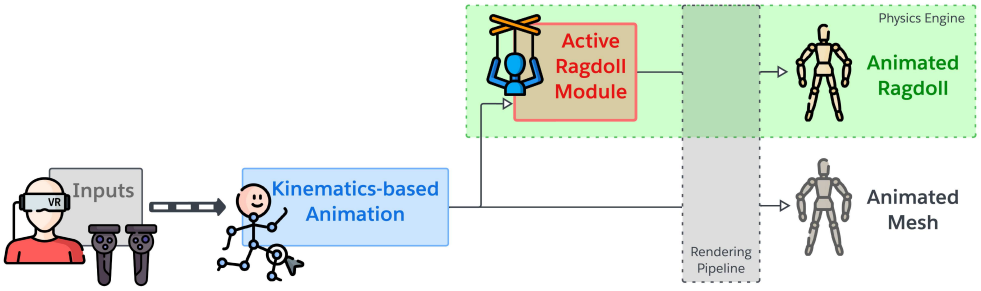



**Figure 3.1:** Real-time avatar animation pipeline concept

Starting from the user's device, the input data are passed to the animation algorithm: this one calculates the geometrical information about the updated position and

rotation of each bone of the avatar 3D model. The output of the animation algorithm is then passed to the rest of the actual rendering pipeline, which will draw the avatar's mesh in the calculated pose.

Our solution can be seen as an additional module that lays between the animation step and the rendering pipeline: it takes the data calculated by the animation algorithm as input, and relying on the physics engine it uses them to properly move a Ragdoll version of the animated avatar. From now on, we will call this module "Active Ragdoll Module".

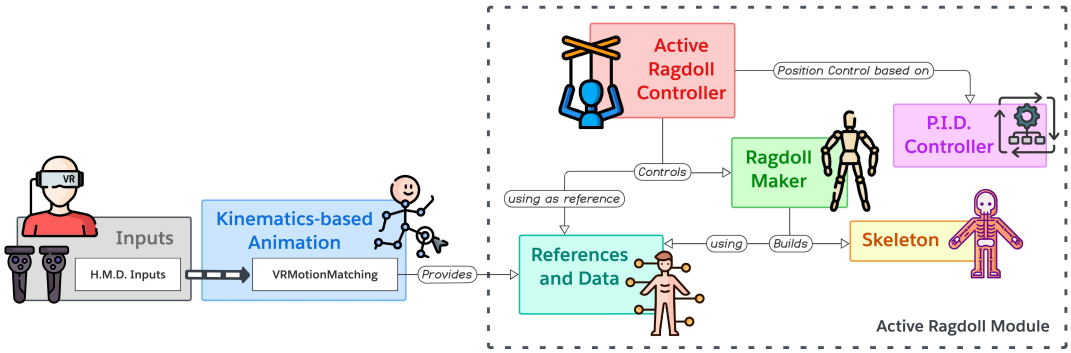



**Figure 3.2:** Active Ragdoll Module: core structure

The core elements of the Active Ragdoll Module are shown in Figure 3.2.

## 3.1 Ragdoll Maker

The Ragdoll Maker element is responsible of building the actual physics ragdoll. Starting from a copy of the avatar 3D model, it takes as input the references of the rig's bones, and a series of structural data and options (e.g., joints constraints information, body mass distribution, rigid body simulation options) to build the Skeleton data structure.

The ragdoll building procedure is not a runtime feature, and it is meant to be performed before the start of the simulation (considering our framework, while working in the Unity Editor).

### 3.1.1 Skeleton structure

The Skeleton is the element that stores all the physics ragdoll information. It is mainly structured as a collection of BodySegments.

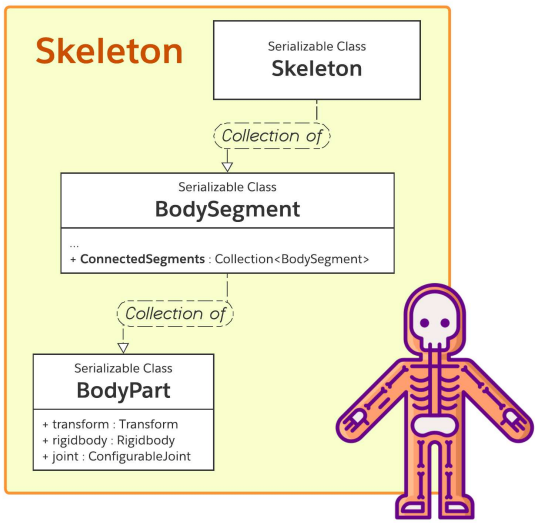



**Figure 3.3:** Active Ragdoll Module: Skeleton structure

A BodySegment is a segment of the human body that can be considered not articulated. To clearly explain this definition, a simple example is provided. Considering a human leg, the knee and the ankle allow a flexible and articulated motion of the art: a leg can't be defined as BodySegment. Instead, considering only the upper part of a human leg (the one corresponding to the femour), there are no joints in the middle of this part that allow to flex it: this section can be defined as BodySegment.

A number of 18 main BodySegments have been considered for the purpose of our implementation:

- Head
- Neck (optional)
- Shoulders (optional)
- Upper arms
- Forearms
- Hands
- Trunk
- Hips
- Upper legs

- Lower legs
- Feet

Two BodySegments can be connected by a joint, which has to be appropriately configured to reproduce the realistic constraints of that particular human body joint.

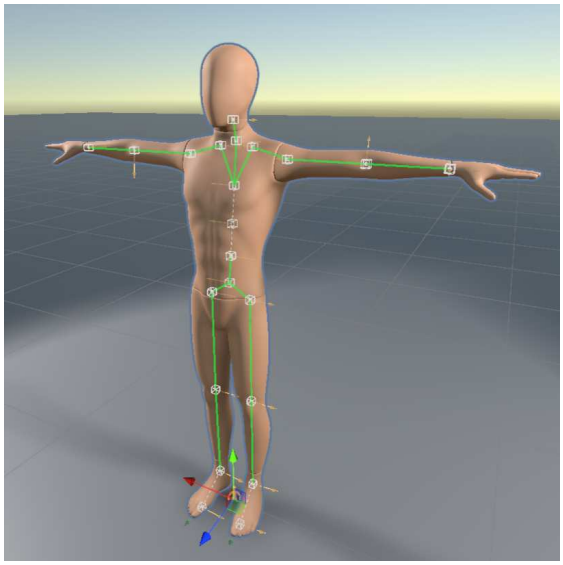


**Figure 3.4:** Visual representation of joint connections: white boxes represent single BodyParts, green lines indicate connections between BodySegments, and white dotted lines represent connections between BodyParts belonging to the same BodySegment.

A BodySegment is, in turn, composed by a collection of BodyParts, connected together with fixed joints. A BodyPart corresponds to a bone of the 3D model rig. It is the most elementary part of the physics ragdoll, and contains references to its rigidbody, its transform and its joint.

When the Skeleton data structure is created, it performs the actual ragdoll building procedure: each bone of the rig, also known as BodyPart, is assigned to a BodySegment and equipped with a Rigidbody and a Joint component. These components are configured appropriately using data and options provided by the Ragdoll Maker.

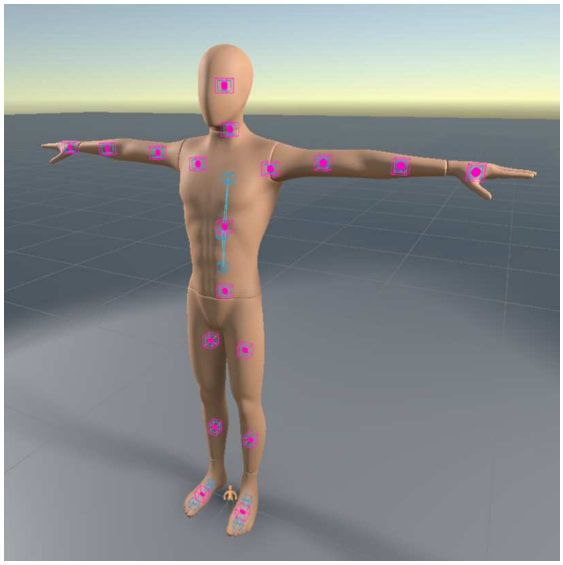


**Figure 3.5:** Visual representation of BodySegments: pink boxes represent the centers of mass of BodySegments, while cyan boxes denote the centers of mass of individual BodyParts. In our example, both the feet and trunk consist of multiple BodyParts.

## 3.2 Ragdoll Controller

The Active Ragdoll Controller element is responsible of controlling at runtime the physics ragdoll created by the Ragdoll Maker.

To achieve this purpose, it acts like a sort of "puppeteer". A copy of the avatar's mesh is animated in the scene using a kinematics-based approach (in this case, Motion Matching): we can call it target rig. The geometrical information about each bone of that rig is used as a reference for the Active Ragdoll Controller: at runtime, it updates the ragdoll's bones rigid bodies to match as closely as possible the position and orientation of the respective target rig's bones.

To make each rigid body follow its respective target, the Active Ragdoll Controller needs to pilot both its position and rotation with the help of the physics engine: that means they have to be controlled through the application of forces and torques rather than through geomteric transformations.

In our system design, rotation control is performed on each BodySegment in order to try matching the target rig local pose. In addition to this, a discrete number of BodyParts have been equipped with position controllers: these allow a further level of control in matching the local pose, while also permetting the ragdoll to follow target rig's macro-movement (for example moving from point A to point B).

### 3.2.1 Position control: PID controllers

In order to guarantee a versatile level of configuration in their behaviour, the position controllers of the Active Ragdoll Module have been designed as PID controllers (2.2.3). This type of control policy allows a straightforward implementation with a considerably low cost in performance, as long as a sufficient range of customization in the desired behaviour by properly tuning its coefficients.

As mentioned earlier, while the rotation is controlled for each BodySegment, only certain BodyParts have been selected to be also position-controlled. These BodyParts include:

- The hips, as they can be considered the main pivot point for tracking the Ragdoll's position in the environment.
- Left and right upper arms, as they could serve as reference points for the upper part of the body. Also, arms are likely to assume more complex poses subjected to gravity, so applying a stronger position constraint could preserve consistency between the target rig pose and the Ragdoll pose.
- Both hands, feet and head, as directly tracking the position of the body's extremities could further improve the quality of the match between the target rig pose and the Ragdoll's one.

# Chapter 4

# Implementation

## 4.1 Code architecture

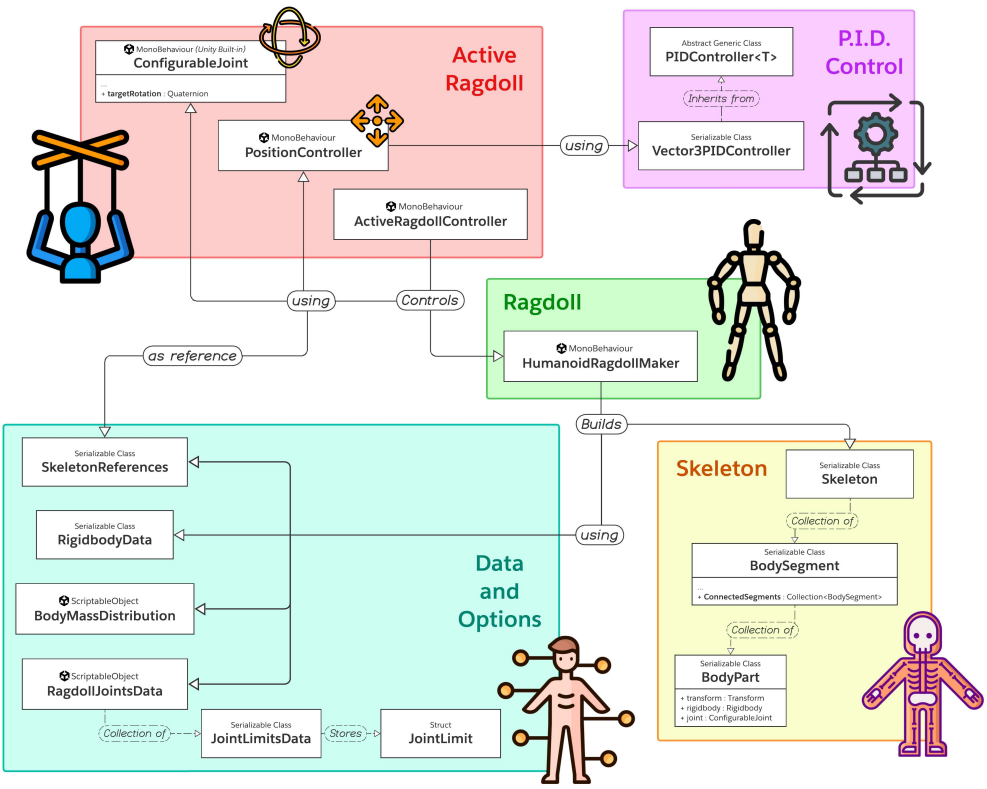



**Figure 4.1:** Active Ragdoll Module: code architecture diagram.

## 4.2 Data and options providers

This section develops a closer look to the main data structures used in the simulation as providers for parameters, options and references. It will also briefly cover how they have been implemented in Unity.

The vast majority of the classes described are marked as Serializable, allowing visual representation in the Unity Editor and consistency between Play mode and Edit mode.

### 4.2.1 Skeleton references

`SkeletonReferences` is a class designed to store the `Transform` references of the main bones of a humanoid rigged 3D model. It inherits from an abstract generic class called `TaggedFieldContainer<TFields,TTag>`, which permits to tag each field of the class with the entry of an `enum`, and retrieve a particular field given its tag. We used the already existing Unity `HumanBodyBones enum` as tag type.

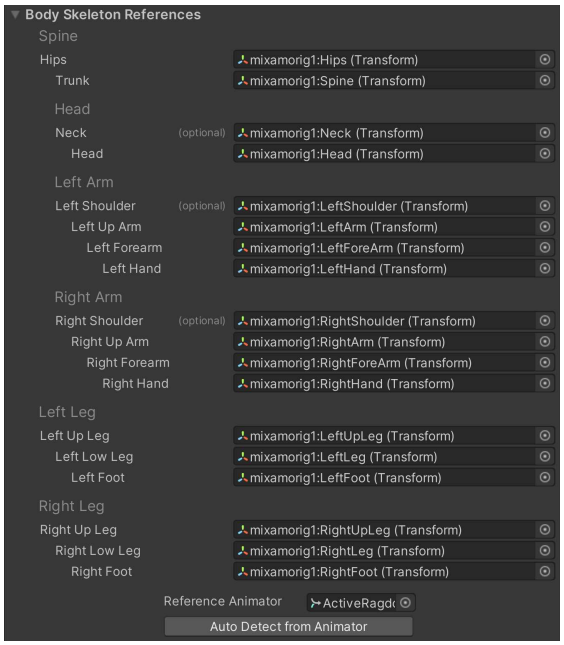



**Figure 4.2:** `SkeletonReferences` custom Inspector

`SkeletonReferences` has 18 `Transform` fields corresponding to the avatar's BodySeg‡ments (according to 3.1.1). To automate the assignment of these references from the Unity Inspector, the class implements a `IAutoDetectable<TDetector>` interface: this allows to auto-detect its fields starting from a specific detector object (in this case, a Unity `Animator`).

### 4.2.2 Ragdoll joints data

`RagdollJointsData` is a Unity `ScriptableObject` designed for setting up all the data concerning Ragdoll joint limits. It exposes a series of fields corresponding to the main BodySegments, each of them of type `JointLimitData`: this serialized class stores parameters for constraints of the joint, including its allowed rotation ranges. The data used in our implementation (Table 4.1) take a cue from previous works [21] and have been empirically adjusted favoring the actual pose matching result, rather than anatomical accuracy.

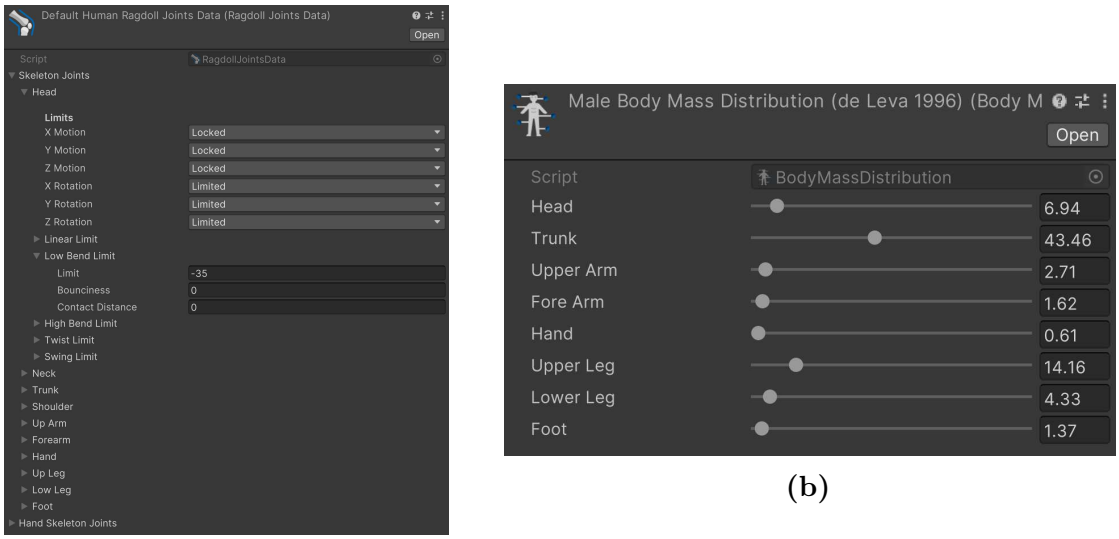
| **Joint** | **Bend** | **Twist** | **Swing** |
|---|---|---|---|
| Head | -35° to 17° | -85° to 85° | -15° to 15° |
| Neck | -20° to 20° | -25° to 25° | -15° to 15° |
| Trunk | -25° to 45° | -25° to 25° | -20° to 20° |
| Shoulder | -12° to 50° | Free | -54° to 54° |
| Upper arm | -85° to 45° | -30° to 30° | -55° to 55° |
| Forearm | -130° to 15° | -60° to 60° | -10° to 10° |
| Hand | Free | Free | Free |
| Upper leg | -65° to 35° | -45° to 45° | -35° to 35° |
| Lower leg | -3° to 150° | 0° | 0° |
| Foot | -25° to 35° | -15° to 15° | -20° to 20° |

**Table 4.1:** Table of joint rotation limits.

The joint rotation axes have been named according to the respective movement of the joint (bending, twisting, swinging). It is worth underlying that the actual joint implementation relies on the Unity `ConfigurableJoint` component: this one permits to specify rotation limits for three axes, but only one of them can have asymmetric degree range. For this reason, the bend axis of each joint has been identified as the primary axis, representing the main movement of that specific joint (for instance, the primary axis of the forearm joint is the one that allows it to rotate from a fully extended position to bending near the bicep). Following the `ConfigurableJoint` convention, the twist axis corresponds to the joint's secondary axis.

### 4.2.3 Body mass distribution

`BodyMassDistribution` is a Unity `ScriptableObject` that stores information about the mass proportion of the main BodySegments. It has 8 float fields (Figure 4.3b) limited between 0 and 100: each field represents the percentage of total body weight that has to be assigned to that respective BodySegment.




**(a)** **(b)**

**Figure 4.3:** `RagdollJointsData` (a) and `BodyMassDistribution` (b) Inspectors.

| BodySegment | Weight % |
|---|---|
| Head | 6.94 |
| Trunk | 43.46 |
| Upper arm | 2.71 |
| Forearm | 1.62 |
| Hand | 0.61 |
| Upper leg | 14.16 |
| Lower leg | 4.33 |
| Foot | 1.37 |

**Table 4.2:** Table of body mass distribution based on de Leva [22].

To be realistically consistent, the sum of all fields (counting symmetric BodySegments as double) has to be exactly 100. For this implementation, the mass

distribution data have been set up based on the study of de Leva [22], as shown in Table 4.2.

## 4.3 HumanoidRagdollMaker component

Setting up a physics Ragdoll could be achieved using components already provided by Unity, which are based on the built-in physics engine. A `Rigidbody` component should be attached to each GameObject that composes an articulated mesh (in this case, a rigged humanoid 3D model). Then, a series of Unity `Joint` type components should be added to connect each articulated part, properly configured to define the respective rotation and position constraints.

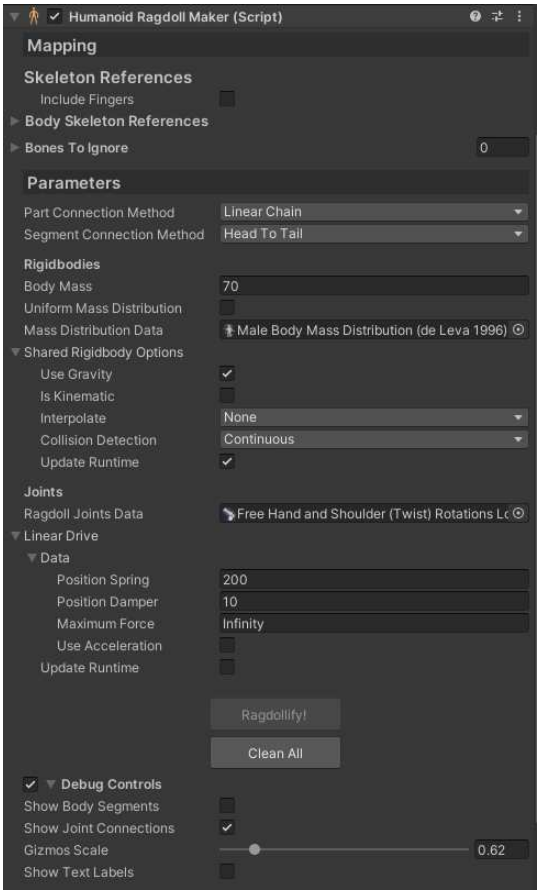



**Figure 4.4:** `HumanoidRagdollMaker` custom Inspector.

If performed manually, this procedure could be long, tedious and prone to errors. To automate it in case of a humanoid rig, we implemented the `HumanoidRagdollMaker` component. It is a Unity `Monobehaviour` designed as a tool for the creation of

humanoid Ragdolls. It is provided together with a custom Inspector (Figure 4.4): the interface allows to properly configure the Ragdoll parameters, and exposes buttons that perform operations in Edit mode, including the actual Ragdoll building procedure.

The component can be attached to the avatar's root GameObject. Since our main objective is not focused on the generation of accurate colliders for a humanoid rig, the avatar model must be provided with colliders already set up for each bone. We created the colliders for our avatars using both Unity's primitive colliders and the Technie Collider Creator 2 tool by Triangular Pixels [23], which allows to create custom mesh colliders inside the Unity Editor.

Once the `SkeletonReferences` (4.2.1) fields have been properly filled, and all the parameters have been set up as desired, clicking the "Ragdollify" button will invoke the `Ragdollify` method of the component class.

The method creates an instance of the `Skeleton` class and stores it in a serialized private field of the component. The `Skeleton` class constructor performs the actual Ragdoll building procedure, creating the whole data structure while, at the same time, populating the rig's GameObjects with `Rigidbody` and `Joint` components. It is important to note that, among all the `Joint` components that Unity provides, we chose to use the `ConfigurableJoint`, as it proves to be the most modular and potentially customizable of them all.

The field `bonesToIgnore` exposes an array which can be populated with bones of the rig that are not meant to be part of the skeleton structure. This feature allows compatibility with more complex or exotic rigs that include for example bones for clothes or body attachments.

### 4.3.1 Ragdoll building algorithm

As mentioned before, the `Ragdollify` method is responsible for initiating the procedure that builds the Ragdoll structure. It performs three main tasks:

- Creates and stores an instance of the `Skeleton` class
- Distributes body mass along the created `Skeleton`
- Applies secondary parameters to every `Rigidbody` and `ConfigurableJoint` (e.g., gravity toggle, collision detection mode, joint's linear drive)

The `Skeleton` class constructor receives a series of parameters set up in the component's Inspector, including `SkeletonReferences` and `RagdollJointsData`, and uses them to execute the real Ragdoll building algorithm.

**Algorithm 1** Ragdoll building algorithm pseudo-code.

**Require:**
- $references$ containing data about rig's bones Transform;
- $root$ Transform of the rig (usually the hips);
- $jointData$ containing data about constraints for each BodySegment.

```
1: function BuildSkeleton(references, root, jointsData)
2:     Map each element in references to a respective initialized bodySegment
3:     BuildRecursively(root, bodySegments[root])
4:     return
5: end function

6: function BuildRecursively(current, currentSegment, jointsData)
7:     if current has no child objects then return
8:     end if
9:     Create new bodyPart from current
10:    Add Rigidbody component to bodyPart object
11:    Add ConfigurableJoint component to bodyPart object
12:    if current is a newSegment then
13:        Add bodyPart to newSegment
14:        if newSegment is not skeleton root then
15:            Connect currentSegment to newSegment with a joint
16:            Set joint rotation axes orientation
17:            Apply jointsData[newSegment] to joint
18:        end if
19:        currentSegment ← newSegment
20:    else
21:        Add bodyPart to currentSegment
22:    end if
23:    for each child in current do
24:        BuildRecursively(child, currentSegment, jointsData)
25:    end for
26: end function
```

As shown in the pseudo-code (Algorithm 1) the procedure consists in a recursive descent that traverses the hierarchy of GameObjects composing the rig. Each call attempts to identify the current GameObject: it creates the corresponding `BodyPart` instance and determines to which BodySegment it belongs.

The `BodyPart` class constructor attaches a `Rigidbody` and a `ConfigurableJoint` component to its respective GameObject. After that, the newly created `BodyPart` is appropriately connected to the existing structure. If it belongs to the same BodySegment as the parent, it is assigned to that BodySegment and connected without degrees of freedom. Instead, if the `BodyPart` marks the beginning of a new BodySegment, it is assigned to this new one and connected to the previous BodySegment following the rotation limits specified in the corresponding `JointLimitsData` for that particular joint.

### 4.3.2 Mass distribution policy

After all the Ragdoll components have been instantiated and properly stored, the `HumanoidRagdollMaker` component performs a distribution of body mass across all the `Rigidbody` components that compose it.

The option to distribute mass uniformly has been made available: in this scenario, each `Rigidbody` composing the Ragdoll will have a mass `m = bodyMass / AllRigidbodies.Length`. However, providing a reference to an object of type `BodyMassDistribution` with appropriately tuned values (our solution is described in 4.2.3) results in a more realistic physical behavior. In this case, a method will assign the appropriate percentage of the whole body mass to the target `BodySegment`.

If a `BodySegment` is composed by more than one `BodyPart`, the computed mass for that segment is evenly distributed among the parts. However, there is an exception for the trunk segment, which implements a specific policy empirically tested to provide a convincing result in the simulation:

1. The trunk body mass percentage is shared between hips, trunk, shoulders and neck `BodySegment`s;
2. 30% of that percentage is assigned to the hips segment;
3. The remaining 70% is evenly distributed among the `BodyPart`s that make up the spine if the neck and shoulders are not present as dedicated bones in the rig;
4. If the rig includes the above-mentioned bones:
    - 8% is assigned to each shoulder;

- 4% is assigned to the neck;
- The remaining 50% is evenly distributed across the remaining bones of the spine.

## 4.4 ActiveRagdollController component

Once the Ragdoll structure has been created, starting the simulation will simply result in leaving all the rigid bodies created in the hands of the physics engine. For instance, if gravity is enabled for all Ragdoll's `Rigidbody`s, at the start of the simulation the entire articulated structure will fall down, colliding with any encountered colliders during the fall, such as a ground plane.

The next step of our proposed solution consists in implementing a controller that acts like a "puppeteer" (as described in Section 3.2). To achieve this purpose, we implemented the `ActiveRagdollController` component. It is a Unity `MonoBehaviour` designed to work together with the `HumanoidRagdollMaker` component.

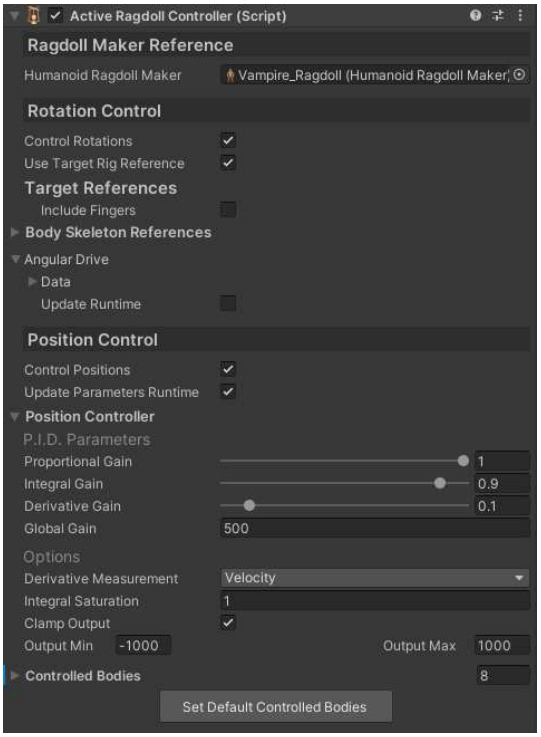



**Figure 4.5:** `ActiveRagdollController` custom Inspector.

Unlike the previous component, the `ActiveRagdollController` acts mainly at

runtime. Its custom Inspector (Figure 4.5) shows two separate sections of parameters that affects the two main tasks it performs: rotation control, and position control.

### 4.4.1 Joint rotation control

To mimic the movement of a twin animated target rig, the Ragdoll needs to be physically driven so that its bones relative rotation matches the target bones orientation. To reproduce this behaviour, we act on each joint that connects two `BodySegment`s, taking advantage of the Unity `ConfigurableJoint` component class which already exposes some useful fields [24] for the purpose:

- `targetRotation` represents the orientation that the joint's rotational drive rotates towards, specified as a quaternion. It is relative to the body that the Joint component is attached to.
- `slerpDrive` describes the parameters of the drive torque that rotates the joint around all local axes. This field is utilized instead of `angularXDrive` and `angularYZDrive` fields since all the Ragdoll's `ConfigurableJoint`s have the `rotationDriveMode` set to `Slerp`.

The `ActiveRagdollController` component performs rotation control by manipulating those fields. Initially (and also at runtime if the corresponding option is enabled), it applies the parameters specified in the `angularDrive` field to all `ConfigurableJoints` connecting two `BodySegment`s. Then, during each step of the physics simulation (i.e., at every `FixedUpdate` method call), the component updates the `targetRotation` fields with the rotations of the respective bones in the target rig, thanks to the mapping achieved with `targetReferences`.

If the option `useTargetRigReferences` is disabled, the default pose of the Ragdoll's avatar will be taken as reference for rotation control.

### 4.4.2 Bones position control

Given that a Ragdoll is an articulated mechanical system, it might seem logical that adjusting the local rotation of each `BodySegment` would be sufficient to mimic the target rig pose.

However, relying only on rotation control fails to address the need for the Ragdoll to follow the target rig's position in world coordinates, essential, for example, when it moves from point A to point B. Moreover, the torques applied by `ConfigurableJoint` components are often not enough fine-tunable, leading to a pose match result that lacks accuracy and is easily disrupted. Lastly, the system

lacks a solution to maintain the Ragdoll upright and balanced in response to the action of external forces, such as gravity.

The `ActiveRagdollController` performs position control precisely to address those criticalities. In line with the solution design (3.2.1), the component exposes an array of `Rigidbody-Transform` pairs. Here, the `Rigidbody` represents the physical object to control, while the `Transform` indicates the target position and rotation for the `Rigidbody`. This array can be populated with `Rigidbody`s of specific Ragdoll's `BodyPart`s designated to be position-controlled, paired with the respective reference bones belonging to the target rig. The custom Inspector (Figure 4.5) provides a button that automatically fills the array with the default `BodyPart`s (hips, upper arms, hands, feet and head), starting from the target `SkeletonReferences` and the `Skeleton` instance of the Ragdoll stored in the `HumanoidRagdollMaker` component.

Unlike rotation control, the `ActiveRagdollController` does not manage the position control execution directly. At the start of the simulation, it instantiates and attaches a new component of type `PositionController` for each controlled body, and sets its parameters according to the ones specified in the Inspector. The rest of the runtime control procedure is performed separately for each `BodyPart` by its respective `PositionController`. The `ActiveRagdollController` limits its operation to either enabling or disabling the overall control when requested, or updating at runtime the parameters of the `PositionController`s if the respective option is enabled.

The `PositionController` is a simple component that uses a PID controller to make a `Rigidbody` follow a target position. The control policy has been implemented in a dedicated class, the `Vector3PIDController`, which inherits from a generic abstract class `PIDController`. The class exposes a simple `Update` method, which returns the controller output result. An example of our PID control policy pseudo-code is shown in Algorithm 2.

The tuning of the PID parameters has been performed using a simple empirical approach, focusing solely on the qualitative outcome of the animation. The parameters proportion is showed in Table 4.3, with output values clamped between $-1000$ and 1000. The coefficient $K_{global}$ can be adjusted to fine-tune the responsiveness of the pose matching. We experimented with values ranging from 500 to 1000, resulting in movements that exhibit varying degrees of inertia. Lower values tend to demonstrate more pronounced inertia effects, while higher values yield stiffer movements that are less susceptible to natural disturbances, maintaining closer alignment with the target pose.

**Algorithm 2** PID controller pseudo-code.

**Require:**
- $\Delta t$ time interval;
- *current* value that has to be updated;
- *target* value as reference;
- $outputMin$, $outputMax$ as clamping range for the output;
- $I_{saturation}$ as a clamping range for the error integral.

$last \leftarrow 0$
$integration \leftarrow 0$
**function** UPDATE($dt, current, target$)
    $e \leftarrow target - current$
    $P \leftarrow K_p \times e$ ▷ Proportional term
    **if** derivative based on value velocity **then**
        $\Delta e \leftarrow -(current - last)$
        $last \leftarrow current$
    **else**
        $\Delta e \leftarrow e - last$
        $last \leftarrow e$
    **end if**
    $D \leftarrow \Delta e / \Delta t$ ▷ Derivative term
    $integration \leftarrow integration + e \times \Delta t$
    CLAMP($integration, -I_{saturation}, +I_{saturation}$)
    $I \leftarrow K_i \times integration$ ▷ Integral term
    $output \leftarrow K_{global} \times (P + I + D)$
    **if** clamp output **then**
        CLAMP($output, outputMax, outputMax$)
    **end if**
    **return** output
**end function**

| $K_p$ | $K_i$ | $K_d$ |
|---|---|---|
| 1 | 0.9 | 0.1 |

**Table 4.3:** PID parameters proportion setup for `PositionController`.

# Chapter 5

# Combining Active Ragdoll with VR Motion Matching

In the previous chapters, we extensively discussed the proposed solution for achieving physics-based animation. This chapter delves deeper into the application of this approach to the VR Motion Matching technique developed by Ponton [2]. It is important to note that VR development was conducted using a Meta Quest 2, with the assistance of tools provided by the XR Interaction Toolkit SDK 2.5.2 [25], ensuring compatibility across multiple platforms.

In the scene, three main groups of objects are needed:

- A rig that manages the VR control systems. In this case, the template is provided by the XR Interaction Toolkit SDK.
- A copy of the avatar set up with the components developed by Ponton [2] that implements VR Motion Matching animation.
- A second copy of the avatar configured with the components of our Active Ragdoll Module.

The bones of the avatar implementing Motion Matching are used as target references for the `ActiveRagdollController` attached to the created Ragdoll.

## 5.1 Coherence between Ragdoll and target rig

When it comes to using full-body models as avatars in VR, it is important to consider the differences in heights and body dimensions of the various users. Mapping these proportions onto the avatar helps to achieve a more precise match of movements.

For the purpose of this goal, the Ponton's implementation of VR Motion Matching already included an avatar calibration module, that adjusts the scale of the avatar 3D model in order to match its height with the one computed by considering the distance of the HMD from the ground.

Since the upper limbs are the primary means through which users interact with the virtual environment, we have integrated this calibration module with an additional control. This ensures that the length of the 3D model's arms aligns with the user's actual measurements, calculated by determining the ratio between the distance from the shoulder bones to the hand bones and the distance from the shoulder bones to the controller trackers. The actual application of arms calibration to the 3D model is managed by a public field `armLengthMlp` exposed by the `ArmIK` component of the RootMotion's Final IK [26] package used to compute inverse kinematic.

This calibration is performed on the target rig. To mirror these adjustments onto the final Ragdoll, we've added a `MatchScale` component at the base of the Ragdoll object. This component ensures that the Ragdoll maintains the same scale as the target rig, aligning their heights. Regarding the arms, since it's not feasible to apply the same mesh deformation from the target rig to the Ragdoll at runtime, we opted to introduce a minimal degree of linear flexibility. We achieved this by slightly loosening the movement linear constraints along the twist axis of the elbow and hand joints (Figure 5.1). Specifically, for these two joints, the `linearLimit` fields of their `ConfigurableJoint`s were set to 0.1 and 0.08, respectively.

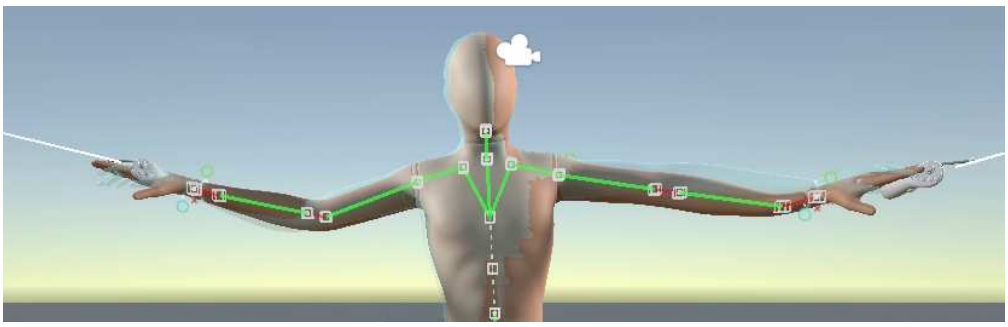


**Figure 5.1:** Ragdoll arms extension to compensate arm stretching. Red lines represents linear distance between loosened joints.

## 5.2 XR Hand Tracking

The emerging technologies in VR headsets are increasingly incorporating augmented reality features, leading to more XR-oriented experiences. One of the most relevant

aspect in this regard is the ability to replace controllers with direct hand control, thanks to the hand tracking features provided by the latest generation headsets. To leverage the versatility inherent to the approach pursued in this project, we have considered testing the results of this method when applied to the implementation of a physics-based hand tracking system.

The `RagdollMaker` component has been expanded to include the ability to extend the construction of the physical Ragdoll even to the fingertips. An inherited `HandSkeletonReferences` class has been created, sharing the same functionalities of `SkeletonReferences`. On the other hand, the `ActiveRagdollController` component undergoes no substantial changes. To provide sufficient precision and responsiveness of movements, 10 more PID controllers are added to the simulation, as all rigid bodies corresponding to the fingertip bones are added as controller bodies to the position control. Those PID controllers share the same parameters with the ones acting on the other controlled bones.

Regarding the actual hand tracking animation, the XR Interaction Toolkit SDK already comes with samples for hand tracking controls, including hand visualization. To reproduce the movement on the target rig hands, we implemented a custom component called `XRHandAnimatorTranslator`. By providing a source and a target `HandSkeletonReferences` instances, this component applies the same geomatrical transforms acting on the animated XR hand sample to the mesh of the target rig's hand (Figure 5.2).

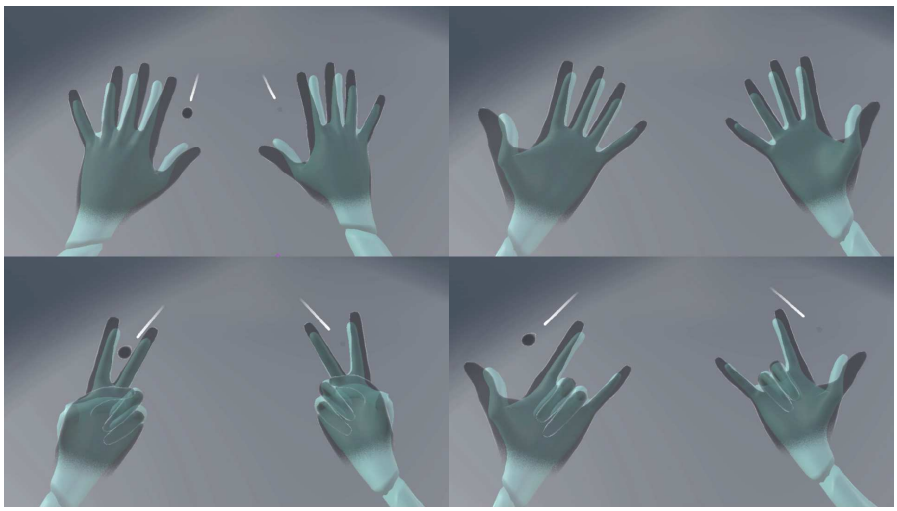


**Figure 5.2:** Avatar hands animation with `XRHandAnimatorTranslator`. The dark grey meshes are the reference hands provided by XR Interaction Toolkit, the transparent cyan mesh is the target rig.

# Chapter 6

# Discussion and Results

The following considerations are based on tests conducted within the Unity development environment. The workstation used during the tests has the following hardware specifications:

- CPU: Ryzen 7 3700x
- RAM: 32GB DDR4
- GPU: Nvidia RTX 3070ti
- Storage: 500GB SSD NVMe

## 6.1 Active Ragdoll Module

Considering the `HumanoidRagdollMaker` component, the description of its implementation (4.3) has already highlighted a series of precautions and automations in the editor aimed at simplifying the Ragdoll setup as much as possible. Having a rigged 3D model equipped with colliders already available, it is possible to build the physics Ragdoll in just a few clicks by attaching the `HumanoidRagdollMaker` component to the respective GameObject, providing an Animator reference to automatically populate `SkeletonReferences` fields (or alternatively assigning them manually), providing the two references to the `BodyMassDistribution` and `RagdollJointsData` assets, and finally clicking the "Ragdollify!" button.

The presence of the `bonesToIgnore` array enhances the tool's compatibility with a variety of rigged 3D models, allowing it to disregard any additional bones in the rig that do not represent part of the humanoid skeleton (e.g., bones related to the

model's clothing, extra auxiliary bones that do not need to be represented by a standalone BodyPart).

As shown in Table 6.1, the execution time of the `Ragdollify` method is entirely negligible and imposes no burden on the application development process.

| **Execution time** | No console log prints | With console log prints |
|---|---|---|
| `Ragdollify` | 5ms | 17ms |
| `Ragdollify` (including fingers) | 8ms | 36ms |

**Table 6.1:** Average execution time measured for the Ragdoll building procedure performed by `HumanoidRagdollMaker` component.

Observing the generated Ragdoll responding passively to external forces, we can conclude that the final parameters chosen to configure the joint limits (4.2.2) allow for flexibility in line with the natural movements of the human body, while the body mass distribution (4.2.3) based on de Leva's research [22], results in a natural reaction to forces as well.

Moving on to the performance analysis, the `ActiveRagdollController` and the `PositionController` components are the ones designated to run operations at runtime. Using the Unity Profiler, we pinpointed specific calls and code sections and observed their impact on the simulation. We focused on the performance of three particular code sections:

- The `FixedUpdate.PhysicsFixedUpdate`, which is a Unity native engine system call visible in the Profiler. It performs most of the runtime physics engine tasks (e.g., compute collisions, update rigid bodies).
- The rotation control loop, which updates all the `targetRotation` fields, executed in the `FixedUpdate` method of the `ActiveRagdollController` component.
- The evaluation of the PID controller output performed by the `FixedUpdate` method of the `PositionController` component

The average execution times are presented in Table 6.2. The time values shown represent the total time spent executing each specific section. This means that if there are 18 Ragdolls in the scene, those values are respectively the sum of 18 rotation control calls and $18 \times N_{ControlledBodies}$ PID update calls executed at each simulation step. In this test scene, Ragdolls do not include fingers in the physical skeleton structure, and major debug features were turned off.

| **Runtime performance** | 1 Ragdoll | 9 Ragdolls | 18 Ragdolls |
|---|---|---|---|
| `FixedUpdate.PhysicsFixedUpdate` | 0.5ms | ∼1.8ms | ∼3.1ms |
| Rotation control | 0.06ms | ∼0.4ms | ∼0.8ms |
| PID controller update | 0.01ms | ∼0.11ms | ∼0.22ms |

**Table 6.2:** Average runtime performances of different code sections.

From an overall perspective, it can be asserted that the overhead introduced by our Ragdoll control implementation in the simulation is non-intrusive. Comparing execution times, the workload that the physics engine must handle to simulate the considerable number of rigid bodies that compose each Ragdoll is more demanding. However, considering that in this project we aim to use the tools developed to obtain a single Ragdoll corresponding to the user's VR avatar, we can assert that our implementation is usable without significantly impacting performance.

It is worth mentioning that enabling the debug features of `HumanoidRagdollMaker` component, useful to visualize the Ragdoll structure in the Scene View, introduces a non-negligible overhead in performance.

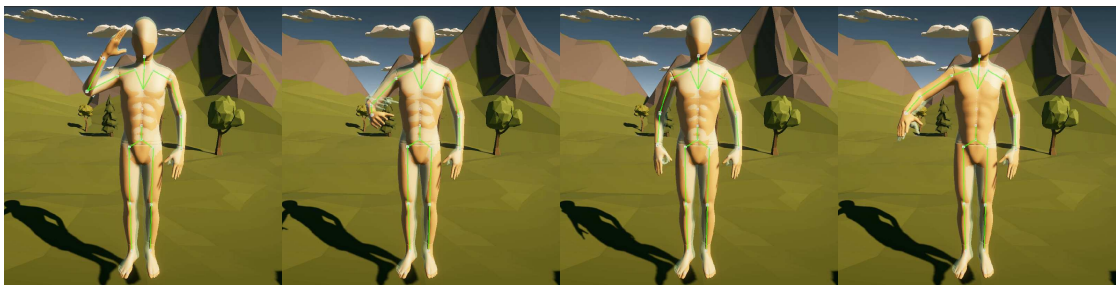


**Figure 6.1:** Active Ragdoll Module: salute animation. Skeleton structure gizmos are visible.

To assess the qualitative outcome of the final Ragdoll animation, we initially tested the developed tools on avatars animated with simple animation clips before evaluating their performance on a VR avatar. The frames proposed for demonstration (Figure 6.1, 6.2, 6.3) were captured in a test scene, leaving the target rig visible, rendered with a transparent material and overlayed to the Ragdoll.

Overall, the Ragdoll accurately replicates the movements of the target rig. With the selected parameters, pose matching remains stable, and the `PositionControllers` are not subjected to irreversible divergences. While there is generally minimal

**Figure 6.2:** Active Ragdoll Module: drunk walk animation.

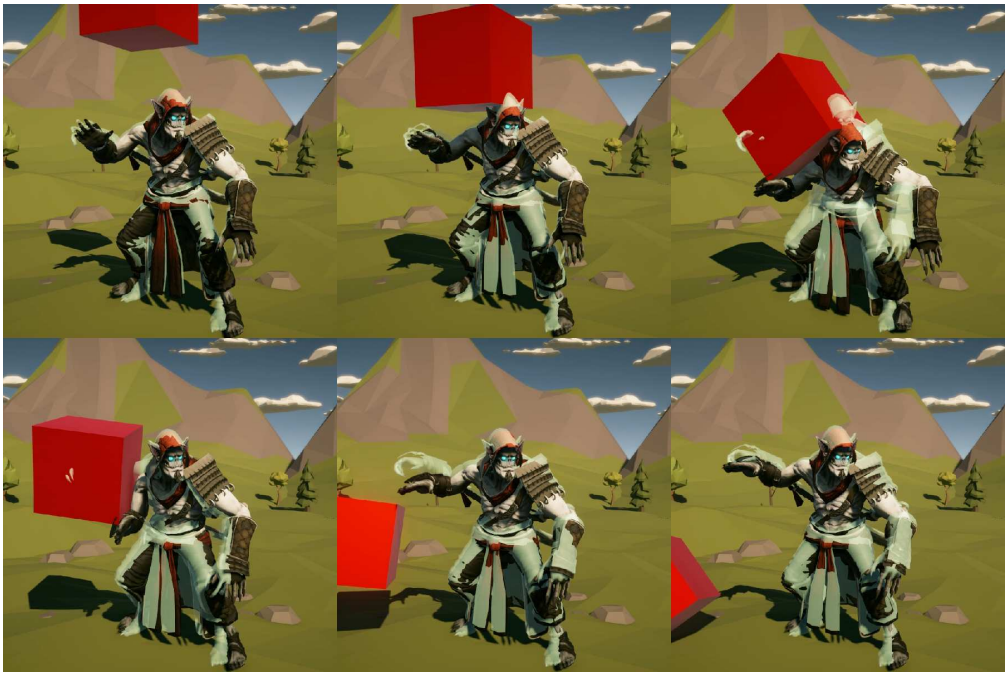


**Figure 6.3:** Active Ragdoll Module: reaction of the Ragdoll to a falling cube. The visual result is really natural, and the Ragdoll returns to its target pose without unwanted jittering or artifacts in the movements.

discrepancy between the target rig and the Ragdoll pose, the extent of this difference varies depending on the speed of the movements performed by the target rig.

Because of the inertia of the Ragdoll's masses, its parts may exhibit some deviation from the target pose, especially during rapid animations. This relative responsiveness may introduce a potential delay in reaching the target pose, but on the other hand it imparts a more physical and natural character to the animation.

## 6.2 Application to VR Motion Matching

Testing the use of our Active Ragdoll Module with a VR avatar does not require additional steps beyond what has been mentioned so far.

Two main scenes have been set up. The first consists of a simple empty testing environment with three mirrors positioned in front of and on both sides of the avatar, allowing users to observe their movements while wearing the headset. The second scene has been arranged as if it was a VR app demo environment, featuring a more complex geometry. It also includes some physics-based props near the avatar that users can interact with, in order to assess both the environment's and the avatar's response to collisions and external forces.

The `ActiveRagdollController` faithfully maps the movements of the avatar animated with Motion Matching. As mentioned earlier, sudden movements introduce a slight delay in reaching the target. For this type of application, disabling gravity on the Ragdoll's rigid bodies has shown an improvement in movement mapping.

**Figure 6.4:** VR Ragdoll: arms movement mapping.

Regarding the arms, the results are quite satisfying: the articulation of movements

calculated by the inverse kinematic solver is well mapped, allowing the attainment of most desired poses (Figure 6.4). In some situations, it may happen that parts of the Ragdoll get stuck due to collision detections, or that the rotation control causes more delicate joints (such as those of the shoulder) to reach illegal conditions. However, the system does not completely diverge, returning to normal after performing adjustment movements.

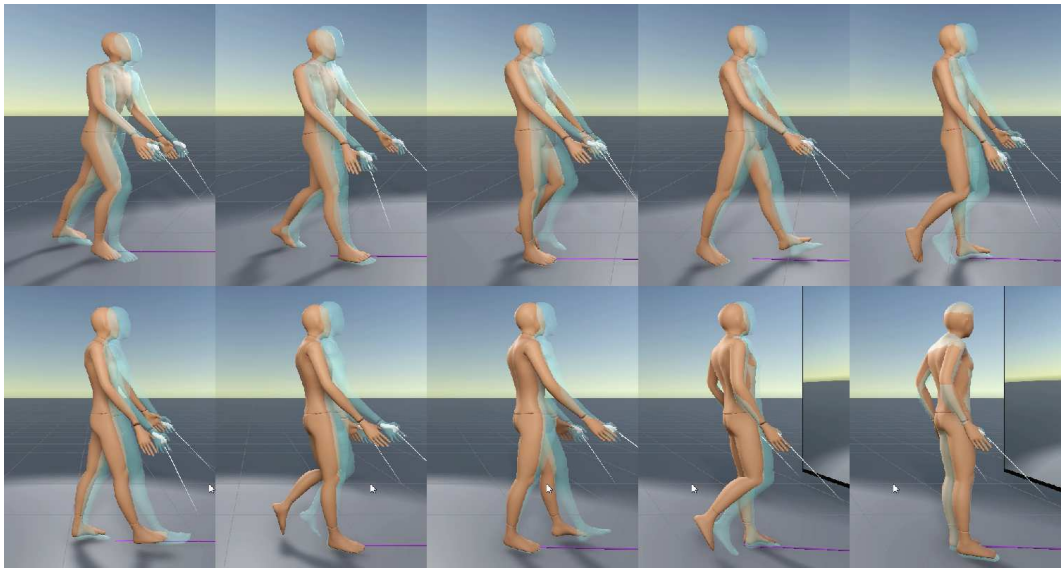


**Figure 6.5:** VR Ragdoll: walking movement mapping.

The locomotion provided by the Motion Matching technique is also properly mapped to the Ragdoll (Figure 6.5). Occasionally, there are some artifacts in the animation result due to collisions and dragging of the feet on the ground.

The primary advantage of implementing physics-based animation is the ability to interact with other elements in the scene governed by the physics engine. In this regard, the controlled Ragdoll performs its job effectively: there are no penetrations between the avatar and other collider-equipped objects. For instance, attempting to pass through a static wall with the hand will result in the hand being blocked by the wall until the user's hand returns to a position it can reach again. This occurs without any manual control over parameters or other aspects of the `ActiveRagdollController` or `PositionControllers`.

Interacting with other rigid bodies in the scene provides a sense of realism, as the scene reacts realistically to the Ragdoll's movements. In the second test scene (Figures 6.6, 6.7, 6.8), the user can push cubes of various sizes, rotate large discs suspended in mid-air, swing an always-standing puppet, or walk among cones kicking them away. It is important to note that this research does not aim to

implement a system for grabbing, handling, or manipulating objects.

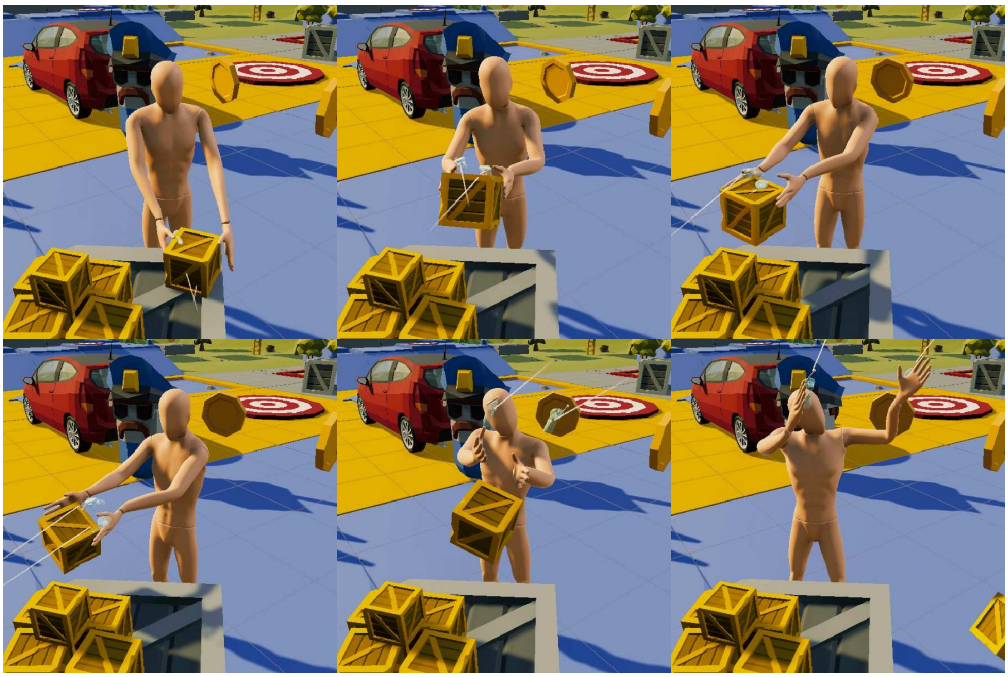


**Figure 6.6:** VR Ragdoll: throwing a cube.

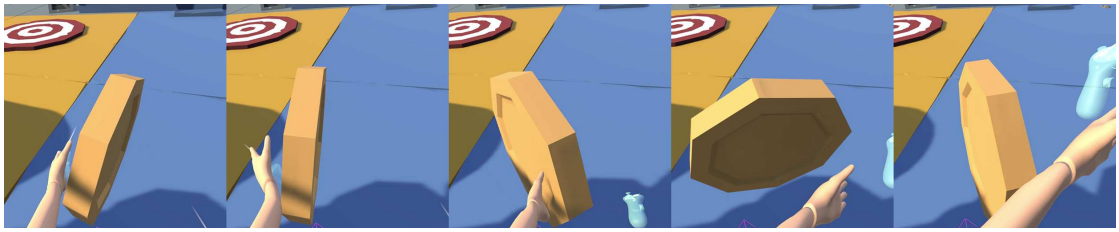


**Figure 6.7:** VR Ragdoll: spinning a coin. Subjective view.

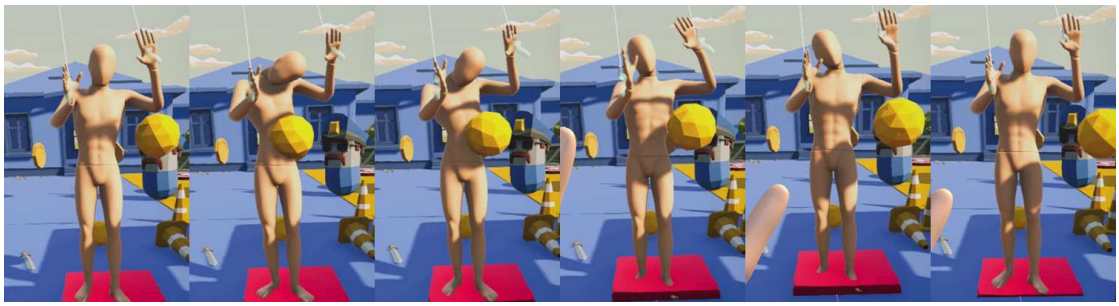


**Figure 6.8:** VR Ragdoll: reacting to a hitting body.

This aspect represents the primary benefit of choosing to implement a physics-based

VR avatar.

### 6.2.1 Fingers Ragdoll for XR Hand Tracking

In Section 5.2, we briefly mentioned the intention to achieve a complete Ragdoll down to the fingertips to test the feasibility of mapping physics-based hand tracking movements. Although the `XRHandAnimatorTranslator` component is implemented with a simple and not particularly refined approach to matching the 3D model's hand mesh with that of the hand tracking, it still allows for a plausible animation of the target rig's hands.

**Figure 6.9:** VR Ragdoll: physics hands.

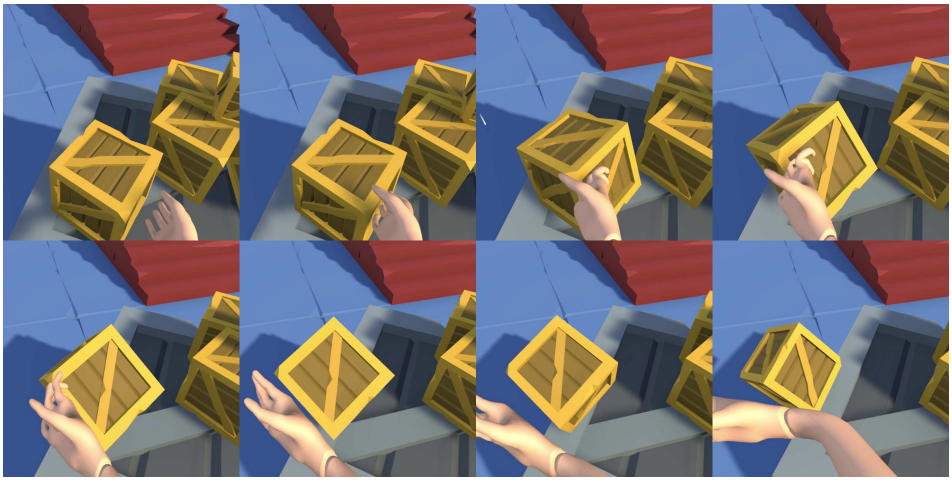


**Figure 6.10:** VR Ragdoll: physics hand interacting with another rigid body. Fingers deformation on contact is clearly visible.

However, mapping movements onto the Ragdoll's hands is more fragile and less stable than the rest of the body. While the fingertips correctly follow their targets (Figure 6.9), the overall movement result is often wobbly, and touching objects often deforms the hand structure (Figure 6.10).

Overall, we can assert that the physics-based hand tracking experiment, as a side aspect of the main research, does not yield equally satisfying results when simply approached by extending the Ragdoll to the fingertips and using the same control methodology.

# Chapter 7

# Conclusions

In this thesis project, we focused our attention on simulating the movements of a virtual character in a way that is compliant with the physics laws. It has been highlighted how this can provide a positive contribution during the development of an interactive experience, whether to meet specific design needs, develop certain interaction mechanics, or enhance immersion in a VR experience.

The objective was therefore to develop a prototype tool in the Unity environment that would allow translating a generic animation technique to the domain of the physics engine. The basic idea behind the design of this prototype was to ensure a versatile, modular, and ready-to-use approach during development process. The developed tools were then tested with different avatars, with greater attention directed towards the application on full-body VR avatars animated with the VR Motion Matching technique [2].

Overall, we can affirm that the Active Ragdoll Module, with attention paid to its usability, met expectations of versatility and ease of use, presenting acceptable performance that does not significantly burden real-time simulation. The fidelity of movement mapping can be considered effective. The combination of the Active Ragdoll Module and VR Motion Matching yields an appreciable result overall, with the desired greatest advantage of eliminating interpenetrations and allowing interaction with other objects subjected to the action of the physics engine.

Our work still offers room for improvement, considering the presence of defects and artifacts in the Ragdoll movements. Despite these challenges, we hope that our work has highlighted the potential of physics-based animation techniques and underscored the importance of designing development tools that ensure versatility and ease of use. We aim to inspire research that can have a tangible and pragmatic impact in a demanding market such as the gaming and VR industry.

## 7.1 Future work

In conclusion, we would like to list below some possible ideas for further developing the Active Ragdoll Module and evaluating potential improvements:

- **Automatic collider generation**: Our current solution prioritizes the development of core features and does not include automatic collider generation for the Ragdoll. Exploring methods to automatically generate precise, well-suited, and optimized colliders based on the avatar's geometry could further enhance the usability and versatility of the Active Ragdoll Module.
- **Rotation control rework**: considering our implementation, the rotation control was managed by taking advantage of the existing properties available in the `ConfigurableJoint` component provided by Unity. Similar to the approach used for position control, it could be interesting to explore the possibility of implementing a custom rotation controller, such as a PID controller handling quaternion values. This would provide greater control over the mathematics governing the rotations of the Ragdoll's joints, allowing for more customization in tuning its parameters.
- **Refinement of avatar calibration**: the method employed in our implementation relies on the simple scale management of certain parts of the avatar. Enhancing the final outcome could involve applying more sophisticated custom deformation techniques, as well as refining the mapping of these deformations onto the physics Ragdoll.
- **Adjustment of feet positioning**: the introduction of collision detection during locomotion has underscored the need for closer attention to the Ragdoll's feet positioning. Therefore, it would be beneficial to add an additional layer of procedural management to adjust the position of the feet targets to prevent penetrations and dragging on the ground.
- **Exploration of hand tracking application**: in this study, the potential extension of the developed tools for the implementation of a physics-based hand tracking system was proposed as a simple side experiment. Delving deeper into this specific aspect could be intriguing, evaluating a more tailored implementation of the Active Ragdoll Module to achieve a more functional animated hand Ragdoll.

# Bibliography


[1] Simon Clavet. «Motion Matching and The Road to Next-Gen Animation». In: *Game Developer Conference*. GDC 2016. Mar. 2016. URL: https://www.gdcvault.com/play/1023280/Motion-Matching-and-The-Road (visited on 03/02/2024) (cit. on p. 6).

[2] Jose Luis Ponton, Haoran Yun, Carlos Andujar, and Nuria Pelechano. «Combining Motion Matching and Orientation Prediction to Animate Avatars for Consumer-Grade VR Devices». In: *Computer Graphics Forum* 41.8 (Dec. 2022), pp. 107–118. ISSN: 0167-7055, 1467-8659. DOI: 10.1111/cgf.14628. arXiv: 2209.11478[cs]. URL: http://arxiv.org/abs/2209.11478 (visited on 10/27/2023) (cit. on pp. 6, 32, 44).

[3] Loïc Caroux, Katherine Isbister, Ludovic Le Bigot, and Nicolas Vibert. «Player–video game interaction: A systematic review of current concepts». In: *Computers in Human Behavior* 48 (July 1, 2015), pp. 366–381. ISSN: 0747-5632. DOI: 10.1016/j.chb.2015.01.066. URL: https://www.sciencedirect.com/science/article/pii/S0747563215000941 (visited on 02/17/2024) (cit. on p. 6).

[4] Kay Stanney and Gavriel Salvendy. «Aftereffects and Sense of Presence in Virtual Environments: Formulation of a Research and Development Agenda». In: *International Journal of Human–Computer Interaction* 10.2 (June 1, 1998). Publisher: Taylor & Francis _eprint: https://doi.org/10.1207/s15327590ijhc1002_3, pp. 135–187. ISSN: 1044-7318. DOI: 10.1207/s15327590ijhc1002_3. URL: https://doi.org/10.1207/s15327590ijhc1002_3 (visited on 02/17/2024) (cit. on p. 6).

[5] Kwan Min Lee. «Presence, explicated». In: *Communication Theory* 14.1 (2004). Place: United Kingdom Publisher: Blackwell Publishing, pp. 27–50. ISSN: 1468-2885. DOI: 10.1111/j.1468-2885.2004.tb00302.x (cit. on p. 6).

[6] Yujie Tao, Cheng Yao Wang, Andrew D Wilson, Eyal Ofek, and Mar Gonzalez-Franco. «Embodying Physics-Aware Avatars in Virtual Reality». In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. New York, NY, USA: Association for Computing Machinery, Apr. 19, 2023, pp. 1–15. ISBN: 978-1-4503-9421-5. DOI: 10.1145/3544548.3580979. URL: https://doi.org/10.1145/3544548.3580979 (visited on 10/27/2023) (cit. on p. 7).

[7] Mar Gonzalez-Franco, Brian Cohn, Eyal Ofek, Dalila Burin, and Antonella Maselli. «The Self-Avatar Follower Effect in Virtual Reality». In: *2020 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. 2020 IEEE Conference on Virtual Reality and 3D User Interfaces (VR). ISSN: 2642-5254. Mar. 2020, pp. 18–25. DOI: 10.1109/VR46266.2020.00019. URL: https://ieeexplore.ieee.org/document/9089510 (visited on 12/01/2023) (cit. on p. 7).

[8] Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. «DeepMimic: Example-Guided Deep Reinforcement Learning of Physics-Based Character Skills». In: *ACM Transactions on Graphics* 37.4 (Aug. 31, 2018), pp. 1–14. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/3197517.3201311. arXiv: 1804.02717[cs]. URL: http://arxiv.org/abs/1804.02717 (visited on 03/01/2024) (cit. on p. 8).

[9] Yongjing Ye, Libin Liu, Lei Hu, and Shihong Xia. *Neural3Points: Learning to Generate Physically Realistic Full-body Motion for Virtual Reality Users*. Sept. 13, 2022. DOI: 10.48550/arXiv.2209.05753. arXiv: 2209.05753[cs]. URL: http://arxiv.org/abs/2209.05753 (visited on 03/01/2024) (cit. on p. 8).

[10] Alexander Winkler, Jungdam Won, and Yuting Ye. «QuestSim: Human Motion Tracking from Sparse Sensors with Simulated Avatars». In: *SIGGRAPH Asia 2022 Conference Papers*. Nov. 29, 2022, pp. 1–8. DOI: 10.1145/3550469.3555411. arXiv: 2209.09391[cs]. URL: http://arxiv.org/abs/2209.09391 (visited on 03/01/2024) (cit. on p. 8).

[11] Sunmin Lee, Sebastian Starke, Yuting Ye, Jungdam Won, and Alexander Winkler. «QuestEnvSim: Environment-Aware Simulated Motion Tracking from Sparse Sensors». In: *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Proceedings*. July 23, 2023, pp. 1–9. DOI: 10.1145/3588432.3591504. arXiv: 2306.05666[cs]. URL: http://arxiv.org/abs/2306.05666 (visited on 10/27/2023) (cit. on p. 8).

[12] Joan Llobera and Caecilia Charbonnier. «Physics-based character animation for Virtual Reality». In: Mar. 1, 2022, pp. 56–57. DOI: 10.1109/VRW55335.2022.00021 (cit. on p. 8).

[13] Johan Gästrin. *Physically Based Character Simulation: Rag Doll Behaviour in Computer Games*. Google-Books-ID: CZjyjwEACAAJ. 2004. 44 pp. (cit. on p. 10).

[14] Gabe Mulley and Matt Bittarelli. «Ragdoll Physics». In: 2007. URL: https://www.semanticscholar.org/paper/Ragdoll-Physics-Mulley-Bittarelli/d80c8dbe9c5864c46e573f92c66df52190be2155 (visited on 02/17/2024) (cit. on p. 10).

[15] Boneloaf. *Gang Beasts*. 2014. URL: https://gangbeasts.game (visited on 02/17/2024) (cit. on p. 10).

[16] No Broken Games. *Human Fall Flat*. 2016. URL: https://nobrakesgames.com/games/human-fall-flat/ (visited on 02/17/2024) (cit. on p. 10).

[17] Recreate Games. *Party Animals*. 2023. URL: https://partyanimals.com/ (visited on 02/17/2024) (cit. on p. 10).

[18] Karl Johan Åström and Tore Hägglund. *PID Controllers: Theory, Design, and Tuning*. Research Triangle Park, North Carolina: ISA - The Instrumentation, Systems and Automation Society, 1995. ISBN: 978-1-55617-516-9 (cit. on p. 10).

[19] Heinz Unbehauen. *CONTROL SYSTEMS, ROBOTICS AND AUTOMATION - Volume II: System Analysis and Control: Classical Approaches-II*. Google-Books-ID: RF1xDAAAQBAJ. EOLSS Publications, Oct. 11, 2009. 416 pp. ISBN: 978-1-84826-141-9 (cit. on p. 10).

[20] Karl Johan Åström and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers, Second Edition*. Google-Books-ID: l50DEAAAQBAJ. Princeton University Press, Feb. 2, 2021. 522 pp. ISBN: 978-0-691-19398-4 (cit. on p. 10).

[21] Karl Grammer, Bernhard Fink, Elisabeth Oberzaucher, Michaela Atzmüller, Ines Blantar, and Philipp Mitteroecker. «The representation of self reported affect in body posture and body posture simulation». In: *Collegium antropologicum* 28 Suppl 2 (Feb. 1, 2004), pp. 159–73 (cit. on p. 21).

[22] P. de Leva. «Adjustments to Zatsiorsky-Seluyanov's segment inertia parameters». In: *Journal of Biomechanics* 29.9 (Sept. 1996), pp. 1223–1230. ISSN: 0021-9290. DOI: 10.1016/0021-9290(95)00178-6 (cit. on pp. 22, 23, 36).

[23] Triangular Pixels. *Technie Collider Creator 2 | Physics | Unity Asset Store*. URL: https://assetstore.unity.com/packages/tools/physics/technie-collider-creator-2-217070 (visited on 03/10/2024) (cit. on p. 24).

[24] Unity Technologies. *Unity - Scripting API: ConfigurableJoint*. URL: https://docs.unity3d.com/ScriptReference/ConfigurableJoint.html (visited on 03/03/2024) (cit. on p. 28).

[25] Unity. *XR Interaction Toolkit | Documentation*. URL: `https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.5/manual/index.html` (visited on 03/04/2024) (cit. on p. 32).

[26] RootMotion. *Final IK | Documentation*. URL: `http://www.root-motion.com/final-ik.html` (visited on 03/05/2024) (cit. on p. 33).