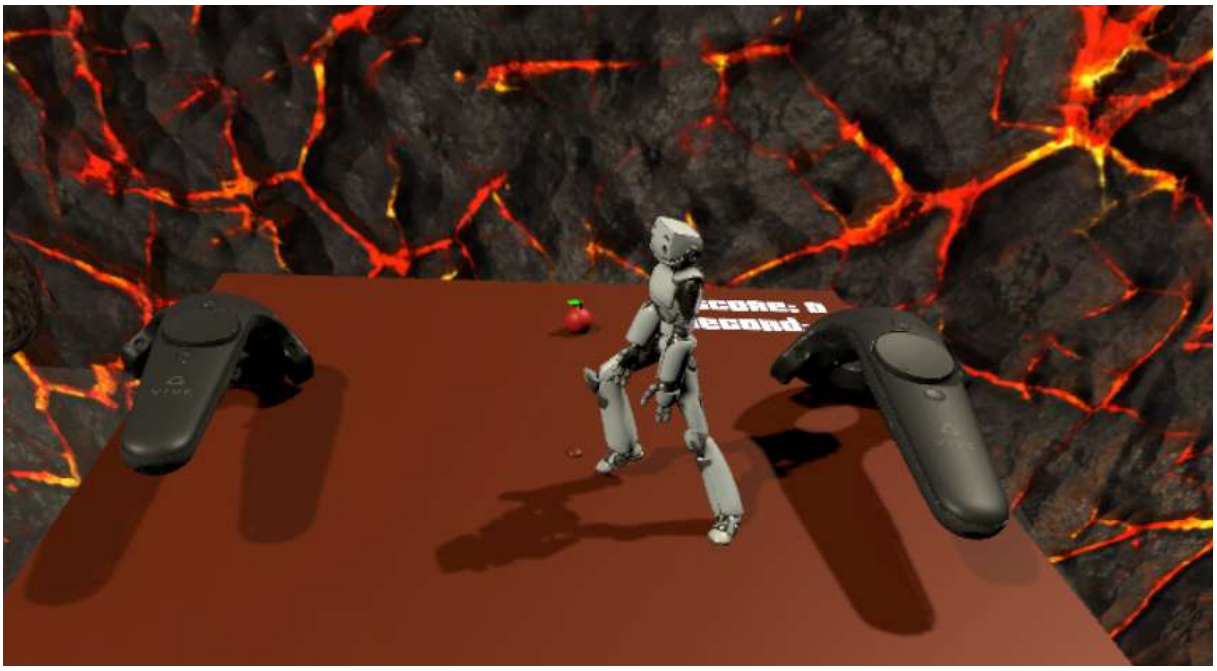
Making Simulated Characters Accessible in Unity



Simone Guggiari

Bachelor's Thesis Project
Spring Semester 2017

Prof. Dr. Robert W. Sumner
Supervised by Dr. Martin Guay

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Disney Research

cgl
computer graphics laboratory

Bachelor's Thesis Project

Making Simulated Characters Accessible in Unity

Interaction Demo: Pushing Characters Around with the Force (like a Jedi)

Simone Guggiari

Introduction

Game environments such as Unity and Unreal Engine offer casual users, as well professionals, the possibility to rapidly create compelling interactive games and experiences. However, creating character animation that response in realistic unique ways to unscripted perturbations is a time consuming task, the *de-facto* animation system is based on blending between predefined motion clips that quickly become repetitive.

One way to obtain novel and unique reactions to unscripted events---such as pushes---is to embed the character into a simulation. The challenge with this however, is just as hard as controlling a robot in the real world. Without joint level torque control, the character simply falls like puppet.

Over the past years, Disney Research and ETH have developed an expertise as well as a high performance code base (called Aikido) for controlling simulated characters. It is comprised of various types of controllers---based on high performance optimization algorithms---which are hard to implement in high level game engines such as Unity.

This project aims at opening the door to simulated characters in high level game engines such as Unity. To showcase the richness of the interactions brought by simulated characters, we will create a small interactive experience where the user learns to push objects around using hand gestures. We call this experience *Forceful: a game that helps you learn how to master the force (like a Jedi)*.

Our concept relies on a simple and robust controller for locomotion that includes a balancing strategy (SIMBICON [1] and [3]). The tasks will involve providing the character with a mesh, wrapping the library as to be loaded into unity, providing the character with simple AI and detecting when to turn it's control into passive mode, or break the connection between joints due too strong forces. Additional challenges for improvement of the controller are listed below.

Task Description
The first task will consist in parameterizing a mesh with the skeleton representation used for simulation in the Aikido library. This requires adapting our C++ FBX file importer and exporter. Then the new character mesh can be loaded into Unity, and our work on updating the joint angles with the Aikido library can begin.

Once the character skeleton is updated using Aikido, we may proceed to creating the scene and loading the app in VR HTC vive and/or AR HoloLens. Next, we provide the character with simple AI and detecting when to turn it's control into passive mode, or break the connection between joints due too strong forces

For additional challenges, we may investigate methods to avoid self-collisions, as well as smooth the distribution of joint torques generated from virtual forces (which are currently mapped with the Jacobian Transpose method).

Milestones

Task	Due date
Start	**Feb 20 2017**
1. Export from RBS to FBX and rig mesh then load in Unity.	**Feb 27**
2. Transfer FBX character mesh to simulated RBS mesh.	**Mar 6**
3. Update character skeleton (state) from Aikido lib loaded into Unity.	**Mar 13**
4. Load application into HTC Vive and HoloLens.	**Mar 20**
5. Scene and assets, together with force interaction.	**Apr 10**
6. Turn into passive ragdoll (set ODE joint limits) when force is too strong, or break into pieces.	**Apr 24**
7. Simple AI: control velocity and orientation	**Mai 1**
8. Investigate additional interactions: fix leg, make ground icy with friction coefficient, and change body masses.	**Mai 15**
9. Transition between different walking styles, and set arm swings to have two characters fight each other.	**Mai 22**
10. Distribute virtual forces to joint torques using quadratic program [2].	**Before July**
11. Activate self-collisions between bodies (in ODE) avoid by changing foot trajectory (maybe also virtual forces).	**Before July**
Thesis written	**August 20**

Remarks
A report and an oral presentation conclude the thesis. It will be overseen by Prof. Bob Sumner and supervised by Dr. Martin Guay, with support from other animation group members.

References
[1] Generalized Biped Walking Control, SIGGRAPH 2010.
[2] Hierarchical Planning and Control for Complex Motor Tasks, SCA 2015.
[3] Intuitive Design of Simulated Character Controllers, Semester Thesis, 2016.

Abstract

This thesis addresses the task of making high-end algorithms for simulating the animation of characters accessible to the general public via widely used platforms such as Unity, and crafting fun and engaging experiences for the audience to show the practical applications of said algorithms.

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Aufgabe, High-End-Algorithmen zur Simulation der Animierung von Figuren, die für das Publikum zugänglich sind, über weit verbreitete Plattformen wie Unity zu schaffen und lustig und spannende Erfahrungen für das Publikum zu vermitteln, um die praktischen Anwendungen dieser Algorithmen zu zeigen.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Topic

Character animation plays a very central role in many disciplines, from cartoons to 3D movies to videogames. It is used to inject life into otherwise static drawings or 3D meshes. Animation means exactly that: bringing images to life, from anima ("soul, giving life to").

animate

adjective | an·i·mate | \ˈa-nə-mət\

In particular, in *interactive* experiences such as videogames, animation is fundamental in giving credibility to the game, making it look alive and kicking. Since a videogame is interactive by nature, much more is required from an animation system. Up until recent years, the standard in the industry has been blending between pre-defined clips, previously recorded or manually animated sequences of motion, which are fixed and require work from the digital artists. This implies that what we experience are pre-recorded sets of motions. Since the number of said motions must be bounded, we only see a subset of all possible motions in our experience, and this limitation means that reactions to external events and unscripted perturbations are repetitive. To solve this problem, in recent years a new approach has been developed, namely that of simulated characters (e.g. SIMBICON, [Stelian Coros, 2010]). These approaches have been only used in research and not in commercial applications up until now. They allow to embed the control of a character in a (physical) simulation, which can then be made to respond to external *stimuli* accordingly. This results in characters inside videogames responding in much more realistic and believable ways to what the player might do. Example of this is when a player is hit by an object, like an arrow or a bullet. Usually a pre-recorded clip is played. Although sophisticated system choose the clip based on a few criteria like position of the hit and current playing animation, the animations quickly become repetitive. With this new approach, it is now possible to simply let the simulation run.

This project however doesn't concern itself with the design of such controllers, but rather focuses on making this technology available to the general public through accessibility in high-level programming engines such as Unity, as well as exploring interactions with such characters.

The goal of this thesis is therefore to make such high performing algorithms available in commercial and popular game engines such as Unity, and explore new ways of interaction and gameplay mechanics that can be achieved by using such controllers.

1.2 Inspiration for our Creation Concept

The inspiration for this project came from a Virtual Reality Demo named *Star Wars: Trials on Tatooine*, by ILMxLAB, a LucasFilm technology and entertainment studio that develops interactive experiences in the world of Star Wars. This demo, puts you in the shoes of an aspiring Jedi and has you fight off waves of enemy stormtroopers swinging your lightsabre to deviate the lasers shot at you. It uses a VR headset to put you inside the action, and the 3D position and orientation of the controllers to simulate the wielding of a lightsaber. The player can physically move in his room and swing the controller. He sees his actions on the in-game character, which adds a lot to the realism and immersion of the experience. A screenshot of the game can be seen in figure 1.1.

However, the problem with this is that the enemies react in always the same way when hit, because pre-recorded animation clips are played each time they are hit. This is a problem we wanted to solve in our own demo.

From this stemmed the idea for this thesis project: to create an experience with the simulation controller to allow the player to interact with accurate and realistic responses to external stimuli, like pushes, hits, or ground obstacles. Also, because the controller we had produced motion that looked that of a robot, we decided to theme the whole application using droids.

1.3 Motivation

To the average nontechnical user, any showcase of complex control systems doesn't arouse the enthusiasm of a more knowledgeable person, that understands the underlying difficulties and therefore the value and possibilities of having such a simulation. To them, the flashiness of the explosions in a brand new videogame or an innovative experience has definitely a higher impact. Therefore, a reason for developing this demo was to allow users to interact directly with the simulation inside a virtual world, which would allow them to fully experience the advantages of a simulated character over a keyframed one, while providing an enjoyable and rich new experience.

Figure 1.1: *Screenshot from the Trials on Tatooine demo, with first player perspective wielding a lightsaber.*

This would also allow us to explore new emerging technologies, such as the recent Virtual Reality (VR) trend, and show that it is possible to apply research done in the field of simulated controllers to practical and fun experiences.

1.4 Structure

As this project involves multiple domains, I took this opportunity to learn as much as possible from each of them. Therefore, this thesis is concerned with the stages of the design of an interactive experience, the software engineering aspect of modeling the system in a clear, concise and understandable way, the application of well-known design patterns to keep the system simple, the architecture of the individual subsystems that compose the whole program. These subsystems include user input, physics, animation, score, in-game objectives, AI, and many others. The thesis also focuses of course on actual programming, problem solving, experimentation of new interaction styles (starting from a common keyboard and mouse setup and ending in extravagant rigs made out of cardboard to use with the virtual reality controllers) and game mechanics (from a simple shooter, through physical simulations to more advanced balancing mechanics), as well as a *lot* of rather simple 3D math problem solving - most of which aren't too complex on their own, but the sheer amount of them, all the corner cases and coordinate changes and representation are challenging when put together. The variety in the tasks has a lot of upsides: mainly, all the experience that can be gained in practical tools, skills, and methodology in working over multiple domains in such an environment.

Since the thesis consists of a series of tasks, each one relying on the previous, with the goal of producing at the end a fully working demo, this report is also structured in a similar manner. For each task the description of the problem is first given, together with an explanation of the underlying concepts that might be useful, and finally the solution or implementation is explained.

In **chapter 3**, we start by discussing differences in control algorithms, such those between active and passive controllers. We analyze why there is the need to have a ragdoll system as well as the already existing active controller, and discuss the different levels at which an active controller can work. We will cover ragdoll physics: what they are, how they were implemented, and the differences between the built-in system in Unity and the one implemented in our engine. We will look at different types of joints and how they work in detail. We will also see how to deal with problems that the constraint solver might cause, such as erroneous force generated due to unfavorable situations and a solution that helped reduce and dampen these "explosions". We will also have a quick look at a few techniques used to create a basic artificial intelligence for the droid. These techniques include Finite State Machines, NavMeshes and autonomously moving game agents.

To create the interactive experience described above, we need to import the necessary tools into the game engine. This includes resolving all incompatibility problems and setting up the environment to work with the external tools. **Chapter 4** of this thesis deals with that. An overview of different format types to store skeletons and articulated bodies is given, as an understanding of how they work is needed in order to convert between them. This allows the interoperability between Unity and our library. We then explore how it's possible to skin a mesh and apply the transformations from one format in Aikido all the way to the final result in Unity. The mapping between different types of skeletons, such as KS (Kinematic Skeletons) and articulated characters such as ARBS (Articulated RigidBody System) and the problems that must be solved in order to achieve this are also discussed. The last part deals with the interface needed to connect code coming from different environments and languages, and have them work in synergy. At this point the simulation will be working inside of Unity. We won't be using Unity's integrated physics, opting instead to use the physics engine present in our codebase to permit greater control over it. This chapter also discusses in depth how it was possible to create a seamless interface that allows rigidbodies created in Unity to be fully working in an external physics engine out of the box, and the many challenges we had to overcome in order to make this possible. We also deal with conversion issues between different types of coordinate systems and rotation representations. We will also treat some of the more bizarre bugs we encountered and the developed solution for them (other than Aspirin).

In **chapter 5** we will discuss how Virtual Reality works, what is the underlying technology that enables it, and how to include it in our project. We will look at plugins that allow this, how they work and how to set them up. We will also see how to write scripts that allow more complex functions inside of virtual reality environment. We also talk about augmented reality and how to include it in our projects inside of Unity.

In **chapter 6**, we venture into game mechanics territory and look at a few of them, such as force generation and the everpresent shooting. We also offer some design patterns that optimize performance based on our goals. Other game mechanics include interacting with virtual objects using the controllers, like picking up objects, cubes and balls, throwing them around, juggling them and much more. Other fun mechanics such as teleporting are also briefly discussed. We see how all of this is accomplished using a mix of scripts that deal with the virtual reality side and input detection, together with physics simulation libraries. This chapter also concludes the part regarding my work on this project. It includes our latest experimentation, such as porting the application to other platforms and devices. We discuss different frameworks and what must be done in order to have them work together. We also look at fundamental incompatibilities that hindered some of our goals. We finally look at other game mechanics that showcase how versatile this controller is for game applications. These mechanics include character balancing on tilting platforms and a small playable game that is the end product of this project. We also look at more advanced topics in the physics engine, the differences between Unity's physics and ODE, and how it's possible to solve some of the problems that arise due to the use of one engine rather than the other. Finally, we explain how our AR demo works and showcase some of the mechanics implemented in this minigame.

Chapter 7 concludes this thesis by providing further ideas on possible game mechanics for anyone interested in exploring other interaction possibilities. It also includes insights on possible approaches and tweaks to other parts of the game that could have also been implemented. The last part of this chapter includes all the learning the developing of this thesis provided, and the main takeaways.

1.5 Acknowledgments

I would like to thank *Martin Guay*, for giving me the opportunity to follow this exciting thesis and gain practical experience working for him at Disney Research, for his patience in teaching me the fundamentals, as well as following my progress and being supportive and full of knowledge along the way. Not to mention all the support in writing this thesis and improving it.
Dominik Borer, that helped me many times in solving coding problems and better understand the framework in which I was working, and in providing useful insight on possible solutions as well as learning resources. I don't think I could have made it without his help.
Bob Sumner, for supervising this thesis and its development.
Alessia Marra and *Maurizio Nitti* for providing all the models and modifying them to fit our needs, as well as crafting the scenes in which our demos take place, and providing tips on how to work in 3D modeling software. The game wouldn't look half as good without their skilled help.
Mattia Ryffel, for his technical support and suggestions on how to improve the demos.
Also, I would like to thank all the people working at Disney Research that helped out or tried the demo, providing feedback.
And finally you, the reader, for taking time to read this thesis.

1.5.1 Final Words

I hope you will enjoy your reading and that some of the things mentioned might be unknown to you, so that your invested time in reading it will be fruitful in learning something new.

2

Related Work

This chapter discusses related work in the fields of procedural animation and biped locomotion. Procedural animation refers to animation clips generated at runtime or altered, in contrast to data driven animation, such as keyframing, in which the static data is exported from an animation tool. These procedural systems produce at runtime dynamic animations that fit the environment and unscripted perturbations. Some of these systems start from pre-recorded or keyframed animation clips, as the generation of believable motion without any reference is even harder. A few examples in which no pre-recorded animation of any form is used are also discussed. These systems are inherently more difficult to create as by not having any reference motion, the controller cannot tweak an existing clip to achieve the desired result. It must compute every joint rotation and bone position from scratch. Controllers that use this approach often exhibit behaviour that looks quite different from what you would expect from a human. This often because the constraints the system must satisfy don't model human behaviour. Such systems include the Google DeepMind controller. Simulation controllers can further be split into two categories: passive controllers, such as ragdoll, and active, which are under the direct control of the system. Examples for both of them are given in the next sections. Controls to which procedural animation systems can be applied include foot placement [Johansen, 2009], arm reaching and punch swinging. Some of the techniques used are Inverse Kinematics (IK), and the application of physical simulations that allows characters to react to hits, falls and much more [Stelian Coros, 2010]).

2.1 A brief history of animation systems

Animation systems evolved over the years. Since 1988, the year in which skeletal animation was introduced, different control strategies have been developed. Firstly, there simply was one animation clip that was played. The joints inside the skeleton followed the transforms stored in

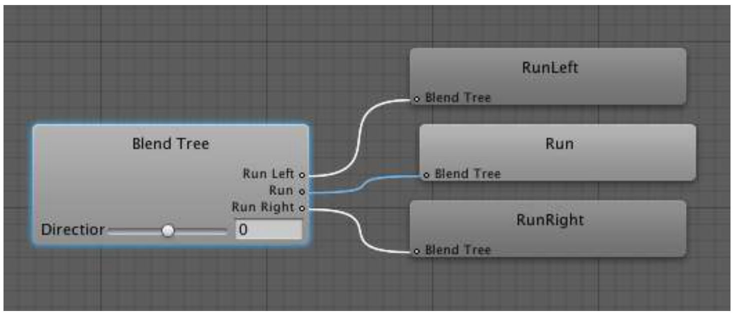


Figure 2.1: *A blend tree with a walking state in which a master parameter controls the direction (-1 = left, 0 = straight, +1 = right) and the three different motion clips that are interpolated accordingly. Screenshot taken from Unity.*

memory, and the character moved accordingly. Then *Finite State Machines* (FSM) came. The gameplay programmer could define which state a character was currently in, and the animation system would automatically play the corresponding animation, and loop it indefinitely as long as the state didn't change. However, when the state was changed, the animations were abruptly swapped. This was solved when *Blend Trees*, or *Decision Trees* came. The idea was to have a certain number of synced animation clips, such as walking left, straight and right, and based on a parameter such as the desired direction of motion, or the angle of an analog stick controlled by the player, interpolate between the clips to obtain a weighted average that better matched the desired result.

Then *Parametric Blends* came, also called *Partial-skeleton Blending* or *Additive Blending*. The idea is to blend different animations using different weights for various bodyparts. It is now possible to blend an animation of a character performing one action with his upperbody while the lowerbody still follows another animation such as walking or running. This allows to generate complex and realistic motion by having just a few clips. Examples include having the character aim in various direction while moving independently, or turning its head to where the player is looking at while performing a given task. This is now all charted territory and research is done to bring new innovations to the dining table.

2.2 Kinematic Controllers

This section treats the most commonly used animation techniques used today in commercial games and applications. The standard animation system used in game engines such as Unity are Finite State Machines and Blend Trees, which were briefly mentioned in the previous section. These models offer transitions between predefined clips, which must be well aligned [**?**] and looping. They are similar to motion graphs [Kovar et al., 2002]. The requirement for the clips of being well aligned and looping adds a lot of overhead work for digital artists, that have to manually tweak the clips and build motion graphs, which grow exponentially in size with the number of different animation clips.

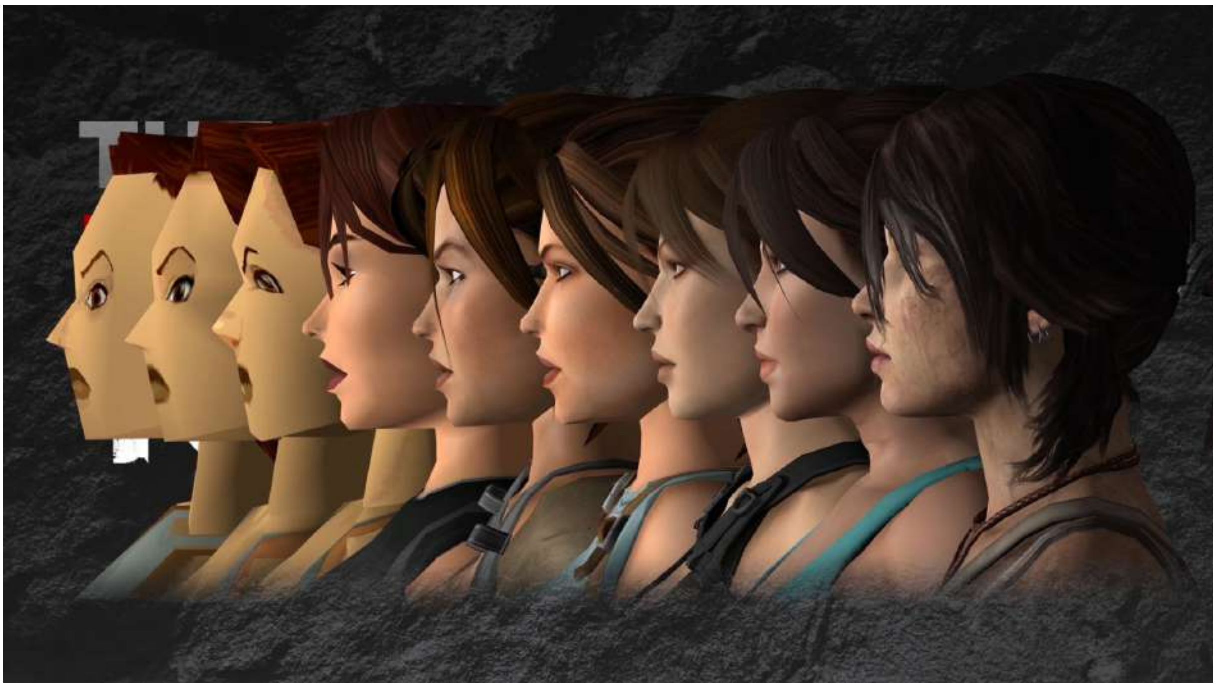


Figure 2.2: *The evolution of one of the most influential and long lasting videogame series: Tomb Raider. The evolution of the model of the notorious archaeologist from the first game in 1996 until today is well remarkable, both in the mesh and the textures, as well as lightning and hair. Note than already in 1996 skeletal animation was used, as it was first introduced in 1988*

2.2.1 Blend Trees and State Machines

Videogames are one of the fastest growing entertainment media industry. Much of the research done in field of procedural animation is applied to them, as videogames are interactive mediums by nature. In recent years, the progress made in some of the related games fields, such as rendering, has been enormous, and the results can be easily seen by comparing recent videogames to those of only 5 or 10 years ago.

However, animation systems haven't undergone such a dramatic change, and this causes expertly modeled and textured characters to move poorly within the game environment, looking ungrounded and often even broken. Much of the research done nowadays aims at fixing this by providing more sophisticated yet simple to use algorithms and controllers. Many proprietary controllers are slowly coming to the market, that can simply be plugged in the animation system like a black box and provide tweaks to the existing animation improving it. Examples of this are Euphoria by Natural Motion, Puppet Master, Final IK and IKinema. Game engines such as Unity and Unreal provide state-of-the-art rendering capabilities and therefore stunning graphics. Many indie games nowadays look simply amazing even without a huge team working on the graphics. However, the animation systems of game engines are still based on age old state machines with blend trees. Therefore to expand on this is necessary to overwrite the animation system. Example of this are more advanced videogames that use extensively what is called complex *LERP Blending*, in which animation clips are mapped to a higher dimensional space, defined by variables such as speed and orientation in the case of biped locomotion. Then, based on the desired state of the character, the closest animation clips are blended in such a way that

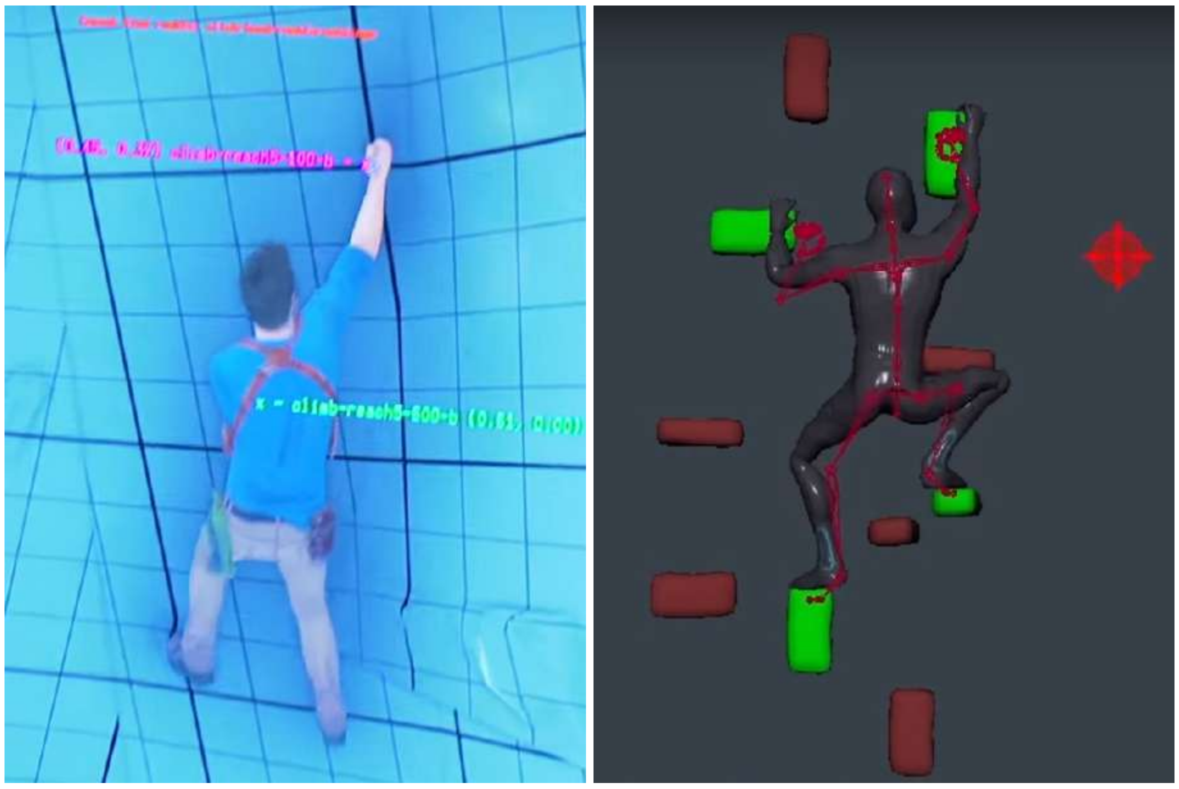


Figure 2.3: Left: Uncharted 4 climbing system. Right: IKinema demo in which avatar is adapted to climbing points.

the resulting motion will be believable. This is the child of blend trees. Some of the techniques previously mentioned are also used, such as *Partial Skeleton Blending* and *Additive Blending*. Examples include when a character has to point in a direction and keep walking. Having all possible combinations of walking and pointing is an inefficient solution, and this technique allows to blend different parts of the skeleton to permit the generation of a multitude of final animations. You can read more about these techniques in [Gregory, 2014].

2.2.2 IK Retargeting

As videogames grow in scale and ambition, and as new game mechanics and features are introduced, always more is required of the animation system in a game engine on which the games are based. Hardware improvement in consoles and PC also allows the implementation of more complex control algorithms, further incentivizing research. Example of this is the recent transition from the PlayStation 3 to the PlayStation 4, in which a substantial increase in hardware performance caused in developers the desire to implement more complex systems. For example, in the recent **Uncharted 4**, a new climbing system was developed. In the game, the player can have his character climb impervious mountains and cliffs. The sheer amount of possible disposition of grappling points positioned by the level designers would require a prohibitive amount of different animations to have hands and feet always positioned at the right point while at the same time responding to the user input (the character leans and puts hands in the direction in which the player tilts the joystick), so a new system was developed.

One possible way to tackle this problem is by having a reference animation clip, such as climbing, that is then adapted at runtime to the climbing points. It can be seen in the above figure (right) that the final pose of the character's mesh doesn't perfectly match the skeleton of the reference motion in pink: this is because *end effectors* are applied to each limb, and are dynamically positioned to the climbing edges by the gameplay system. The animation system then proceeds to use IK modifiers to remap the animation clip to have the end effectors match as closely as possible the desired position. Therefore the resulting animation still resembles the recorded clip, but is made procedural and fits the requirements. Retargeting the clips using layered inverse kinematics (IK) reduces drastically the amount of hand-crafted different clips.

Examples of such systems are IKinema (link) and Final IK (link to Unity asset store).

Another very useful application of such controllers is in the adaptation of foot placement based on the underlying terrain. This is treated extensively in [Johansen, 2009]. Here the controller automatically adapts the foot rotation and inclination to fit to the terrain, to appear grounded as well as avoiding the very well known sliding feet problem. The result of such controllers are best viewed when the character walks on stairs or terrain with different slopes, which in videogames are frequent occurrences. The controller in the paper is also very flexible, as it can be applied to a multitude of different skeletons with different morphology, such as quadrupeds other than humanoids.

2.2.3 Motion Matching and Deep Neural Networks

In Ubisoft's latest hit, the game For Honor, a control system called Motion Matching is introduced. The idea behind it is to have raw data of motion capture, and when a particular motion is needed, to find the best snippet within that data. In traditional animation system, actors would perform many different motions in a mocap (motion capture) studio. Such motions include movements like idle, walk, jog, run, sneak, climb and fall. These captured animation would then be imported in a 3D software, and artists would start working on it. They would cut clips for every different gait and type of motion, speed and direction. They would polish the data and then create loops for each clip while also synchronizing them. Then complex state machines were created with complex rules for transitions, blending and such. This is obviously a complex and time demanding system to implement.

In the motion matching approach, the artists don't edit raw data. All the data - usually about 10-15 minutes of unedited motion - is stored into a single big file. The mocap data is stored in an unstructured list of motion, and each frame the animation system tries to find which pose in this raw data best fits for the next frame. Based on the player's stick input, the desired direction of the character is computed and the algorithm finds the best piece of motion for the next frames that brings the character in that direction. In the end what motion matching does is stitch together a lot of small clips to generate a responsive controller. This is suited as well to respond to perturbation events. If the motion capture data contains reaction to pushes and hits, the algorithm will automatically select those pieces of motion when a similar perturbation is applied. This is a relatively simple idea that has the potential to create complex motion. To speed up the algorithm, many intensive data pre-processing computations can be carried out before the game is shipped. These computations create meta-data, called "motion fields" that connect similar motions on the motion file, that represent possible transitions. Then at runtime

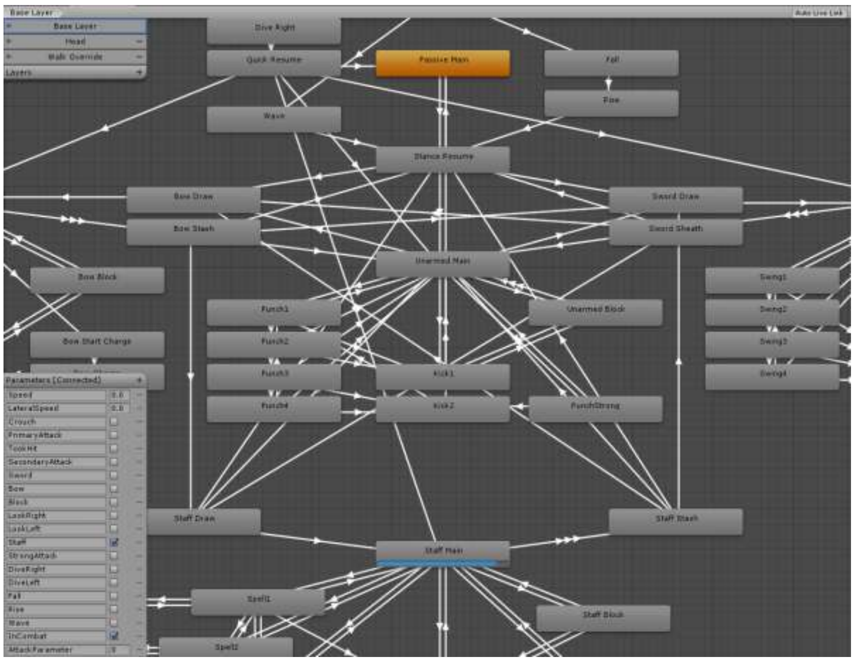


Figure 2.4: *A screenshot of a complex state machine that arises when many different motions are available for a character.*

the algorithm knows which clips are available as next choice.

More about the underlying algorithm is described in [Lucas Kovar, 2002] and [Yongjoon Lee, 2010].

Neural Networks

Different approaches have also been experimented. A notable one is that obtained at Google DeepMind. Agents learned to walk, run, jump and turn to navigate in diverse and complex environments. This behaviour emerged from reinforcement learning, meaning that a reward function gave signals to the agents about which performed behaviours were the best. The paper emphasizes how the novelty of the approach is that rather to design specific reward functions to obtain the desired behaviour: by using different environment condition a robust behavior emerges on its own. This is a complete different approach from all the previously mentioned other ones, all of which produced the resulting motion starting from predefined clips and used a model to tweak them. The approach described in the paper is much more general and was applied to locomotion because it is a notoriously hard task, but could have been applied to other areas as well. However, I believe it is also important to know of these different approaches. The paper is [Nicolas Heess, 2017], and a video with the results can be seen here (youtube link).

2.3 Simulated Characters

It is also possible to increase the realism of character motion by embedding them in a real time simulation. This approach has been developed for many years, with research as early as

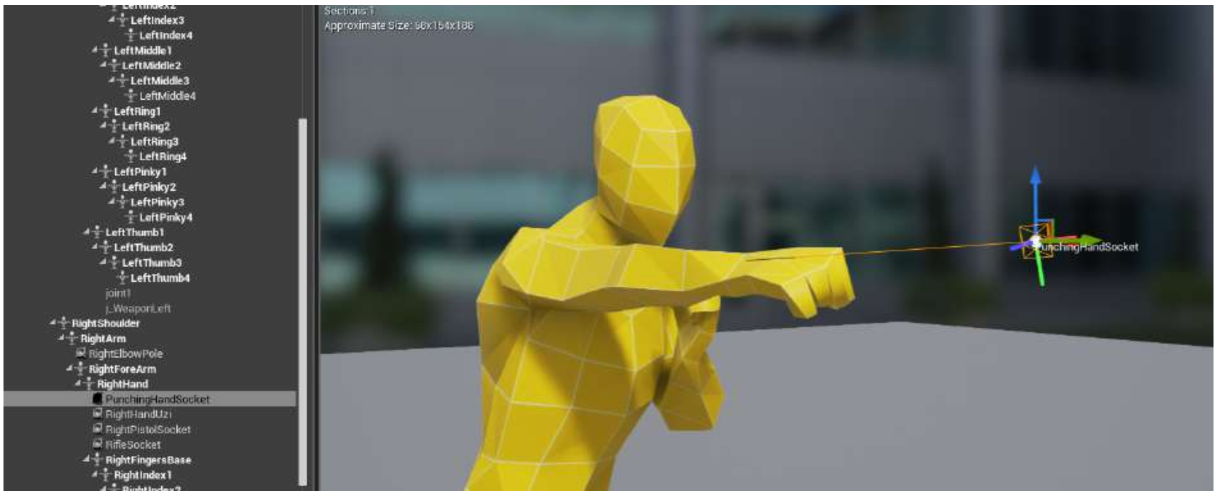


Figure 2.5: *The IKinema system in Unreal Engine with the fist trying to follow its target transform.*

[Hodgins et al., 1995]. Different approaches have been tried, such as the inverted pendulum [Coros et al., 2010] [Yin et al., 2007]. These controllers require a lot of parameter tuning to get the simulation to behave correctly and realistically.

2.3.1 Passive Characters: Ragdolls

Another popular controller is the blend of actual animation clips and physical simulations, such as ragdolls. The ragdoll was introduced in videogames to simulate the fall of a limp body when one character died. The playback of a few prerecorded animations would quickly become repetitive, and the sheer amount of different ways for a limp character to interact with its surroundings while falling meant that a new approach was needed.

In this type of simulation, a ragdoll follows the animation clip as closely as possible, with added constraints that prevent object interpenetration. This uses the underlying collision detection and physics system. The skinned mesh is then attached to the ragdoll. What this approach provides is a greater realism and feeling of weight. If a character is animated to swing a punch, then as the fist collides with its target, the ragdoll embedded in the physical simulation will prevent the hand from penetrating the target. This differs from traditional animation systems, as they just follow the keyframes in a clip. The hit and sudden change in velocity will propagate through the arm and the ragdoll, and the resulting motion will appear much more realistic. It is also possible to apply forces to the ragdoll, such as when being hit by a punch, which will also propagate through the character body altering the final pose. This is made possible by joints that connect the different parts of the body. These topics are treated more extensively in chapter 5.

Combinations of these methods provide the best results, as can be seen in the Puppet Master online videos. (link to Unity's asset store videos)

Euphoria

Euphoria is a middleware produced by Natural Motion, that also synthesizes animations at run-time, by combining physics, AI and biomechanics. Their team includes *behaviour engineers*, that study how character can respond to different situations and create behaviour modules that can be then used by the character. Their system looks very realistic as it doesn't just use a ragdoll simulation, but also different types of motion such as self preservation (using the hands to protect itself when falling, or reacting when being shot at). It was used in AAA games such as Red Dead Redemption, Max Payne 3 and GTA IV. Here it is possible to see their system in action (youtube video of Euphoria demo) and their respective website (Natural Motion website).

2.3.2 Active Controllers: SIMBICON

Active controllers are a system in which an underlying algorithm computes the desired motion that the character should follow based on the task at hand. A simple way to think about passive and active controllers is the following: when you are active, and want to reach a certain place or perform a determinate action, you are producing motion by means of muscle contraction on your own self. The "algorithm" that controls this exhorted motion is deep integrated in your brain, and the producing of this motion comes natural to you. You don't have to think about the single muscles you need to contract to keep balance. On the other hand, if you stop trying to stand upright and just let you fall on the ground, or on a softer surface, you won't be exhorting any force and your body will just crumble, with joints and bones still maintaining certain constraints. Active controllers are conceptually a different approach than passive ones, and often try to solve a different underlying goal.

The controller on which this Bachelor Thesis is based is described in [Stelian Coros, 2010]. It was implemented by Dominik Borer for his semester thesis [Borer, 2016] and is part of the high performance library called Aikido. Everything described in the following chapters uses this algorithm as its underlying controller.

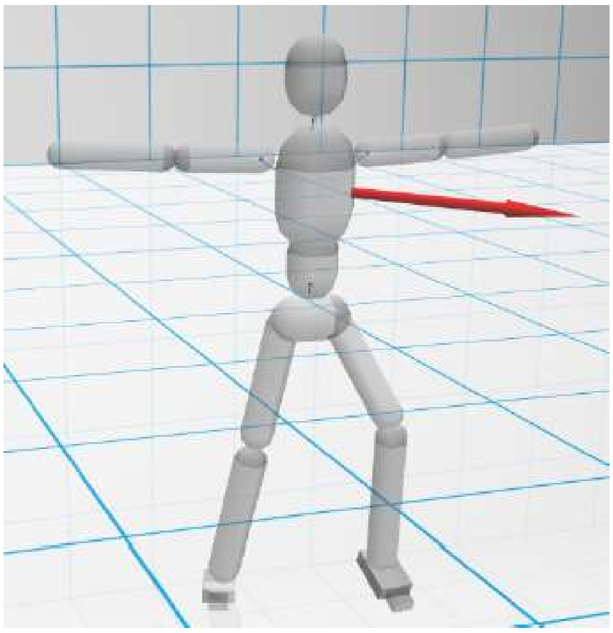


Figure 2.6: *A screenshot of the application implementing the SIMBICON controller. Taken from [Borer, 2016].*

3

Controller and AI

We start this chapter discussing how in an interactive application such as a videogame we need to include different types of controllers, based on which state the simulated character is in.

3.1 Differences Between Active and Passive Controllers

We distinguish between two types of controllers: *passive controllers*, such as ragdolls, in which only a physical simulation with constraints is run for the character, and *active controllers*, in which and underlying algorithm produces suitable forces and torques to simulate the action of muscles based on some defined objective. These can be further split into high and low level active controllers: the former defines some high level goal, such as motion generation to reach a distant position, whereas the latter tries to solve a short time goal such as finding the suitable placement of the next step to keep balance and avoid falling.

3.2 Passive Controllers

In this section we will discuss the specific needs for the passive simulation needed in our application. The passive simulation must complement the strengths of the active controller described earlier, such that in the end the resulting animation produced in both the active and in the passive simulation results coherent and applicable inside of the demos.

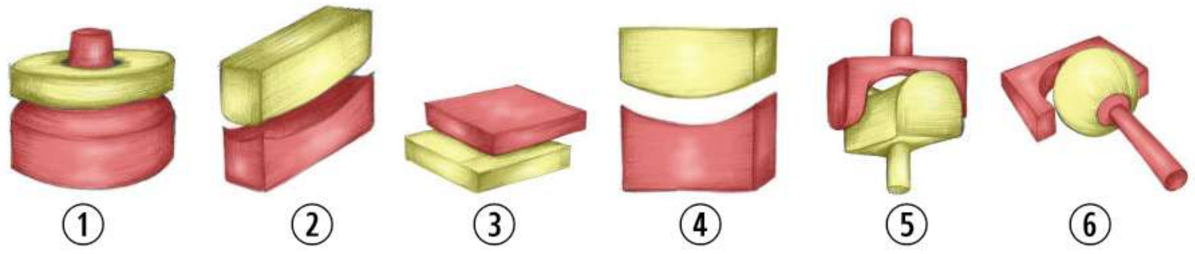


Figure 3.1: *The different types of joint found in the human body. They are:* ***1.*** *Pivot joint* ***2.*** *Condyloid joint* ***3.*** *Gliding joint* ***4.*** *Hinge joint* ***5.*** *Saddle joint* ***6.*** *Ball and socket joint. Image taken from Teach PE's website.*

3.2.1 Ragdolls in our Simulation

When the droid was hit too hard in our simulation, it would fall. However, due to the underlying controller that tried to move the legs in a suitable position to keep the body in balance, and the fact that the character had already fallen, the resulting motion was very weird, almost retarded. The character on the ground was moving frenetically its legs in all directions. This was definitely breaking the impression of intelligence and realism, and the solution was to disable the controller. But only disabling the controller would have frozen the character, so a ragdoll system had to be implemented to continue the motion.

3.2.2 Current State

Even nowadays, no algorithm exists that closely mimics the skeletal and muscular structure of the human body and that can therefore closely simulate its fall or its being pushed around. However, many generalizations can be made, that allow less complex models that still look relatively realistic to be created. This is the topic that will concern us in this section.

3.2.3 Types of Joints

In the human body there are a lot of different joints, often categorized into 6 different types, as shown in figure 3.1.

For example, the pivot joint is present at the top of the neck, the hinge joint in elbows and knees, and the ball and socket joint in the shoulder and the hip. A simplification was made, and the only used joint types in our model are hinge, saddle (also called universal) and ball-and-socket joints. How these joints are modeled as constraints can be seen in figure 3.2.

These joints differ in the amount of degrees of freedom they have. In the following table you can see how many DOF each of these 3 types of joint has.

In this chapter we will explore more advanced topics, such as ragdoll physics, with a focus on joints and constraints, how to break them and the problems that arise from a fully simulated character. We will also have a brief excursion into AI terrain, talking about FSM, chasing

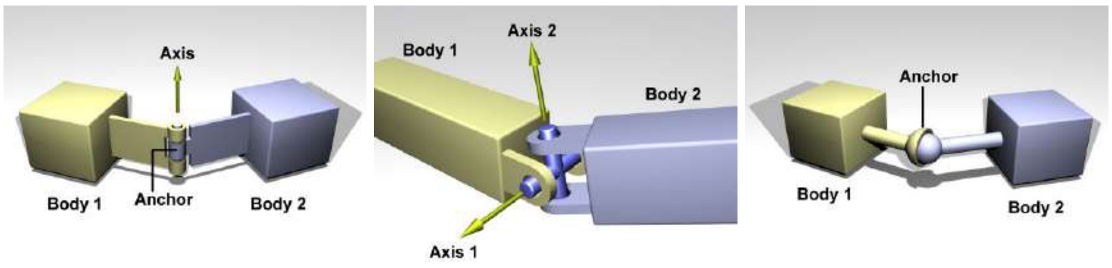


Figure 3.2: *different types of joints available inside the Open Dynamics Engine (ODE). They are, left to right: Hinge joint, Universal joint, Ball-and-socket joint. Notice similarities between the human body equivalents above (joint types 4-6). Taken from the Open Dynamics Engine website*

Table 3.1: *Amount of DOF per Joint type*

Joint type	# DOF
Hinge	1
Universal	2
Ball and socket	3

behaviour and NavMeshes.

3.2.4 Turning into Passive Mode

To be able to obtain coherent motion when the active controller couldn't handle the motion anymore, such as when the droid had fallen, we had to disable the active controller and activate a passive simulation. In the following sections we will discuss the creation of the ragdoll system and how it was activated when needed to take the place of the active controller. We will also discuss problems and challenges that arose and our solutions for them.

3.2.5 Creation of the Ragdoll

To create the ragdoll, I set up appropriate constraints to make the falling of the character look realistic. Without constraints, the rigid bodies wouldn't be attached, and therefore the simulation would resemble a pile of bones falling all disconnected from each other. However, if you add constraints but make them too generous, meaning you allow a great angle of flexibility for the joints, what happens is that when the character falls, he crumbles upon himself and therefore looks very unrealistic, as if he didn't have a spine and was completely flexible, like a mass of jelly without joints and tendons. A lot of tendons connecting muscles are present in the human body, and due to the maximum stretch each supports before tearing, many constraint that stiffen up the movement when the body is limp are present. Therefore a lot of time was spent analyzing existing ragdoll system and their constraints. Since different skeletons between different characters and developers often have a different structure - in the number of bones, how they

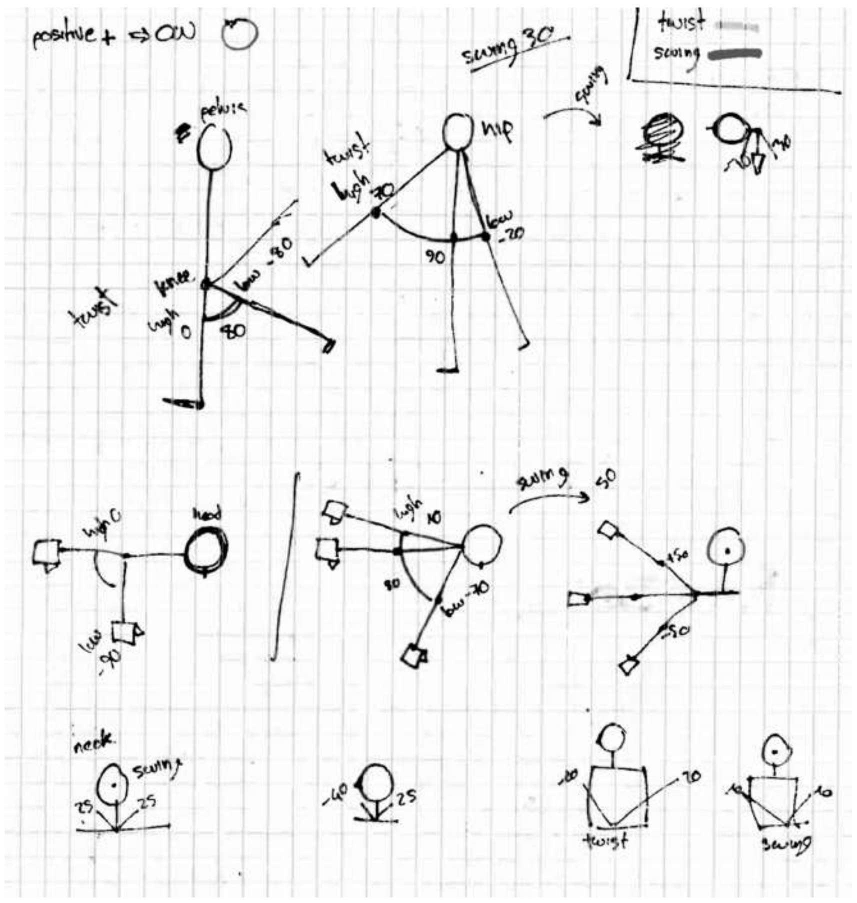


Figure 3.3: Initial sketch of suitable angles of rotation for all the joints in the model seen from different angles. Both swing and twist axes are marked.

are connected, the respective weights, etc - it doesn't exist a "one size fits all". After some early prototypes and a lot of tweaking, I found a combination of angle limitations for the different joints that looked good.

3.2.6 Unity Representation

Unity provides a built-in system to simulate ragdolls: since they are primarily composed of rigid bodies and constraints, what this system does is to create a skeleton made of existing objects, each assigned to a limb, with appropriate weights and connect them up with different types of joints. The wizard for creating this came very useful to test how their algorithm worked in selecting the different joint limits for the angles of the different axes. However, since it isn't open source, it wasn't possible to directly access the code of how it was computed. The way Unity represents the different types of joint is very interesting: instead of defining specific types (such as hinge and universal, like in ODE) in separate components, all of them have all 3 axes of rotation available, and are therefore technically ball-and-socket joints. However, for each of the 3 axes (twist axis, swing axis 1, swing axis 2, as they are called in Unity), it is possible to set low and high-twist (or swing) values that limit the range of motion.

These values limit the rotation in the respective axis. With these limitations it is possible to

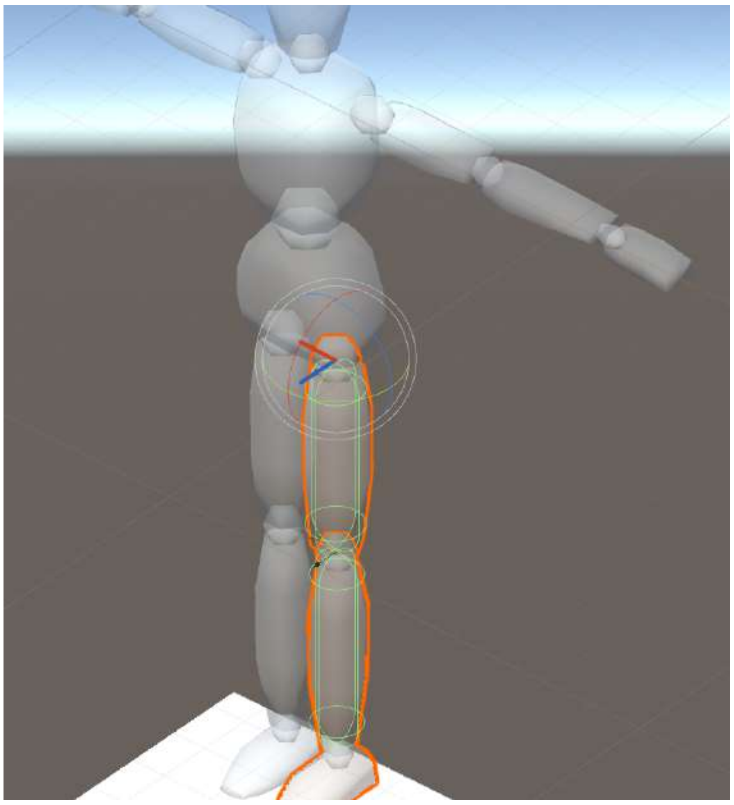


Figure 3.4: *A ragdoll in Unity in which can be seen the underlying collision primitives and joint axes. As they were tiny, the picture is retouched to made the axes bold.*

Table 3.2: *How to create joint types using Unity's only joint: "×" means setting both limits to 0 (disallowing movement on that axis)*

Axis	twist	swing 1	swing 2
pivot		×	×
hinge	×		×
universal	×		
ball and socket			

create the other two types of joints as well, and create realistic constraints for ragdolls: for example, to create a pivot joint, simply set the allowed angle for both swing axes to 0. Similarly, to create a hinge joint, set one swing and the twist axis limits to 0.

The ragdoll was setup as a RBS file, in which the rigid bodies matched the SIMBICON character's ARBS and the joints were tweaked to reflect the new limits.

3.2.7 Triggering

There needs to be a well defined point in time in which the controller is disabled and the ragdoll simulation takes over. Since the reason for implementing a ragdoll simulation is that the controller doesn't work well when the body of the ragdoll is very close to the ground, a logical solution is to set a threshold for the character's body height relative to the ground and then detect when it is lower. This is a very simple rule that allows for automatic detection of when to activate the ragdoll mode.

In practice this simplicity works very well, and by trying different parameters for the trigger height we found one that was suitable. When this condition is triggered, the controller is deactivated, the ragdoll loaded and posed to match the state in the last frame of the controller. From this point on the physical simulation takes over.

3.2.8 Explosion Problems

As mentioned in the previous section, to have a ragdoll appear plausible, it is important to set joint limits that aren't too broad. However, the controller needs a lot of freedom to be able to rotate the hips joints and knees in order to not be limited in the placement of the feet. Therefore, when the controller was disabled and the ragdoll simulation took over, it could happen that the constraints for one joint weren't satisfied the initial moment. This caused the solver for the joints to generate a force directly proportional to the amount by which the joint constraint wasn't satisfied. This meant that if the leg was widely extended, a great force was generated to bring it back within the allowed range in the shortest time possible. Unfortunately, when the droid was falling, he would very often have one leg before the other, and therefore in falling he would start performing a split. However, as soon as the triggering happened, the great inward force for both legs caused the simulation to snap the legs back together. This in turn generated an upward force to avoid interpenetration with the ground. And this upward force was so great due to the speed at which the legs were being closed that the ragdoll would often explode in the air. This was because the constraint solver had very stiff constants. We had very little control over the SIMBICON prior to the transition to the ragdoll state, and what this implied was that we couldn't prevent unfavorable positions when the transition happened.

To solve this problem, the ragdoll velocity was artificially dampened for a few frames after the transition, to ensure that the explosion wouldn't happen.

3.3 Active Controllers

Active controllers were briefly explained in the previous chapter. In this section we delve in the specific inner-workings of the active controller which is use throughout this project to provide the reader with some context.

3.3.1 Low Level: SIMBICON

This controller uses a different approach to allow a biped model to walk. An inverted pendulum model is used to compute foot placement to prevent the body of the character from falling. Intuitively this is the same as when somebody tries to balance a pencil vertically on the open palm of the hand. The underlying hand has to move in a certain fashion to balance the pen and prevent the fall. This is exactly what happens with the feet placement. It also uses a PD control to try to match a target trajectory provided by a motion generator, which can be set to work with different walking styles. This produces suitable torques that will be applied to the character to enable the walking motion.

3.3.2 High Level: AI with Global Goals

Up until now, the SIMBICON character would simply walk straight ahead. This was obviously not so engaging for the player. To add a simple AI to it, we decided that the character should be able to chase the player and, when close enough to him, punch him.

I implemented a simple FSM (Finite State Machine) to control the state of the droid. He could either be chasing or punching the player. The transition between states were simple thresholds with the distance to its target. An image for this basic FSM is shown in figure 3.5.

When chasing the player, although it had to go in his direction, aiming for the current position would result in a poor chasing behavior, as the continuous movement of the player would make the droid look irresponsive. If you observe an agent who is chasing another, like a child chasing a ball, he usually doesn't aim for its current position, but rather for the position that he predicts the ball will be when he will reach it, and makes adjustments as he gets closer. This approach shortens the time the chaser needs to reach its target and provides the illusion of intelligence. The book "Game AI by Example" [Buckland, 2004] explains many types of movement for

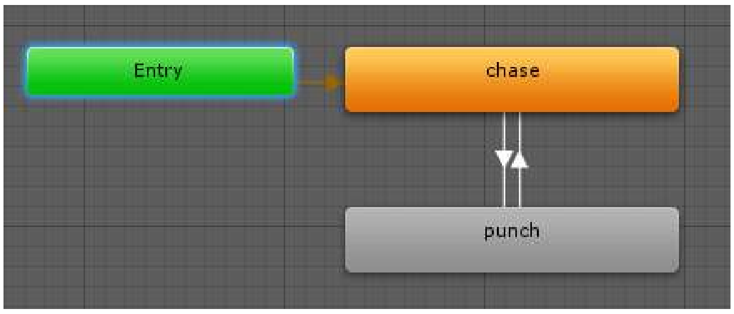


Figure 3.5: The simple logic that controls the behaviour of the droid. It begins in the chase state, and as soon as the target is within range, a transition is performed to the punch state. If the target escapes, chase mode is entered again.

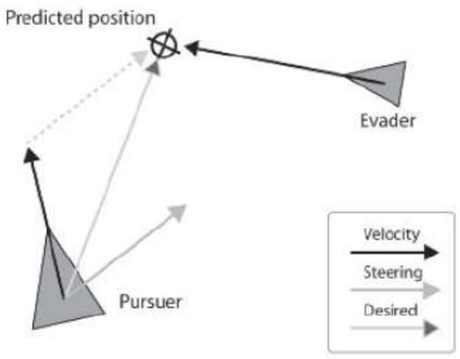
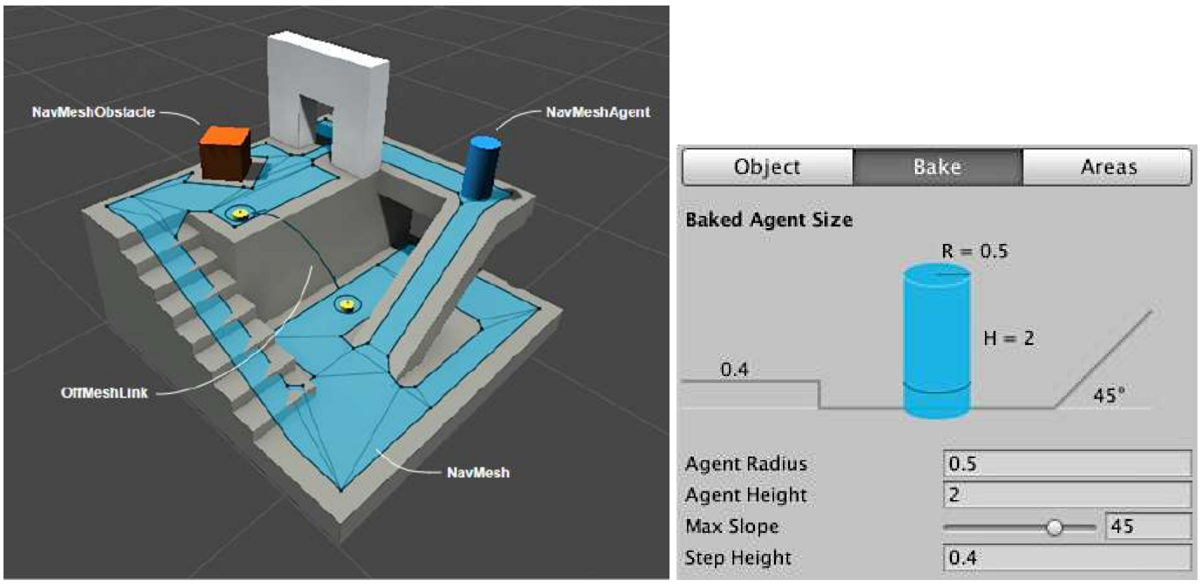


Figure 3.6: How to compute the steering velocity of the pursuer given its current and desired velocity. Taken from [Buckland, 2004].

AI, including this one. This position is computed automatically, based on the selected target position, velocity and distance.

Another thing to consider is that sudden changes in the player's position or velocity caused the droid to quickly re-adjust its target position. This inside the simulation meant a sudden shift of weight, often so quick that the droid had no time to react and place its foot accordingly, and therefore fall. In the real world, if you are running and must suddenly change direction, you take a bit of time to shift weight and readjust without falling. This problem was solved by learning the maximum changes in direction and speed the controller supported, and then smoothly damping the different motions to be within the boundaries. This added a very small amount of lag to the controller but allowed the character to continue its chasing. Also, by limiting the maximum turning rate of the droid, it was possible to make it look more realistic by avoiding sudden rotations. The previously high level component I programmed to communicate to the controller all the SIMBICON parameters such as step width, heading, walk cycle duration and desired velocity, provided an easy way to apply the desired parameters to the underlying controller.



Agent Radius 0.5
Agent Height 2
Max Slope 45
Step Height 0.4

Figure 3.7: *In this figure the automatically generated NavMesh for a given map can be seen in light blue. The NavMesh is generated by computing the walkable area while keeping into account several factors, such as the agent's radius, height, and maximum step height and slope. These parameters were configured to match the SIMBICON controller's.*

3.3.3 NavMesh

Since the scene contains obstacles, by simply having the droid chase the target position, there was a high chance that he would collide with its environment. To add obstacle avoidance, I used Unity's built-in NavMesh system.

A NavMesh is a flat mesh that represent the walkable area: this can be computed at compile time, or baked before the game is run since most obstacles are static. This allows NPC to have a well defined area for their movements which will ensure no collision with the environment. To reach a specific point, the following steps are taken:

- if the point is outside the NavMesh, it's projected to the nearest point within the bounds. This ensures that the droid will only try to reach points on walkable area.
- the triangles in which the droid and its target find themselves in are determined.
- the dual graph of the NavMesh is then used. Each area is now represented by a point, and connected areas have an edge between them in the dual graph. By knowing the start and end areas (computed in the previous step), we know the start and end vertex in the graph, and any graph search algorithm (like the popular A* using an euclidean distance heuristics) can be applied to find a suitable shortest path.
- The segments from the actor and target to the center of their area are added, and the path can now be smoothed to yield the final result.

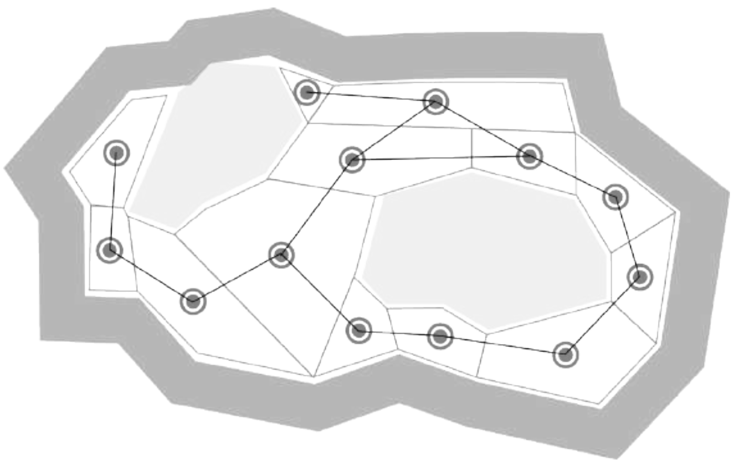


Figure 3.8: *The dual graph of the NavMesh, which is used in conjuction to graph algorithms to answer queries.*

3.3.4 Chasing the Player

Due to the path smoothing, sometimes the target would overshoot its target rotation, causing it to endlessly spin. I fixed this by adding an extra level of indirection between the droid and the player: the droid had a target constrained to be within a maximum distance from itself. This target was equipped with a NavMesh agent, as described above. The advantage of this approach was that the behaviour of the droid could be controlled by programming how the target capsule behaved, therefore the AI was completely decoupled from the droid.

Capsule System

The capsule system that enabled the decoupling from the high-level planning AI for the navigation, which is tightly coupled to Unity's NavMesh is shown in figure 3.9.

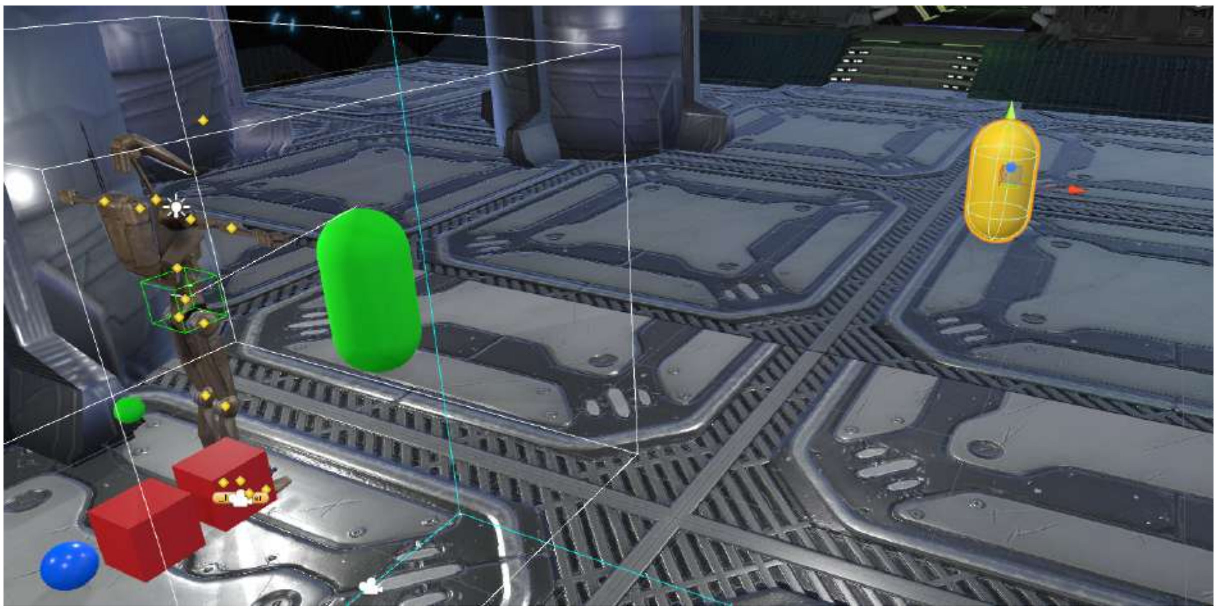


Figure 3.9: *System of capsules for droid. The green and yellow capsule are normally hidden, but here they are shown to explain how the AI works. The droid closely follows the green capsule, which is constrained to be within 1 meter of the droid. This green capsule seeks the yellow capsule, which is the target, and is equipped with a NavMesh agent component, which permits to avoid obstacles. The yellow capsule possesses a versatile script Player.cs that controls the chasing behaviour depending on how it's set to move. It can, for example, copy the position of the target, or move following a certain pattern. This indirection system allows the droid to be very flexible and exhibit different behaviours without having to directly tweak its controller.*

4

Wrapping Aikido into Unity

In this and in the next chapters I will explain all the steps that are needed to go from a standalone application in C++ to an interactive game in virtual reality.

In this chapter, it is explained how to transfer the transforms from one representation in the simulation, the ARBS to the representation used in game engines, the KS. We will see how to skin a mesh and display it on top of the skeleton, then loading the SIMBICON in Unity, and updating the skeleton state to move like in the simulation. This will provide us with the ability to work inside Unity with all the algorithms and controllers implemented in the Aikido library. We will also discuss how the physics interface in Unity was connected to the existing engine, and all the conversion and calls that had to take place to enable this.

Different Representations For this first part regarding my work on the project, the goal was to display a mesh on top of the SIMBICON character, that was presently only composed of capsules. The mesh needed to be skinned and follow the underlying skeleton. There were also incompatibilities between the different representations of the character to solve. On one hand, the simulation uses jointed rigid bodies - called ARBS (Articulated Rigid Body System) - to drive the simulation. The simulation consisted therefore in transforms for each rigid piece and constraints that limited the allowed position and rotation they could have. On the other hand, traditional skeletons and skinned meshes use joints, and utilize only their rotation to compute skinning matrices to deform the mesh. This means that computations have to be performed to go from the set of rigid bodies to the joint orientations, as well as map which joint takes data from which rigid piece. Then everything had to be imported in Unity, the skinned mesh, its skeleton and the underlying controller, and each frame the data from the simulation had to be imported in Unity and applied to the skinned mesh. After doing this we could start using the advanced features such as input, VR, better rendering, deployability to multiple platforms and much more, which are treated in the next chapters.

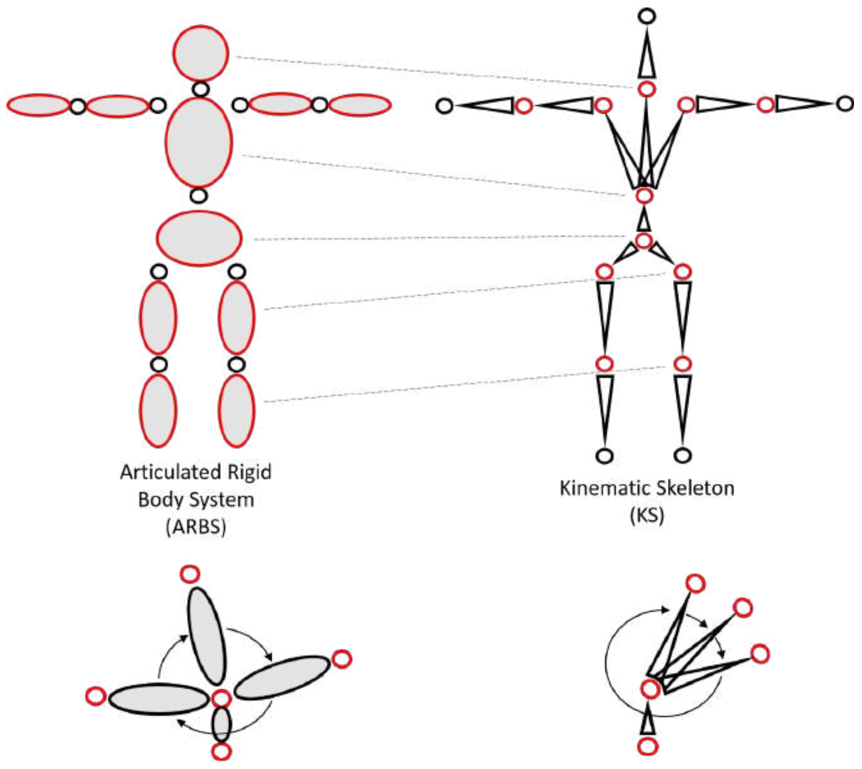


Figure 4.1: *The ARBS (left) and KS (right) for a humanoid biped. Notice the similar appearance but different underlying structure in the joints and bones/rigid bodies. The marked lines represent the mapping from rigid bodies to joints. Note that the KS is the dual of the ARBS and vice versa. In the figure under it can be seen how some configurations can be reached by the ARBS but not by the KS, since the former has more freedom than the latter.*

4.1 Overview of the system

The main idea behind this system is the following: we want to be able to simulate characters in a physical environment. This requires a simulation in which collision is taken into account, meaning that our character must have collision primitives representing its body geometry. These rigid bodies are both simulated and under the control of the SIMBICON algorithm, which can apply forces and torques to them to simulate the action of muscles. The Articulated rigid body System (ARBS) is a format used to represent a collection of rigid bodies connected with joints that are simulated. At each frame, the resulting position and orientation of the rigid bodies must be extracted and applied to a skeleton, so that a skinned mesh connected to the skeleton can be visualized with the corresponding deformation and animation. Game engines and motion authoring software such as Maya use a particular hierarchical structure to represent skeletons composed of joints and bones. This representation is called a Kinematic Skeleton (KS). The problem is the inherent different representation of the articulated rigid body system used on the simulation side with respect to the kinematic skeleton used to deform the visible mesh.

The problem is that although both ARBS and KS have joints, in ARBS the joints don't possess transforms, and are instead just constraints that limit the space in which the rigid bodies can move. In the KS however, the joints are the ones possessing a transform. Usually just the rotation is used as no translation or scaling is allowed. Translating joints would cause a stretch in the underlying bones, which are actually just the space between parented joints. A direct consequence of this fact is that the skeleton cannot be broken into pieces. Moving joints far away doesn't break the bones, like one would imagine if working on the ARBS. Here, if the joints contraints weren't present, by moving the rigid bodies far away one could break the bones. However, in the KS, moving the joints causes the space between parented joint to change and therefore the bone, which is nothing more than the connection between two joints, will be scaled and so will the visible mesh.

A naive approach would be to map joints in the ARBS to joints in the KS, but this is not desirable as then the ARBS could reach states not applicable to the KS. This can be seen in figure 4.1. A much more refined mapping is to have rigid bodies in the ARBS map to joints in the KS. Using this approach the transforms have to be converted from one representation to the other since in ARBS all the transforms are in world space, whereas in the KS joints are relative to their parent, hence in local coordinates (except for the root). The algorithm for computing the final positions of all the individual joints in the KS from the position and orientation of the rigid bodies in the ARBS was already in place. It just required a suitable mapping between the "limbs" or rigid bodies in the ARBS and the joints in the KS. That's what I did in this first part of the project.

4.2 Transfer Between ARBS and KS

This section discusses how to transfer motion from one representation, namely that of Articulated Rigid Body System (ARBS) to the most widely used in commercial engines of the Kinematic Skeleton (KS). We will start discussing the formats in which they are stored.

4.2.1 Kinematic Skeleton

Autodesk .fbx is a complex file format used to store meshes, skeletons, animations and scenes. Since it's a proprietary format, Autodesk provides a SDK (software development kit) to convert from and to this format. This file format is a hierarchical collection of nodes, each of which contains its name and a series of attributes, or properties, which are tuples of fundamental data types. It can also recursively contain other nodes. This recursive structure is therefore well suited to store tree-like structures, such as skeletons or animation clip data. This format is widely used to store the Kinematic Skeletons we will be using throughout this project. Knowledge of this format was taken from here.

4.2.2 Articulated Rigid Body System

Another file format the reader should know about is the RBS (Rigid Body System), which we will mention many times. It is often referred to as ARBS (Articulated Rigid Body System). This isn't an official file format, but rather is a simple text file with .rbs extension used to store a collection of rigid bodies, each of which can contain several properties, such as mass, position and orientation. Joint information can also be stored, effectively enabling the storing of skeletons, in which bones are rigid bodies, as well as simple collections of unconnected rigid bodies or collision detection primitives (CDP).

Note: this duality of joints and rigid bodies as bones is troublesome due to the usual representation in game engines that differs from what we use in our simulation. Therefore algorithms have to be developed to convert poses from one representation to the other.

4.2.3 Our Character Models

Using a 3D modeling software - like Blender or Autodesk Maya - it is possible to model a mesh, rig it (the process of adding bones) and then skin it (assigning vertex weights for each bone), export it to the FBX file format and then use this file on our side. In the codebase were already available a few simple applications to view and load the content of the FBX and RBS files in the environment. A few problems and crashes arose, due to the importer settings and the structure of the file. To fix them, I made sure that the named hierarchy matched exactly and that at least one keyframe of animation was present, even when no animation was needed. This was the initial bind pose, or T-pose. This is needed to compute the inverse skinning matrices for the deformations. Characters are often modeled and rigged with their arm straight to the sides to allow easier texturing and skinning.

The model we used in the first part of the project was a Star Wars themed droid, the B1 Battle Droid, freely downloaded from sketchfab.com under the Creative Commons Attribution 4.0 License. This indeed is the droid we're looking for.

Later however we will start using another mesh model for the droid, that better suited the look we were going for. This character is named Deezee and can be seen in figure 4.4.

4.3 Applying Transfomations to the Mesh

The existing code SIMBICON was showing its output on a humanoid composed of simple geometric primitives, such as capsules and boxes. They followed a named hierarchical structure, which can be seen in figure 4.3. All primitives are capsules except for the feet and toes, which are cuboids to better adapt to the terrain.

These primitives served both as meshes for the rendering, as well as collision detection primitives (CDP) and bones inside the skeleton. Since a skeleton needs information on how the different bones are connected, joints were used to constraint the position of the different CDP and attach them together. In the rightmost image of figure 4.3, all the named joints are shown. It is possible to see from the legend that there are three distinct types of joints. The difference between them is the number of degrees of freedom (DOF) they have. How joints work and how these constraints are implemented will be treated later on.

Figure 4.2: *The mesh of the droid (textured and posed) that was used in the game. Taken from sketchfab.com*

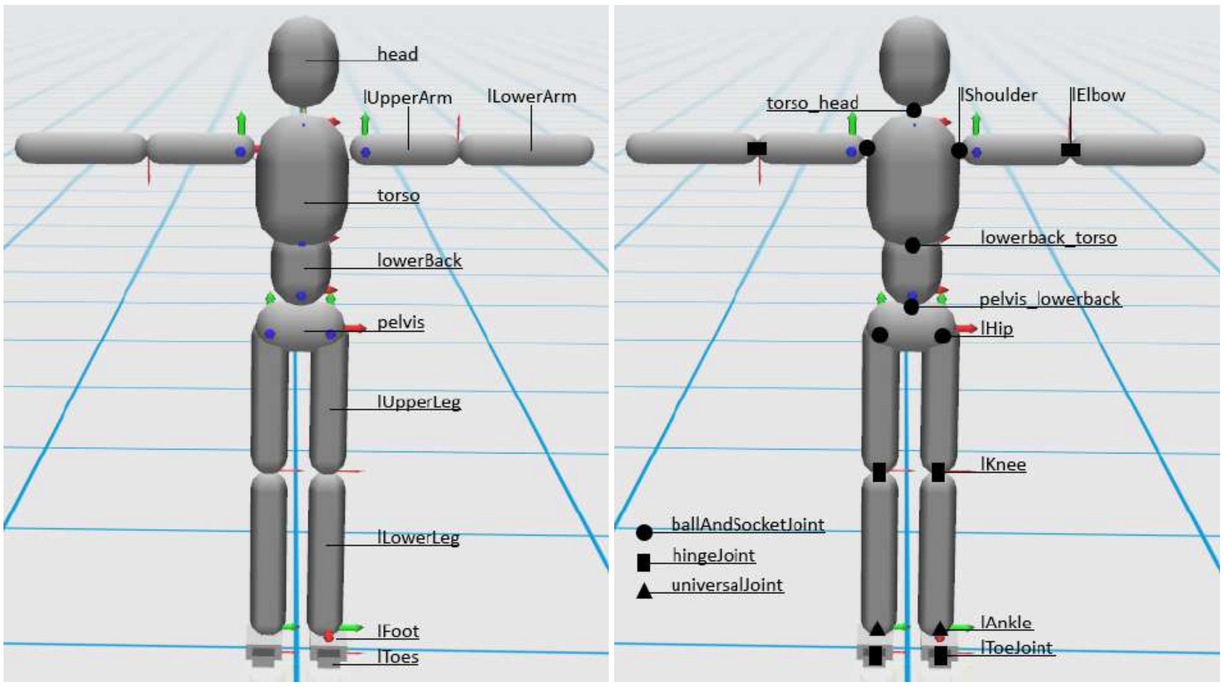


Figure 4.3: *The underlying robot inside of the SIMBICON simulation*

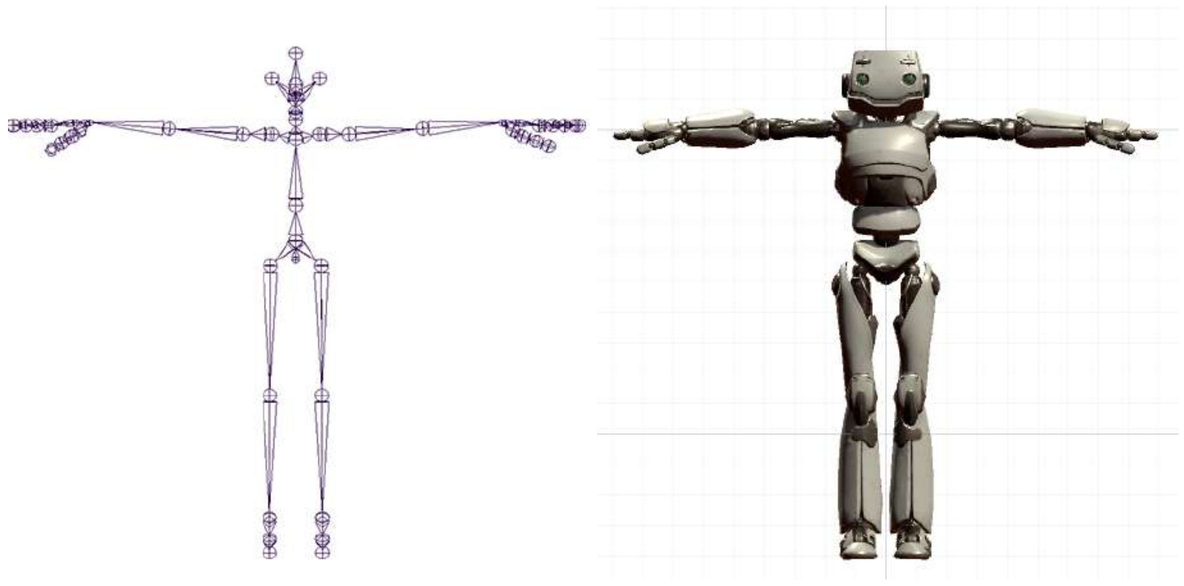


Figure 4.4: *Left, the skeleton made up of connected octahedral bones. Right, the 3D mesh to which it is applied to provide a deformation.*

4.3.1 Skinning

To have a rigid mesh deform according to some pose or animation, two components are needed: a static 3D mesh of the model, and a skeleton composed of bones and joints: then a mapping has to be done to constraint the motion of the mesh from its enormous number of DOF (each vertex has its own position) down to a more manageable one, typically of the skeleton. This process is called skinning. By assigning a set of weights to each vertex relative to the bones, when the bones change position the new position of the vertex is computed by blending the different transforms of the bones according to the weights. Linear blend skinning (LBS) is a very simple skinning technique. However how it works is outside of the scope of this thesis, so it won't be treated. Many game engines limit the amount of weights per vertex to 4, since rarely a single point is influenced by more than 4 different bones. This allows to reduce storage and use simpler datastructures (an array instead of lists) and speed up the computation by a generous amount.

4.3.2 Mapping

In figure 4.4 you can see a mapping of the character between joints and bones.

One notable remark must be done here: the underlying difference between how skeletons work and how the rigid body system implements the articulated character.

As previously stated, a skeleton is usually nothing more than a hierarchical collection of rigid pieces known as joints. Starting from the root (usually defined to be the pelvis), all the attached joints are children of their parent recursively. A joint in a skeleton represents a transform which can be translated and rotated, but usually not scaled (to preserve the rigidness). These transforms are manipulated by the artists to create animations, and used by the game engine to apply animation to skinning meshes. The transforms of the joints are stored in local frame, relative to their parent. When the world position of a joint is required, a walk up the tree ending in the root is performed, and at each node the transform is multiplied by the transform matrix of the current node. This yields at the end the world position of a particular joint.

Since bones don't stretch, usually joints are constrained to only rotate and not translate. If a joint were to be translated, the distance to their parent joint would change. This implies that the visual mesh would stretch. On the other hand, the way a rigid body system is setup is to contain both rigid pieces (rigid bodies) which are equipped of a transform, and joints, in this case different from what previously defined. These joints aren't transforms now, but rather indications of how two rigid bodies are positioned with respect to each other

4.3.3 Optimizing the Performance

Note that, in order to improve the performance, instead of walking up to the root at each frame and computing a matrix multiplication at each node, an efficient design pattern can be used to avoid unnecessary computations and improve performance. This is the *Chain of responsibility pattern*, in which precomputations are stored at each node. They are marked as out of date with a *dirty bit* only when the transform changes. When walking up a tree, only the needed computations must be carried out in a lazy manner, avoiding the exponential amount of computation necessary and therefore improving performance by a lot. The caching mechanism built into this pattern allows children of a node to avoid computing multiple times the same matrix multiplications.

4.4 Turning Native Aikido into Managed Aikido

Up until now we had been working on the *C++ side*, meaning all the code compiled inside of Visual Studio, and ran in its own application. The first step to make the SIMBICON more widely available was to transfer it to a commercial and well known development framework, and one of the most popular nowadays is Unity 3D (webpage). However, since all of the code was written in unmanaged C++, and Unity allows to only write custom scripts in the managed language C# (and also JavaScript and Boo), there was a need to make this code callable from inside of Unity.

4.4.1 Dinamic-Link Libraries

What we did was to compile all of the C++ code into a DLL (Dinamic-Link Library), meaning all the functions were now inside a .dll file, and they could be called from a different environments, such as the Unity's C#. By loading this file as an asset in Unity, we then wrote a script that provided an interface to the necessary functions inside of the DLL, and then directly use this interface from other C# scripts like if they were written natively inside of Unity.

4.4.2 Open Dynamics Engine

In the C++ code, the physics engine used is the Open Dynamics Engine (ODE), by Russel Smith (link). This is a freely available open source library that can be used to simulate rigid bodies, and has a well defined C++ API. It also supports collision detection and many types of constraints, such as joints. Therefore, any user wanting to implement a physics simulation in his application or game, without deep knowledge of the underlying equations and numerical methods for solving them, can start simulating, applying forces, constraints and much more to its objects by calling the appropriate functions in this library. This comes in handy. The SIMBICON simulation uses this library for its physics simulation. However, to avoid unnecessary coupling to this particular library, one subtle yet very useful design pattern is used: the facade pattern.

4.4.3 Facade Design Pattern

In this design pattern, the interaction between two different subsystems is decoupled (made as independent as possible, so that modification on one side will have the least effect on the other) by defining a simple interface between the client and the provider. This allows for the client to access the much more complex underlying interface without having any knowledge about it. The main advantage in our case, is that modification in the ODE code and library will result in modification on our side only in the facade file, not everywhere in our project where access to these functions is needed. It also provides a very convenient way to swap the underlying library for physics simulation (for example by using the Bullet physics engine, or any other) and still have only one file to modify to have everything work flawlessly.

4.5 Interface with Unity

The way Unity allows to call unmanaged code from within its environment is very specific. All the functions inside of the DLL file have to be "wrapped" into a C# file to provide the caller in other files and classes with the necessary functionalities. Unity works exactly this way under the hood: since it's a commercial product, and not an open source one, Unity doesn't distribute its source code, which is also written in C++ to allow greater flexibility and be more performant. However, it still has to provide users of its framework access to the necessary functions. Therefore, Unity ships with a few .dll files of its own, such as *UnityEditor.dll*, or *UnityEngine.dll*, which contain all the particular functions for that subsystem. The code that can be seen is the one that calls into those DLLs at specific points, which is basically just an interface. This is a great reference to see what is possible to do natively.

In my code, there is one file, *AikidoAPI.cs*, which exposes all the necessary functions of the DLL and deals with calling the appropriate entry points in the file. This is great because once one function is setup it's easy to call it, just like you would do with a normal function.

4.6 Implementing the Physics Interface

We will now focus on integrating physics from an external library and using a simple interface that allows us to use the high performing Aikido library for most of what is displayed inside of Unity. The advantage of using Unity's integrated physics engine is that it is as easy as adding a component to the desired objects (a rigid body component) and then Unity takes care of all the rest. The interface is simple, but the object simulated in one physics engine obviously cannot interact with one simulated in another, like the ODE we were using. So it was time to switch from one to the other. Up until now, to add a physical object it was necessary to create a text file with extension .rbs, and then fill in all the parameters for the object in question, such as initial position and orientation, mass, moment of inertia, size and more. On the other hand, Unity provides a great graphical interface to create objects, drag them around in 3D space and see where they actually are, as well as modifying parameters such as mass, and all the derived quantities such as moment of inertia (MOI) will be calculated automatically based on other parameters such as shape or material. It also allows to select, view and modify the collision primitive to be used.

The interface I implemented uses the power of this already existing graphical interface, called Gizmos, to position objects dynamically in the scene, and to fit them inside the physics engine.

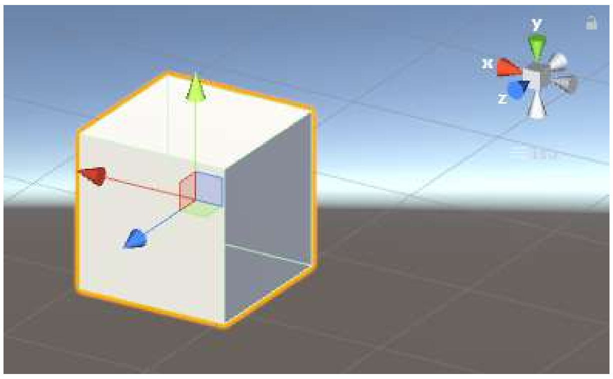
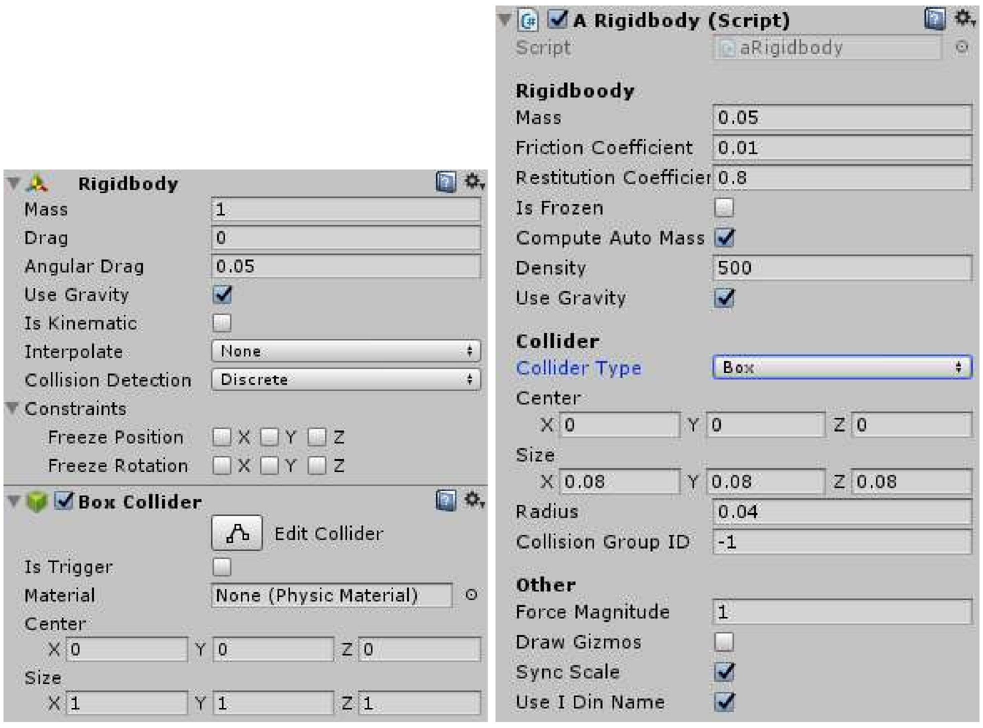


Figure 4.5: *Visual Gizmos for object manipulation in Unity.*

By applying this component (called *arigid body.cs*, short for "Aikido rigid body" - I know, imaginative..) it is possible to control all the necessary parameters. Some of them are mass, coefficients for friction and elastic restitution, if the object is kinematic (called "is Frozen" in ODE), and options to specify if the mass should be computed automatically based on the object size and density. This component also includes parameters to specify the collision primitives used for this object in the collision detection engine. The user can choose between a handful of primitives, such as cube or parallelepiped, sphere, capsule and plane. The script can also automatically fit a primitive to best match the object mesh. This script, which is a central component of the synergy between Unity and Aikido, automatically computes all the derived values, and then writes to file a .rbs file containing all the parameters. When the application is launched, this file is retrieved and loaded into ODE, all automatically without any tweak from the user. It is now possible to update it in real time inside of Unity. The script takes the computed state inside of ODE and applies it to the object in Unity. The script can also be queried for additional information regarding the rigid body, like in any physics engine, like angular velocity for example. Also changing variables such as mass or if the object is kinematic at runtime is possible, as well as adding external forces. The script defers much of what it isn't strictly required until needed, in a lazy evaluation fashion, and this permits to simulate more objects since each one is less expensive. The script also takes care of initializing the necessary variables not written to file inside of the ODE. In my early tests, the objects were simulated correctly but no matter where



```
Rigidbody
Mass                  1
Drag                  0
Angular Drag          0.05
Use Gravity           ✓
Is Kinematic          ☐
Interpolate           None
Collision Detection   Discrete
Constraints
  Freeze Position     ☐ X ☐ Y ☐ Z
  Freeze Rotation     ☐ X ☐ Y ☐ Z

✓ Box Collider
                      Edit Collider
Is Trigger            ☐
Material              None (Physic Material)
Center
  X 0    Y 0    Z 0
Size
  X 1    Y 1    Z 1
```

```
✓ A Rigidbody (Script)
Script                    aRigidbody

Rigidboody
Mass                      0.05
Friction Coefficient      0.01
Restitution Coefficiei    0.8
Is Frozen                 ☐
Compute Auto Mass         ✓
Density                   500
Use Gravity               ✓

Collider
Collider Type             Box
Center
  X 0       Y 0       Z 0
Size
  X 0.08    Y 0.08    Z 0.08
Radius                    0.04
Collision Group ID        -1

Other
Force Magnitude           1
Draw Gizmos               ☐
Sync Scale                ✓
Use I Din Name            ✓
```

Figure 4.6: *Left: the component in Unity that allows the manipulation of a rigid body in the physics engine and a Box collider used in to detect collisions. Right: my custom component that allows both the setup of rigid body properties as well as different types of colliders.*

I placed them, they would always change position. This was caused by data races, because the ODE was overwriting the initial position and orientation.

4.6.1 Data Races

Many problems arose due to the use of external functions inside the DLL. The most prominent were caused by data races between functions called from the DLL and others inside the managed portion of the scripts in Unity. Often one call to a function in the DLL would ask for a specific resource, but on the ODE side that resource still wasn't loaded, resulting in a crash. Inside of Unity, there are a multitude of predefined functions that are called from the framework. It is very important to understand how they work and in which order they are called such that one can engineer the calls in a way that makes sense and that avoids a faulty order. In Unity, it's possible to write scripts that inherit from the native object *MonoBehaviour*. All these objects are stored internally by Unity in a collection. When the game is started, functions such as *Awake()* and *Start()* are called on each object to allow initialization. However, it is not possible to know which Awake() will be called first on a given set of objects, therefore if dependencies are present this could result in data races and faulty program behaviour. Therefore it is important to decouple as much as possible and use different functions to make sure that order is respected. For instance, Awake() is always called before Start() on all objects. There are many other functions available to use, such as *FixedUpdate()* for changes that happen before the Unity's internal physics update. After that, all functions that check for triggers going off and collisions are automatically called, such as *OnCollisionEnter()*. Input events are processed after that, as well as game logic, such as the Coroutine calls and animation update, including *Update()*. Since in this project we mainly tweaked the physics and animation, respecting function order becomes apparent as a fundamental requirement to have the system work. These are highlighted in figure 4.7. Many other functions for scene rendering and decommissioning are also available. This figure depicts the full lifecycle of a MonoBehaviour object. It is also worth mentioning that Unity makes a heavy use of a few well known game programming patterns, to be precise sequencing patterns: these include the **Game Loop** pattern and the **Update Method** pattern. If you are interested you can read more about it here in [Nystrom, 2014].

4.7 Axis Orientation and Conversion

One small problem that caused a LOT of troubles in our journey was how different frameworks oriented their axes. Two things influence how they are oriented: the handedness of the system and which axis is considered up. Handedness refers to the direction in which the third axis is oriented with respect to the other two. When fixing the first two axis (X and Y), in right-handed systems the *negative* Z axis points forward, whereas in left-handed systems the *positive* Z axis points forward. This implies that in right-handed systems the positive rotation around one axis is in counterclockwise direction, whereas in left-handed it is clockwise. They are called this way because the orientation of the axes resembles the first 3 fingers on the right hand respective left hand when they are extended orthogonal to each other.

Different programs decide to support different handedness and rotation. It might look like a trivial problem at first glance but the consequences of such a decision are numerous. The industry standard finally settled on right-handedness, although not too many companies enforce this. There isn't a standard for which axis should point upwards, but usually the decision is between the Y or Z axis. I personally find right handedness and Z-axis up the best combination, because much of the mathematical literature uses right handedness and orienting the Z-axis up allows for easy 2D game development as well as the programming of game world position easier, without having to switch the Y and Z components in all vector conversions between 2D and 3D. However, orienting the Y axis up also has its advantages, such as UI coordinates and side-scroller platformer games.

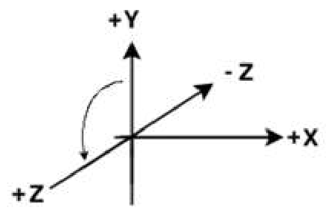


Figure 4.8: *Right-handed axes are shown in this figure.*

It is speculated that Microsoft, when it developed DirectX, deliberately made the system left-handed to discourage conversion between DirectX and OpenGL, its main competitor. This would have made it harder for developers to port code for DirectX to OpenGL. Speculations aside, what matters is that in the end the lack of an enforced standard creates a lot of problems of compatibility between different programs, exported files etc. And our application wasn't an exception to this. The hardest part in solving this problem wasn't conversion of positions, solved by matrix multiplication, but rather conversion of quaternions. Another fact that made conversion from one system to the other harder was how the individual components of the quaternions were stored inside of the four dimensional array. One had the scalar parameter

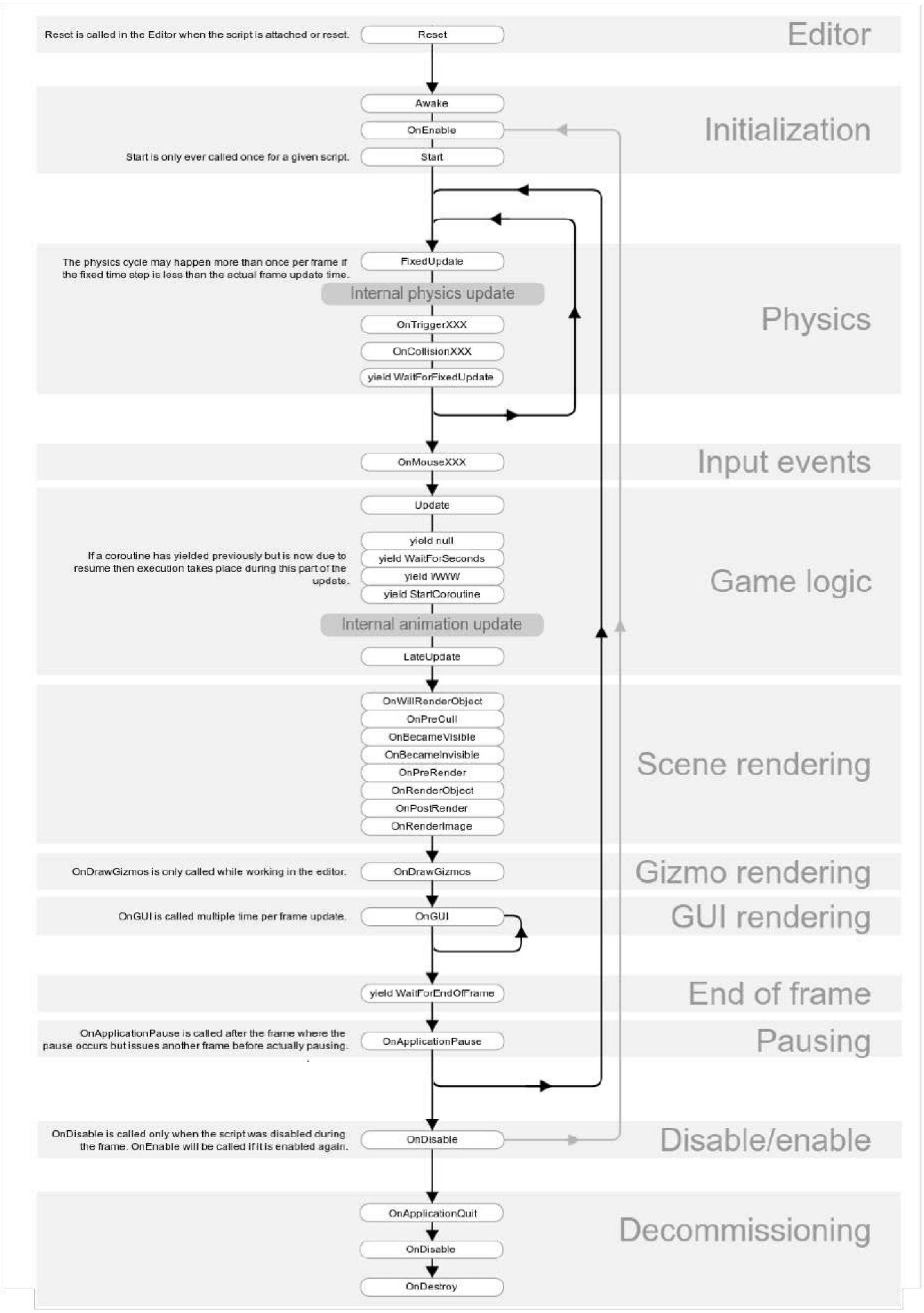

Editor
Reset is called in the Editor when the script is attached or reset.
Reset
Initialization
Awake
OnEnable
Start is only ever called once for a given script.
Start
Physics
The physics cycle may happen more than once per frame if the fixed time step is less than the actual frame update time.
FixedUpdate
Internal physics update
OnTriggerXXX
OnCollisionXXX
yield WaitForFixedUpdate
Input events
OnMouseXXX
Game logic
Update
If a coroutine has yielded previously but is now due to resume then execution takes place during this part of the update.
yield null
yield WaitForSeconds
yield WWW
yield StartCoroutine
Internal animation update
LateUpdate
Scene rendering
OnWillRenderObject
OnPreCull
OnBecameVisible
OnBecameInvisible
OnPreRender
OnRenderObject
OnPostRender
OnRenderImage
Gizmo rendering
OnDrawGizmos is only called while working in the editor.
OnDrawGizmos
GUI rendering
OnGUI is called multiple time per frame update.
OnGUI
End of frame
yield WaitForEndOfFrame
Pausing
OnApplicationPause is called after the frame where the pause occurs but issues another frame before actually pausing.
OnApplicationPause
Disable/enable
OnDisable is called only when the script was disabled during the frame. OnEnable will be called if it is enabled again.
OnDisable
Decommissioning
OnApplicationQuit
OnDisable
OnDestroy

Figure 4.7: *The sequence in which functions are called on each script inheriting from Unity's MonoBehaviour class. Its understanding is very important as it avoids a lot of bugs related to data races.*

Table 4.1: *The order in which the different component of a quaternion are stored in the 4 dimensional array in Unity and in the Aikido Library*

	Unity	Aikido
Q[0]	x	w
Q[1]	y	x
Q[2]	z	y
Q[3]	w	z

w in its first array position, whereas the other had it in its last. It's hard to say that one is better, since positioning the w at the end makes sense, as then in both 2D and 3D vectors, as well as quaternions, the order of components is x, y, z, w. I find this more suited to enforce consistency. However, in the literature usually a quaternion lists its scalar component w first, and follows therefore the alphabetical order. Therefore this difference has also to be taken into account.

4.7.1 Handedness Bugs

One bug that caused a lot of head scratching was a particularly nasty one: at this point we had a simulation in which the droid walked straight inside of Unity. We now wanted to be able to see the response to dynamic perturbations such as when a ball was thrown at the droid. This worked fine for the first seconds of the simulation, however after the droid moved a few meters the thrown balls would just go through it, as if it were a ghost. We figured it was some bug with the collision system so we went ahead and tried to fix something that actually wasn't broken. The problem was actually that, due to the different axis orientation, the underlying simulation was correct but the rendering of the mesh was mirrored. Since the droid was slowly drifting left, the mesh was displayed drifting right. However, the underlying collision primitives were also going left, so by trying to hit the visual mesh no collision was happening. One could see a collision with an invisible object by aiming the ball to the left of the droid. We fixed this bug once we visualized it in the original apples, in which the rendering showed the droid drifting left and not right. This was the last bug in our conversion system, and once it was fixed by showing the underlying skeleton in Unity, the simulation worked as it was supposed to.

4.7.2 Pseudovectors

One peculiarity to mention is that of *pseudo*vectors: as it is neatly explained in [Gregory, 2014], the cross product of two normal vectors a and b, $c = a \times b$ doesn't actually produce a normal vector, but rather a *pseudo*vector. The difference is subtle: under the normal transformation that are so frequent in 3D applications, such as translation, rotation and scaling, pseudovectors don't exhibit any difference from normal vectors. But when the underlying basis is changed, like from right-handedness to left-handedness, what needs to be done is to reflect one axis, multiplying it by -1. A normal vector therefore is mirrored along one axis, as you would expect, but a

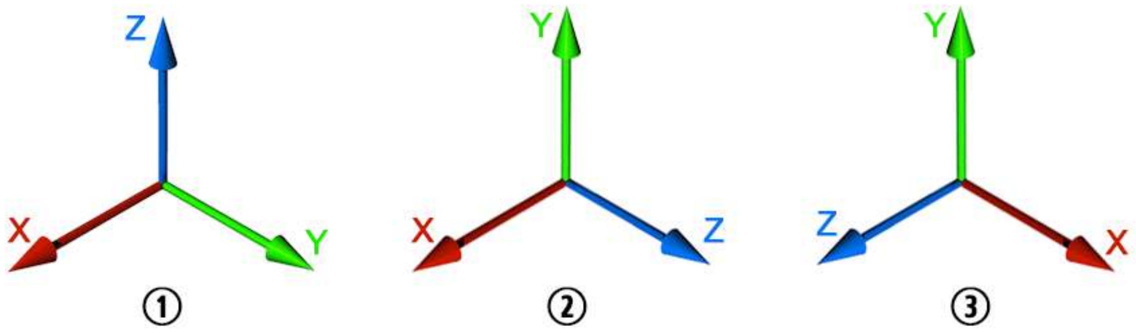


Figure 4.9: *The different ways in which different programs and frameworks orient their axis:* ***1.*** *Blender* ***2.*** *Unity* ***3.*** *Aikido, Maya*

Table 4.2: *By specifying these two parameters, the orientation of all 3 axis are uniquely defined*

Framework	Handedness	Up Axis
Blender	R	Z
Unity	L	Y
Aikido	R	Y

*pseudo*vector must both be mirrored and also change direction, meaning multiplying by -1 the *other* two components. The position and its derived quantities, such as linear velocity, are normal vectors, also called polar vectors, and therefore behave normally. However, angular velocities are *pseudo*vectors, and therefore have to be taken care of accordingly when converting. This small peculiarity resulted in a lot of headaches and funny behaviours when implementing physics: often, if the code for the handedness conversion was faulty, angular velocities were working fine, and vice versa. Once I was absolutely sure that the conversion code couldn't contain any error, by picking up one object and throwing it with the Vive controller inside the simulation, it would spin in the exact opposite direction that you would expect in real life, so this had to be taken care of. It is also interesting to read about the *wedge product*, denoted $a \wedge b$. For all of that, I suggest [Gregory, 2014].

5

Virtual Reality and Augmented Reality

In this section, we will see how VR is introduced in our application, and how we prepared the system to integrate the game mechanics that will be introduced in the next chapters.

5.1 Target Platforms

One of our original goals was to deploy on multitude of platforms while keeping compatibility between all of them.The ability to deploy to a variety of different devices the same application would show how flexible and portable our demo was. As target we had to be able to deploy to Windows 10, both to traditional desktops and the new Surface tablet, as well as to the HTC Vive. We already had the application that could both run on the Vive and on normal desktops, but changing was cumbersome, and the programmer had to manually change a lot of things inside the scene, such as the active camera, the input method, changing the setup of the SteamVR components, and similar tasks. A script was made that could take care of everything by simply selecting the desired platform in a drop-down menu.

5.2 Loading the Application in VR

5.2.1 VR Introduction

Nowadays, VR headset are gaining popularity by the day. Almost everyone has heard of them, although not too many people in the general public have had the possibility to try out one for themselves. This is actually an advantage: by porting our application on this platform, not only is the player immersed in this world, but the novelty aspect of this technology further enhances

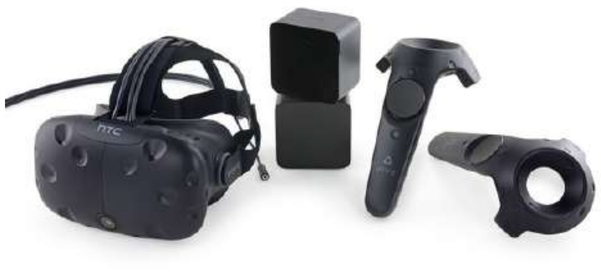


Figure 5.1: *The components of the HTC Vive setup: the headset with integrated screen, the base stations positioned in the room that track the sensors, and two controllers that have both button and position tracking.*

the awe the player feels with regard to our SIMBICON controller. Therefore one of our main goals was to have it working on the most popular of the VR headsets, the HTC Vive.

5.2.2 Technical Aspects

The Vive consists of a headset, two controllers and two base stations. These base stations track both the headset and the two controllers, which in total are equipped with a total of $70 \quad (32 + 2 \times 19)$ infrared sensors, and a gyroscope and accelerometer. Thanks to the high resolution display and low screen and tracking latency, the Vive provides a smooth gaming experience and most important, no sickness feeling if used the right way.

5.3 VR Implementation in Unity

The HTC Vive is developed as a collaboration between the Valve Corporation and the manufacturer HTC. They decided that native support will be given to the Unity platform via a freely available plugin, downloadable directly form the Unity store: SteamVR. This plugin takes care of the tracking of both the headset and controllers, and provides a well organized interface to access their 3D position and orientation, as well as the linear and angular velocity, to read controller inputs, send the video and audio feed from Unity to the headset, and much more. This is a great feature as it allows developers to jump right in and start developing for this platform very quickly, just after a small learning period, without the need to code a custom tracking algorithm. Many online free tutorials are also available, and following this very well made one (link) made the process of learning the basics much quicker, and provided many insights on what to implement next.

I jumped right in, and the first objective was to set up everything and to be inside of the virtual world. This was achieved rather quickly. Just a few scripts after, it was possible to wear the headset and see everything from a viewpoint inside of the scene, and walk/look around it. The way SteamVR works is by having a **[SteamVR]** object inside the scene, which handles a few things: smoothing of the tracking data from headset and controller, as well as a few game features, such as pausing the game when the user enters the system menu and syncing the framerate of the headset to that of the game. It also has a **[CameraRig]** object, which contains

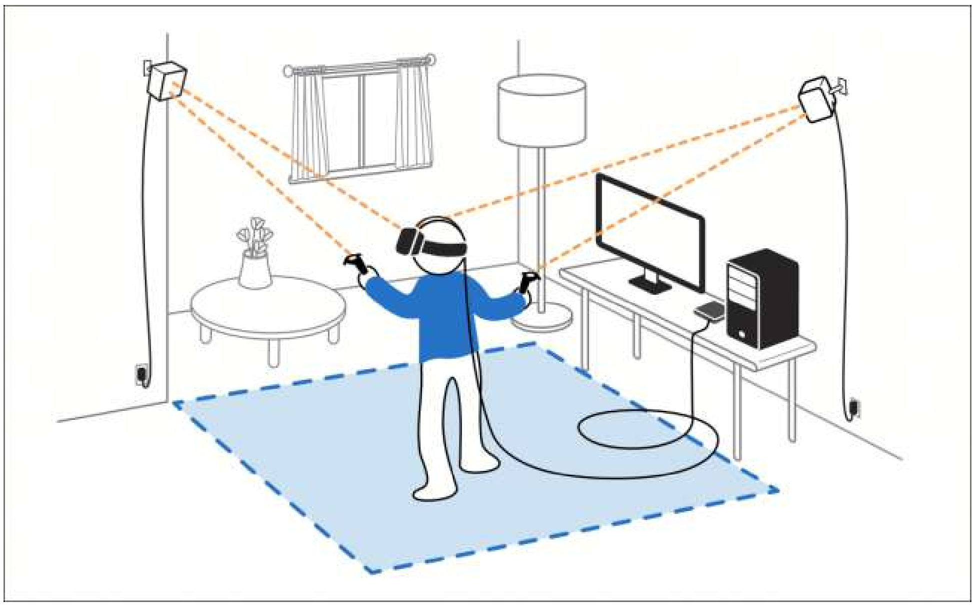
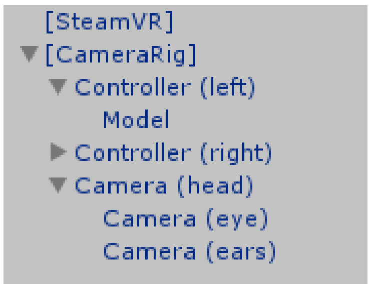


Figure 5.2: *This figure shows the schematic setup of the HTC Vive. See how the player wears the headset, which is connected through a cable to the computer, and also holds both the controllers to interact with the virtual world. The two base stations track both the headset and the controllers, and the player can safely move inside of the designated area. Virtual walls are displayed in-game for additional safety. Image taken from (link).*

virtual models for both controllers and the associated scripts, as well as a virtual camera to which the tracked position of the real world headset is applied, such that the in game camera follows the real world movement of the headset. A virtual listener component is also present, such that the audio that the player hears comes from the same position in-game. They are basically 3D transforms with scripts attached to them that deal with input and tracking, and that can be queried for location data and buttons state.



```
  [SteamVR]
▼ [CameraRig]
   ▼ Controller (left)
        Model
   ▶ Controller (right)
   ▼ Camera (head)
        Camera (eye)
        Camera (ears)
```

Figure 5.3: *SteamVR objects.*

The next step was to be able to interact with the scene using the controllers. One fun interaction was to be able to pick up objects using the controllers, and to throw them around, having them collide with surrounding objects. To implement this, I wrote a script that tracked the position of the controllers, and stored all collisions with objects in the scene marked as physically simulated. If a collision happens, and the user presses the trigger button, the colliding object will be picked up by the controller and held. The exact working of this system can be seen in the file *Grabber.cs*, which keeps track of colliding objects, as well as potential objects to grab and the current object that is being held. It also deals with having the object currently held follow the position of the controller. One problem that arose was jittering, meaning that due to small tracking errors the object would vibrate while being hold in the hand, and sometimes fall due to moving out of the "grabbing zone". This was fixed with some clever parenting of empty objects and smoothing. To be able to throw object, it was necessary to get the controller's linear and angular velocity at the time the trigger was released and apply these velocities to the object held up until that moment, such

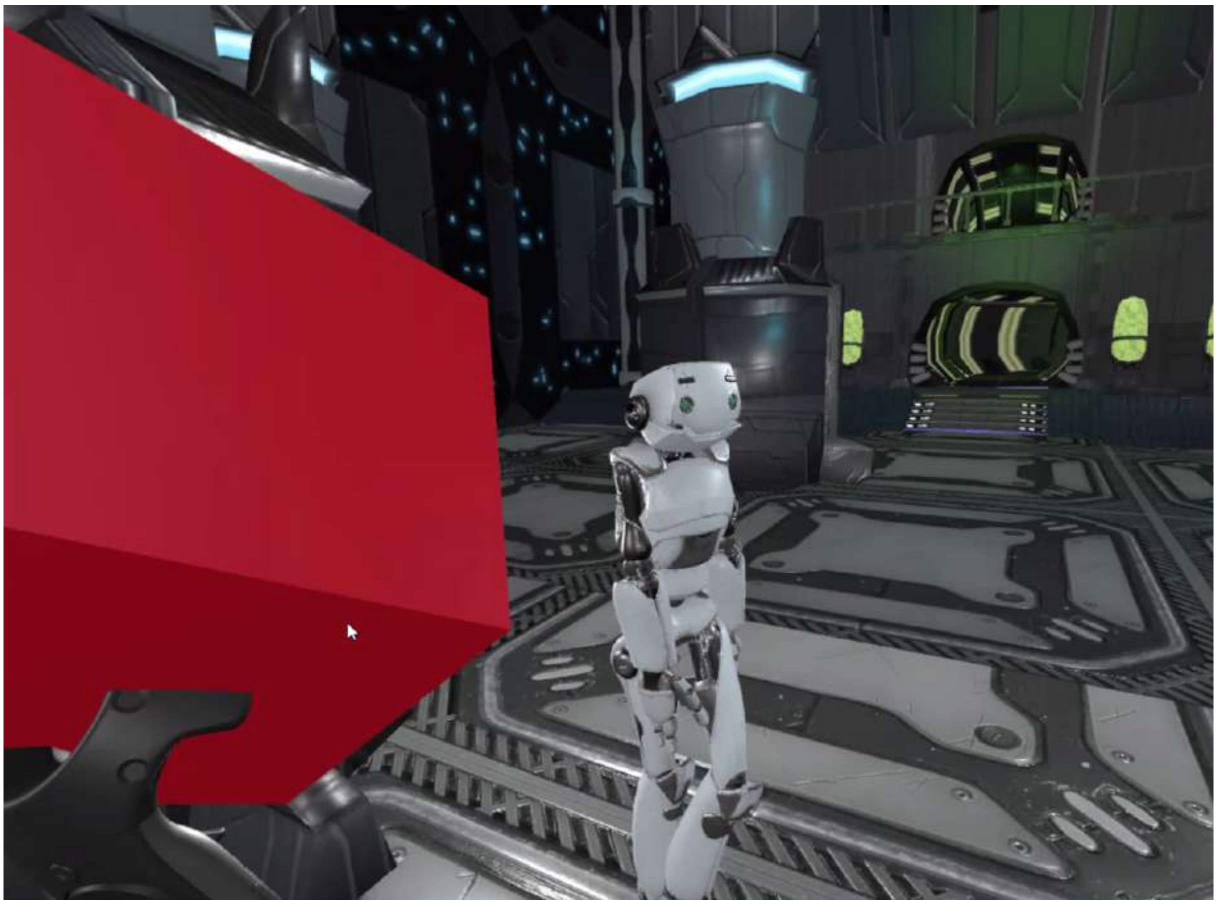


Figure 5.4: *The result of this first phase can be seen in this screenshot, in which the player is currently holding a cube with the vive controller and is preparing to throw it against the droid.*

that instead of stopping mid-air and falling down, it would continue its trajectory and keep its momentum, like you would expect.

Once done, what we had was a physical simulation in which it was possible to interact with the environment. Due to the general purpose of the physics system, it was possible to stack objects and throw balls at them, juggle small spheres, or try to hit moving objects.

Already such a simple application yielded many positive reactions, fun and a lot of laughter, although not much was implemented yet. However, the physics simulation still used Unity's integrated engine, and not the external library. I was about to start working on bridging between different physic engines and permit to setup rigid bodies in one and simulate them in another, while displaying them in the first. One of the most important tasks was about to begin!

5.4 Augmented Reality

Wikipedia defines Augmented Reality as a "live direct or indirect view of a physical, real-world environment whose elements are augmented by computer-generated sensory input such as sound, video, graphics or GPS data". In our case we use this term to describe applications in which a video feed from the real physical world, captured through a camera mounted on a device is augmented with 3D models of objects to provide the illusion of those object being part of the real world. This interaction paradigm is relatively new and allows for the creation of innovative and fun demos and games. In the following section we will see how to use a package that enables the inclusion of AR inside of Unity.

5.4.1 Vuforia

Vuforia is a computer vision package that enables augmented reality applications by means of tracking real world objects and markers. It is very well integrated with unity and allows the setup of custom markers to be recognized by a camera, such a webcamera or one mounted on a hand-held device such as a tablet or smartphone. The designer must scan those objects into a database, and the software will automatically detect tracking points to use in the video feed to infer the 3D position of the object. Thanks to this package, the recognition of the position, orientation and scale of objects in the real world is made easy. The objects recognized return a 3D transform that can be used inside Unity to position objects and overlay their meshes on the feed from the camera. The resulting display with the virtual objects on top of the real world can be used for a multitude of different applications. In the last section we will be looking at how this package was utilized to produce another demo, this time in AR.

6 Interaction Use Cases

We will see how the main game mechanics are implemented to interact with the character previously loaded. We start from game mechanics such as force generation from controllers and shooting, and explore much more complex paradigms such as character balancing and interaction with physical objects in AR.

6.1 Main Game Mechanics

It was now time to make the simulation interactive. So far what we achieved was to have the SIMBICON character fully simulated inside of Unity, together with all the physics being computed on the C++ side with a smooth integration inside the Unity interface, which allowed the droid to respond to all the objects in the environment.

6.1.1 Force Interaction

We now wanted to be able to simulate a force, like a Jedi would do in the well-known movies, to be able to push the droid around in a physically realistic way.

Since the implementation of hand gesture recognition using the HTC Vive headset camera could have been a thesis project of its own, we decided to use the buttons on the controller to generate the force. By squeezing the grip button (number 8 on the controller chart) the force would be generated.

Another possibility that could have been implemented, instead of using buttons, was to detect sudden changes in acceleration in the controller and then apply the force based on its direction and speed. I decided not to go with this solution as it would have likely ended in people punching things around them.

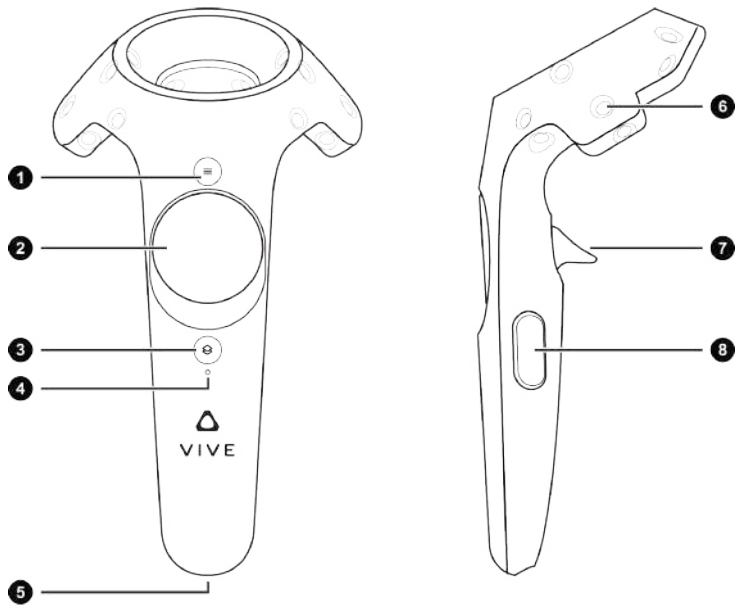


Figure 6.2: *All the available buttons and features on the Vive Controller:* ***1.*** *Menu button* ***2.*** *Trackpad* ***3.*** *System button* ***4.*** *Status light* ***5.*** *USB charging adapter* ***6.*** *Tracking sensors (IR)* ***7.*** *Trigger* ***8.*** *Grip button*

The force was modeled as an outward expanding cone from the controller tip. All the parameters of this cone, such as length, diameter, speed of expansion, force applied and change in time of said force are easily controllable from a simple component interface inside of Unity. Each time this cone would come in contact with an object designated as interactive (meaning it was marked as a layer which the force script had in its layermask list), an outward force would be applied to it. This force was based on the collision point, as well as the angle, the distance from the center of the cone and the contact time. This provided good results (plausible, since realism isn't really a possible metric here).

Figure 6.1: *Jedi kid.*

Once the force had been applied to an object, this was added to a "whitelist" of objects that wouldn't react to this particular force cone anymore. This was done to ensure that the same force wasn't applied multiple time to the same object, which would have otherwise resulted in explosions. Details of the implementation can be found inside the file *Force.cs*.

6.1.2 Shooting Mechanics

Another interaction type we (obviously) wanted to explore was to add a shooting mechanic inside of the game. We already had the gun assets, so we just needed to program it to shoot lasers. Usually what is done in videogames, is that instead of shooting an actual physically simulated bullet, a simple raycast is performed from the tip of the gun and it's checked against the collidables in the scene. This keeps the code simple and intuitive, but also has its drawbacks, such as bullets with infinite velocities and not taking gravity into account. In some more advanced

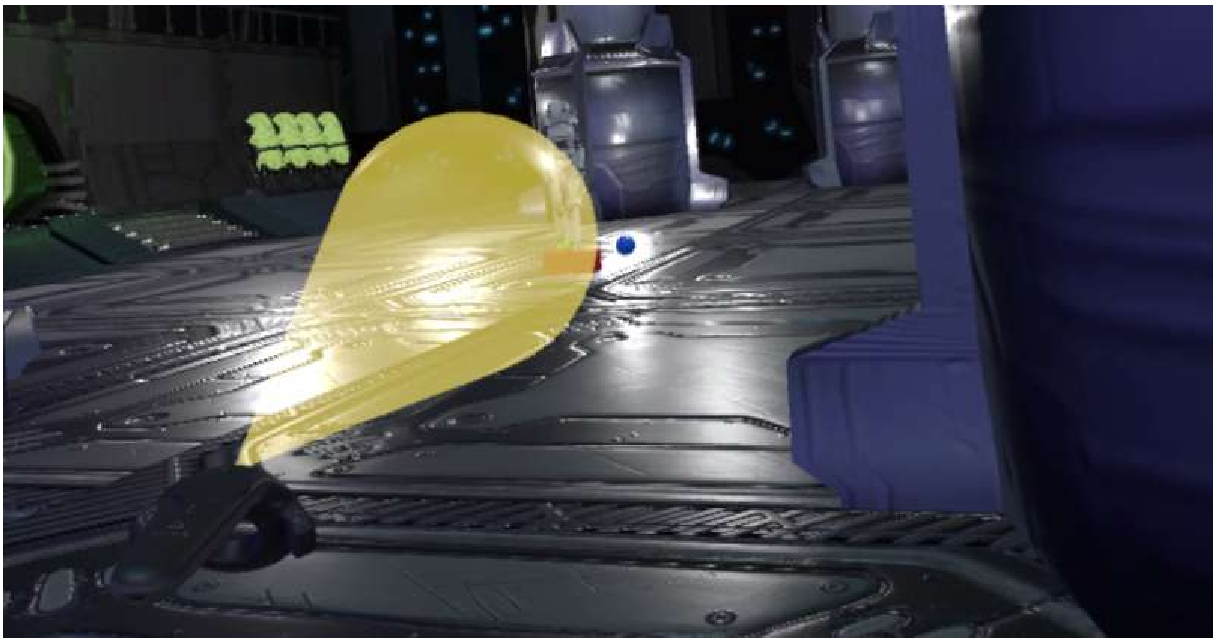


Figure 6.3: *The cone of generated force that applies an impulse to all objects with which it comes into contact. Note that usually the cone isn't visible. Here it is shown for illustration purposes.*

games, such as *Battlefield*, even the bullet is simulated. This is the approach that is also implemented in our game, the only reason being that this would allow the character to respond to hits without any extra programming, since bullets (lasers) would be simulated and have their own mass, speed and force. With this approach we expected to get realistic reactions from the model out of the box. All positive aspects of having an actual simulation!

Not much is to be said about this game mechanic, other than we wanted bullets to act like lasers in the well known movies, and therefore bounce off surfaces and not lose speed. The laser couldn't be set as a kinematic object (meaning it was fully controlled in script but would still have effect on other rigid bodies when a collision occurred), because it would not bounce off surfaces. So we used a little hack, manually setting its velocity each frame such that it would be constant, and setting its rotation to always point in the direction it was heading. The laser did bounce off surfaces since its restitution coefficient ϵ was set to 1, but without this tweak it would still lose velocity due to the other physical materials not being completely bouncy and still dissipating energy. It would also not be kept straight. To picture this, imagine throwing a stick around, and picture how it would spin after a bounce. To add some extra control to the shooting mechanic and be able to obtain different reactions from shooting at the droid, I implemented a gun slider that allowed the player to select how powerful the lasers that he shot should be. By increasing the power, one could obtain more drastic reactions from the droid, whereas with less powerful ones the droid would just react slightly to the perturbations.

6.1.3 Object Pool Design Pattern

However, each time the player shot a bullet, a considerable slowdown happened. This was due to how the new bullets were instantiated: firstly a new rigid body (.rbs) was computed and

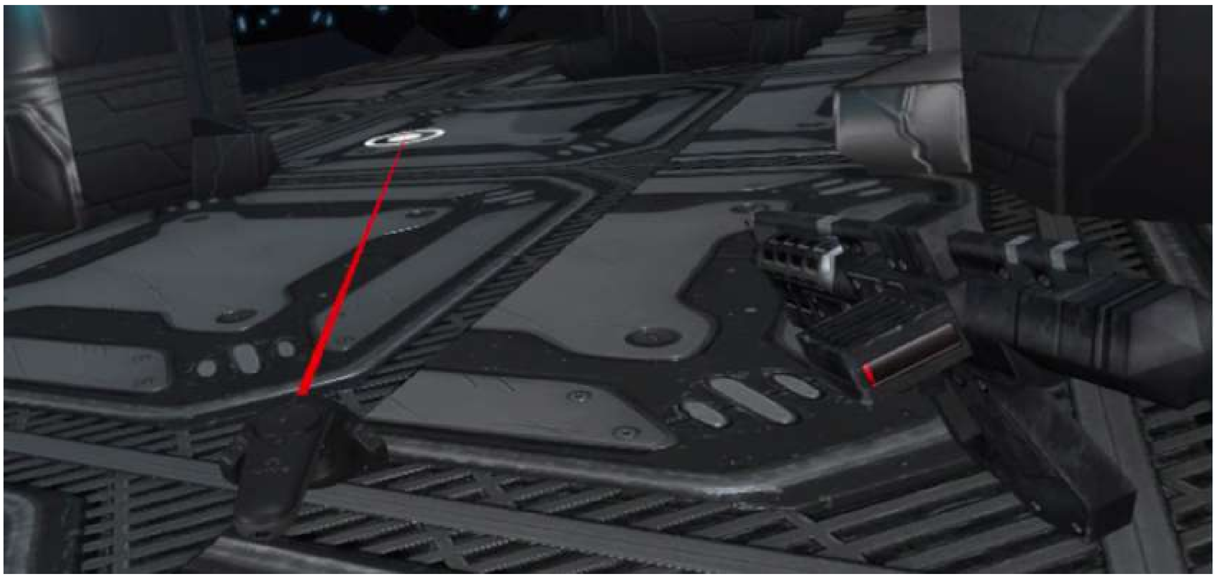


Figure 6.4: *How the teleporting mechanic looks like in the game. This screenshot is taken from the perspective of the player, and the gun with the aforementioned slider can also be seen.*

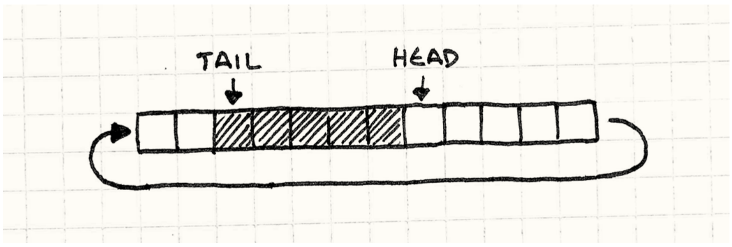


Figure 6.5: *Ring buffer layout in memory. Iimage taken from [Nystrom, 2014]*

written to file, and then a call from inside Unity would ask through the DLL to Aikido to load the file, which would in turn ask the underlying physics engine (ODE) to add it to its set of simulated objects. This was causing the lag. To reduce all of this overhead, a simple design pattern came very useful: the *Object Pool pattern*. Instead of allocating memory each time the player shoots and a bullet has to be instantiated, and then destroying this object after its lifetime of a few seconds has expired, at the beginning of the game, while all the assets are being loaded, the pool creates a fixed amount of bullets and deactivates them, by hiding the mesh renderer and disabling physics simulation for that object. This means that all the files are already created and that these objects already live inside the memory and all relevant data structures are present. When the player shoots, what needs to be done is to ask the pool for an inactive bullet, mark it as in use and we are ready to set its position, velocity, and other variables. When another bullet is needed, the same process is executed, and the pool automatically makes sure to return an unused object. When the object has to be destroyed, it is simply given back to the pool, which deactivates it for us and marks it as free. This eliminates all the overhead and makes the gameplay experience smooth like butter. The pool can simply use a boolean array to keep track of which objects are currently in use, but in our general case, since we know that the objects have a fixed lifetime, we know that they will be destroyed in the same order they were instantiated. This allows us to use a buffer ring, keeping track of the first free object and the first used one.

The only problem that can arise happens if all the objects are currently in use, since the pool doesn't have any free object to return. Many approaches are available, the one I went with was to take the oldest object and reuse it. This could cause weird behaviour, like having objects disappear from the scene, but since bullets move at a high velocity it is not very noticeable. Also, since the gun is tweaked with a multitude of parameters, such as muzzle speed, bullet force and time between shots, it is possible to tweak the latter parameter and the lifetime of the bullets such that the player will never run out of bullets.

If you are interested in reading more about this and other design patterns, [Nystrom, 2014] or [Vlissides, 1994] are both great resources.

6.2 Visual Randomness

In the demo, users were able to throw objects at the droid as well as shoot it. The animations in response to such hits and perturbations are created dynamically at runtime, and are therefore always a little different. However, since the velocity of the fired bullets is constant, if the player hits the droid in the same spot, the resulting simulated animation results *visually* similar. The same can be said by throwing cubes and spheres at the droid. Since similar objects have similar mass, by hitting the character in the same manner with the same object, the animation results similar. We decided to artificially tweak the forces to enhance the reactions and drama. Visually enhancing them would exaggerate the reaction and make the experience more fun.

We tried different approaches, such as adding more primitives to enhance the precision of the collision detection and have it match more closely the visual mesh. This didn't give us much visual benefit but slowed down the simulation considerably. We moved on to the next approach, adding random forces at contact events: this defied the purpose of having a realistic simulation. With these random forces the character would just react in a non-plausible manner to hits. We therefore went with an alternative choice, namely that of generating a random spherical cone each time a collision happened with the droid. A force vector was generated in this cone and the resulting difference with the original force applied to simulate perturbations in the collision. As I wanted to preserve the collision force magnitude, by just adding a perpendicular vector, the magnitude would increase. It would also not be uniformly sampled inside the cone. I therefore decided to go with a slightly more sophisticated solution: sampling a new vector inside a spherical cone.

To do that, I used the fact that in spheres, slices of same height have the same surface area. This is known as Archimede's Hat-Box Theorem.

If we sample $z \in [-1, 1]$ and $\phi \in [0, 2\pi[$ uniformly at random, we are sampling uniformly on the sphere surface (it would be like sampling on the enclosing cylinder and transferring the resulting coordinate to the sphere surface). To limit the allowed angle, just sample $z \in [\cos\theta, 1]$, where θ is half of the allowed cone angle. Therefore we get the vector

$$v = (\sqrt{1-z^2}\cos\phi, \sqrt{1-z^2}\sin\phi, z)$$

which is of unit length since $(1-z^2)(\cos^2\phi + \sin^2\phi) + z^2 = 1$. This random vector is oriented up, and has length 1. It must now be scaled and rotated to match the collision vector size and

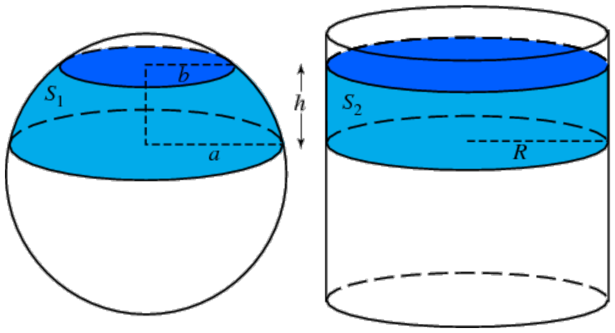


Figure 6.6: *Figure in which the corrispondence between the surface areas of slices of a sphere and of a cylinder are shown. Image taken from the http://mathworld.wolfram.com/ArchimedesHat-BoxTheorem.html*

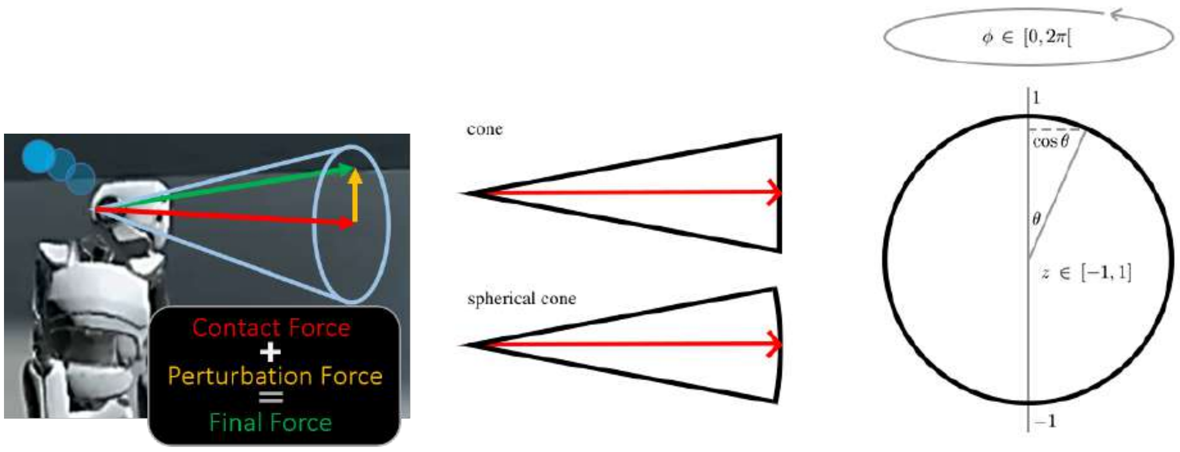


Figure 6.7: *Left: how the difference between the forces is applied to the point of contact to obtain the final result. Middle: the subtle difference between a normal and a spherical cone. Right: the computation of a random vector inside a spherical cone.*

orientation we want to perturbate. Scaling is trivial, but rotation is not. In my code, I use cross products and quaternion math to compute a suitable quaternion that, if multiplied with the above computed vector v, returns a new vector in the correct orientation. All the details of the above explaination can be seen in the source code of the file *RandomForce.cs* and *Util.cs*, the function *RandomVectorInsideCone()*.

6.3 Balancing Game Mechanics

At this point in development, we wanted to explore new ways of interacting with the SIMBICON. One idea came from seeing one old short movie by Wolfgang and Christoph Laurenstein, titled *Balance*. The characters were made out of play-doh and used stop-motion as animation technique. In this very enigmatic short movie, which can be seen on youtube (link), four mysterious characters are seen walking on a suspended plane, which inclines based on where they are standing. If they all go to one extremity, then the plane tilts to that direction and they all risk to fall down. We thought that our SIMBICON controller was suited to this kind of application, and wanted to develop an application in which the player could control the tilting of a virtual plane over which the character was standing. The character would then automatically try to keep balance thanks to the underlying algorithm.

6.3.1 Ball on Table

The first step was to have a simple simulation in which a ball would be balanced on a plane, which could be controlled either via keyboard and mouse, a virtual joystick, an accelerometer in a tablet or via the Vive controller. I implemented two versions, one using Unity's integrated physics and one calling the external DLL functions I previously programmed, therefore using the ODE physics engine. Using Unity's physics was straightforward, and everything worked straight away. When the plane was tilted, the ball responded rapidly and coherently to the motion. However, using the ODE physics engine, one problem presented itself: the collision detection wasn't working properly, and a sudden change in plane inclination caused the ball resting on the surface to penetrate the collision primitive and then pass through it, falling down. Only by tilting the plane extremely slow we were able to have a decent simulation. This would have caused a lot of troubles when we switched to SIMBICON, as it meant that the feet of the character would have become stuck inside the plane, which obviously was not desirable. The problem was caused by setting directly the position and orientation of the plane via script when the user pressed the corresponding button. I discovered that the ODE physics solver used the linear and angular velocity as well to make sure no object interpenetrates another. This is useful for fast moving objects. To understand why, imagine a game in which the player shoots a bullet in the direction of a thin wall. The bullet travels very fast, and the game runs at a fixed amount of frames per second (usually 60), which means that the bullet will cover a remarkable distance between two frames as he moves very quickly. What might happen with not such sophisticated collision detection algorithms is that in frame 1 the bullet still hasn't collided with the wall, and that in frame 2 it has completely surpassed the wall and finds itself on the other side.

Collision detection systems have to take into consideration such cases, and one common solu-

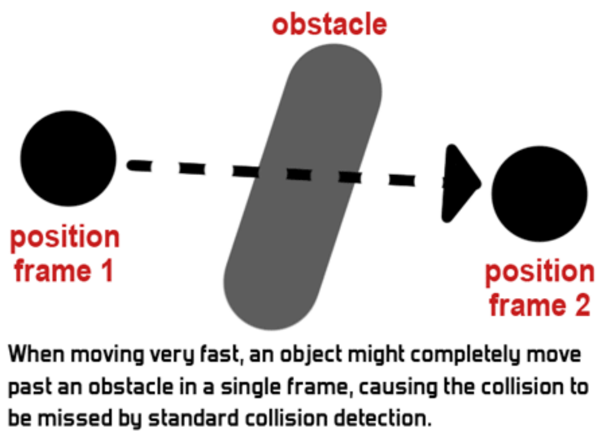

obstacle
position frame 1
position frame 2
When moving very fast, an object might completely move past an obstacle in a single frame, causing the collision to be missed by standard collision detection.

Figure 6.8: *A bullet moves very fast towards an obstacle. In frame 1 it is on the left, in frame 2 on the right, and collision detection fails. Image taken from Joost's dev blog (link).*

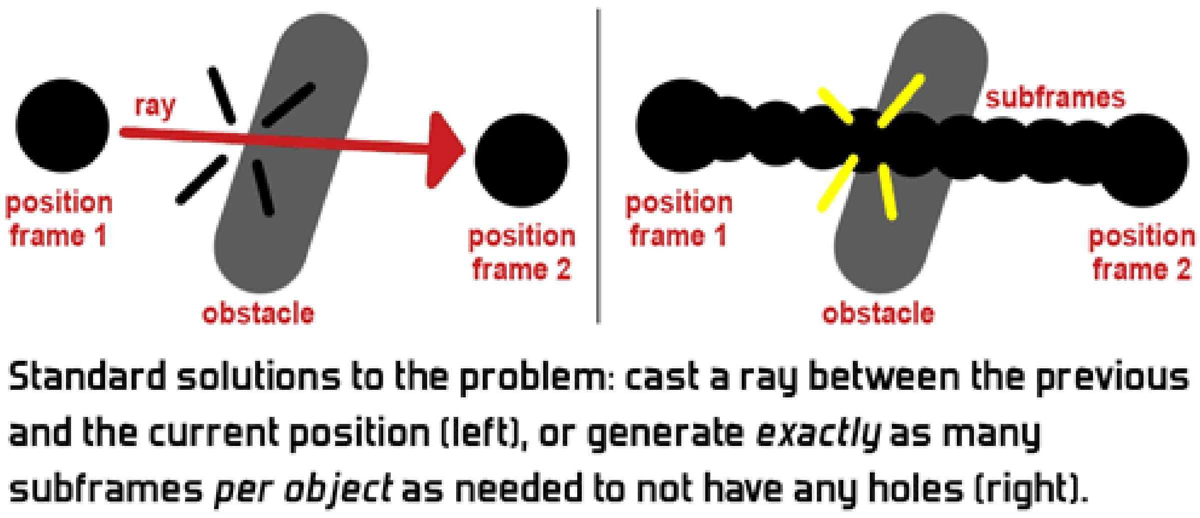

ray
position frame 1
obstacle
position frame 2
subframes
position frame 1
obstacle
position frame 2
Standard solutions to the problem: cast a ray between the previous and the current position (left), or generate exactly as many subframes per object as needed to not have any holes (right).

Figure 6.9: *Possible solution to detect collision with fast moving objects.*

tion is to "stretch" the collision primitive for objects moving very fast, such that collision will be detected. Another solution is to either increase the framerate, so that in a single frame the moving object covers less distance and is therefore less likely to completely pass another object. The framerate of the rendering can even be decoupled from the framerate of the simulation, and this allows to increase one without a big increase in computation costs. However, this approach might not always be feasible due to hardware limitations. Another simple solution is to shoot a ray from the moving object with the length of the traveled distance, and check if it collides with something.

However, no matter which solution is used, it is important for the system to have access to the velocity (and also angular velocity) of a rigid body. By simply setting the final desired position and orientation, the code cannot correctly compute the collisions. Unity, having access to both the previous position and orientation and the new ones, didn't need to have the velocities set accordingly, as by knowing the framerate (delta time elapsed between two consecutive frames) it could compute them on its own. This allowed Unity to adjust the collision detection primitives accordingly and avoid interpenetration. This was the reason it worked well regardless. ODE is a bit less sophisticated, so instead of simply setting the position and orientation I computed the necessary velocities to reach the desired position and orientation in the next frame. This solution fixed the interpenetration issue. This brings us to two different approaches which can be used when dealing with collisions: the force driven and the impulse driven approach.

- The **force driven** approach works by computing forces and torques to apply to the rigid body once interpenetration is detected. This solves it in the next frames. Is is a soft collision solution as a short amount of time is required for the forces to change velocities.
- The **impulse driven** approach works by applying an ideal instantaneous force, which results in a sudden change in velocities, as if an impulse had been applied: therefore, the velocities instead of the forces are tweaked. This is a hard collision solution, as instantaneous change in velocities is seen.

I decided that an impulse driven approach would have been easier, more responsive to the player's commands and therefore more playable, so I decided to implement it.

6.3.2 Robot on Table

Adding the SIMBICON to walk on the table was straightforward. One challenge that arose was that the controller of the SIMBICON computed the desired position of the feet to avoid the character from falling using the assumption that the underlying ground was flat and at height $y = 0$. Therefore, when the character tried to walk on the inclined plane, it would often stumble as the foot landed too early or too late with respect to what it was expecting, depending on the plane inclination. The solution to this was to tweak the controller, such that an additional parameter could be passed to the function for the computation of the desired step position. This additional parameter is an heuristic for the height difference of the step position given the current one. Since the final position of the foot is not decided yet, it's not possible to compute the height difference exactly. However it can be estimated based on where the reference motion would put it. This allowed us to pass the inclination of the plane from the Unity side and feed it to the controller, which will take into account this difference in height when computing the step position. This worked very well, as with this small tweak the robot was now able to walk along very steep surfaces without falling out of balance. It was great that the controller worked even though the passed parameter wasn't exact but only an estimation of the height difference: this emphasized that the controller was adaptive up to a certain point to an irregular terrain.

6.4 Tablet Simulator

One idea we had earlier on was to be able to run the simulation on a tablet, like the Windows Surface. We wanted to use its integrated sensors to detect the orientation and have the simulated character walk as if it were balancing on top of the actual tablet. Unfortunately it was not possible to access directly the device sensors within Unity's environment due to framework version limitations. Unity's standard API for accessing the accelerometer worked fine with Android and iOS devices, but didn't work with the Surface since it's a hybrid PC. A lot of time was spent trying to write external libraries that could access the underlying stream of data directly from the sensor and then feed it into the Unity environment. I was able to create applications that displayed the raw data as orientation of the device, and export it to a DLL. However, Unity's limitation allowing DLLs only up to framework version 3.5 and the requirement of a framework of 4 or higher for using the Windows API for accessing raw data, forced us to abandon this project.

It was possible to use Unity's integrated API to access the accelerometer data, but this worked only when building the project as Windows Store App, which required exporting the project and compiling it twice. This didn't allow for rapid prototyping inside of the editor and since we were experimenting and iterating very fast, this would have killed productivity. Also, the Aikido DLL didn't work inside the exported project for the same compatibility reason as before. Therefore, we decided to focus our efforts on the Vive platform, and to virtualize everything.

6.4.1 Tablet VR

The HTC Vive allowed us to follow the original idea of balancing the character on a tiltable platform. Only this time, instead of holding a real world object like the Surface, we could do the same inside of the virtual world. This was achieved by having one controller provide position and orientation information that could be applied to the virtual plane. However, using two controllers to tilt the virtual plane was preferred, holding one in each hand. One controller has 6 DoF, and therefore together they have 12. However any rigid body in 3D space can have rigid transformations applied to it (translation and rotation, but not scale), so it also has 6 degrees of freedom. To have them match, I could either add constraints in the real world to the controllers (attaching them to a rig) or create a script to compute an averaged 3D transform in the virtual world. I tried both approaches. The script, instead of taking the 3D transform, tracked single points attached to the controller and triangulated them to have a final transform for the plane. This worked quite well.

Rig A primitive rig made out of cardboard was also created to have an actual object to tilt in the real world. It was now possible to use a single controller to track the plane position.

6.4.2 Snake Game

At this point we could have a droid character or robot walk on a dynamically inclined platform, and it was possible to control it in virtual reality while being completely immersed in the environment. The only thing missing was some actual objective for the player to aim towards, as not much could be done in this sandbox other than experimenting around. As it will be discussed in the last chapter, many ideas of possible games were brainstormed, but this one seemed the most appealing: the idea was to tilt the plane while simultaneously balancing the character in a way to direct it to reach and collect small coins. The idea was similar to the popular old game Snake, in which you must direct a virtual snake to eat apples without biting its tail. The droid was set up to always go in the direction of the steepest point of the plane, just like a ball would. This

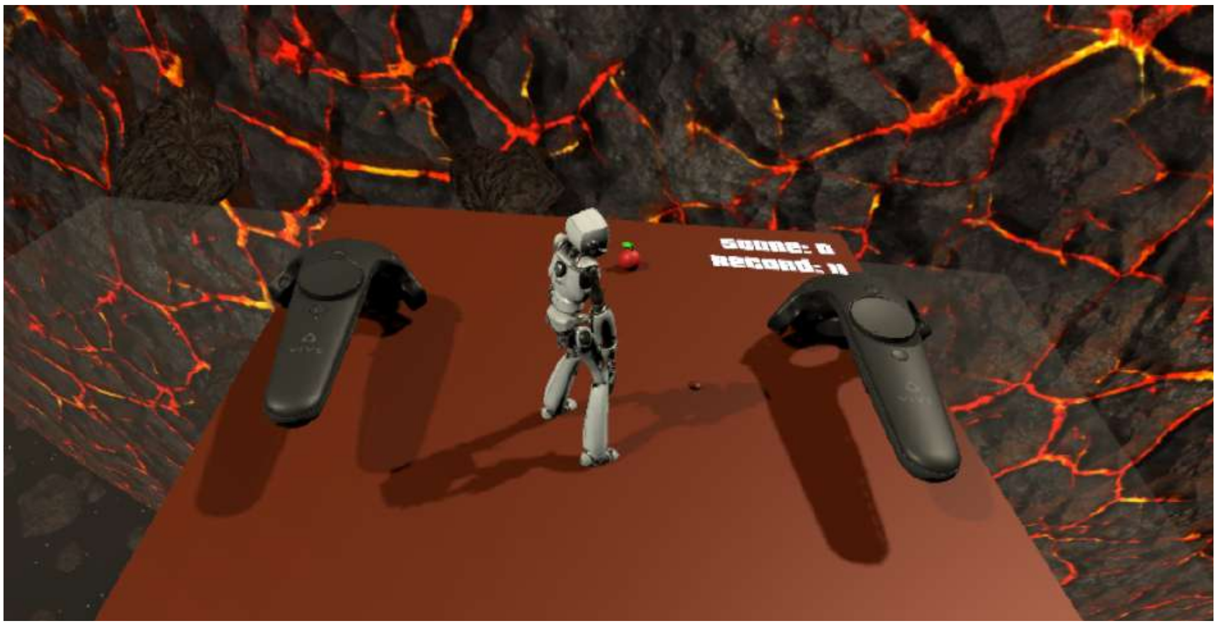


Figure 6.10: *A screenshot of the in game scene while playing. The player is controlling the position and orientation of the plane using the controllers, and the droid moves to the steepest point with the goal of reaching the red apple and score a point.*

way, the player could indirectly control how it moved by gently tilting the plane in the direction of the spawned collectable (small apples).

6.5 Augmented Reality Demo

One of the last interaction concepts we wanted to explore as this project was coming to an end, was to port our virtual droid to the Augmented Reality scene. We wanted to create a simple scene with a few obstacles, which then could in turn be automatically recognized and inserted in the virtual world. The droid would then use the underlying controller to respond to the unscripted perturbations. We decided that our scene had to include an inclined ramp, which could be positioned anywhere on the play-field and that the droid could walk upon. We also decided to include some rubble terrain, to show that the controller is robust even with non-flat terrain. Finally, we decided to include some cannons what would shoot projectiles at the droid. When hit, the droid reacted accordingly keeping balance like in the previous demos.

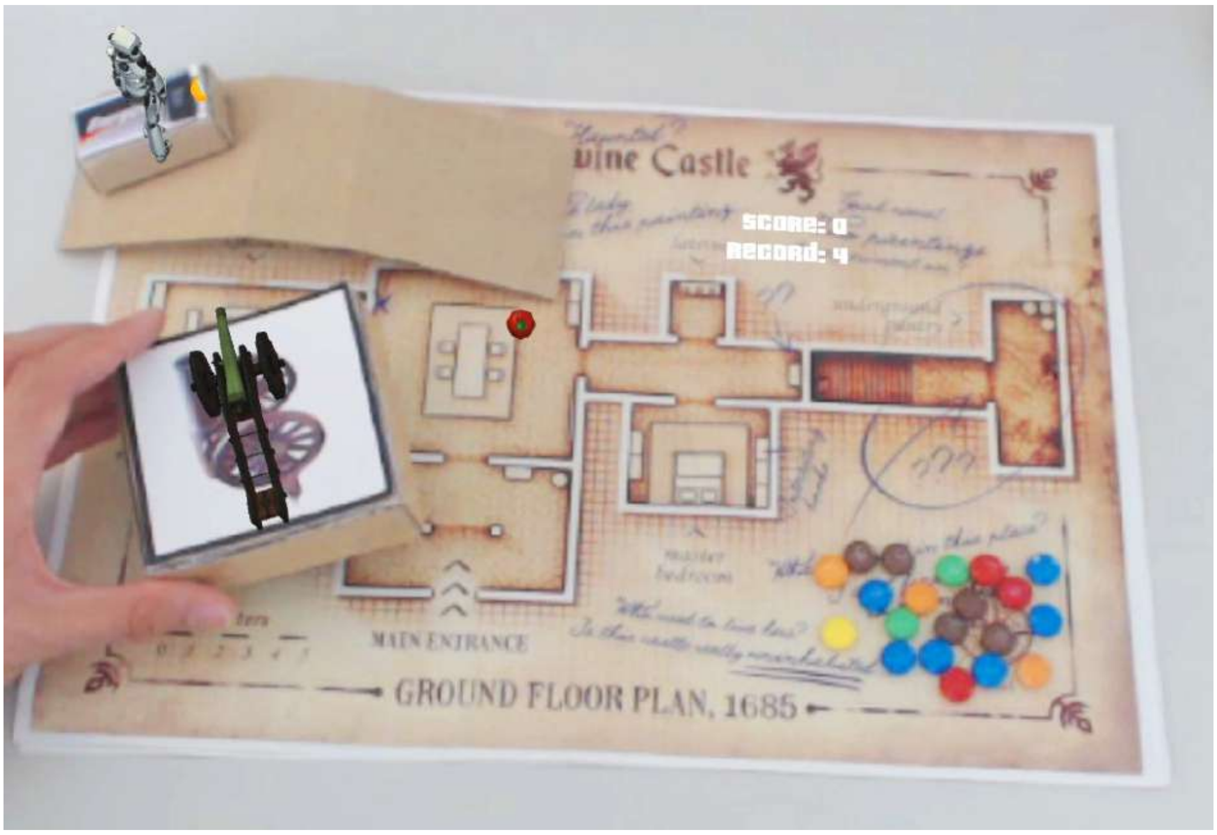


Figure 6.11: *A screenshot from the latest demo. The droid can be seen while it keeps balance from being hit with a banana shot from the cannon. The cannon is spawned on top of the cardboard marker and can be oriented to hit the character. It is also possible to see a cardboard ramp in the real world, which the droid can climb as the ramp is brought into the virtual world and made interactive. An uneven terrain with candies can also be seen. The droid can walk on this terrain and keep balance although the terrain isn't flat.*

7 Conclusion and Outlook

7.1 Game ideas

As this project was about exploring new ways on how to interaction with our simulated SIMBICON, during a few brainstorming sessions several ideas, more than the amount that was possible to implement, were conceived. Some of the most interesting we unfortunately didn't have time to implement are treated in this section, to give a broad overview on how a physical simulation and a controller for a character such as SIMBICON can be used for new interactive experiences.

Having a character that reacts so well to hits and punches, one idea that naturally comes to mind is to have a *boxing simulator*, in which the player immersed in the virtual world has to fight against the character. The implementation wouldn't have required much more than it was already in place, as in our original simulation it was possible to grab objects and hit the droid with them. The droid also assumes a fighting stance when it gets close to the player, so this idea could have been easily explored. The main parts required for this game would be the implementation of an AI, based on which the droid could exhibit a more realistic fighting behaviour. The AI could have been implemented with a simple FSM and several variables, such as "angriness" based on punch received and successful hits. Furthermore, with the advancement in physical rigs that allow players to interact with the virtual world, such as exoskeletons that provide haptic feedback, it is possible to make the player feel the blows and hits. This would definitely make for a compelling experience. This simulation could have been also adapted to meet another idea, that of having two characters fighting each other. It would have needed having two separated AI, with each AI's target set as the other AI, and keeping them isolated, each AI wouldn't differentiate whether it was up against a player or another AI.

Another idea was an endless runner, in which the droid had to be controlled with the goal to have it go as far as possible, and avoid falling. This could be further expanded into a game in which the droid has to get out of a labyrinth, and one player has to control it. To control the droid, multiple strategies could have been used, such as the one used in our demos or more traditional commands.

Cooperative or competitive games could also be developed for this type of controller. For example, in a competitive two-player game, one player could be trying to fulfill some objective by controlling the droid, whereas the other player would try to prevent him from doing so. This could be accomplished if the other player was given the ability to throw stuff at the droid, or cause any type of perturbation to it that could make it unstable and eventually fall. Cooperative games could also be developed for this controller, in which each player has a small task to perform with regard to the droid, such as one described earlier.

A further idea was to have the droid autonomously move on a balancing platform, not under the control of the player, and the goal for the player was to move in the real world to match a position diametrically opposite to that of the droid, to balance the plane. The droid would react dynamically to changes in the plane orientation when the player wasn't quick enough to move in the correct position, and the consequent balancing of the SIMBICON would further require the player to adjust his position quickly. However, we decided not to implement this simulation because we didn't have a rig that simulated the player to also be on a tilting plane, and this would have diminished the impact of the experience. But this concept just illustrates one of the many possibilities for the controlled droid.

Many other game ideas can be developed for this controller. This shows that the SIMBICON is inherently a widely versatile algorithm.

I hope that some of these ideas might be of inspiration to anybody interested in exploring further interaction possibilities to expand even more the range of applications to which simulated controllers can be applied.

7.2 Learnings and final words

I believe the main goal of a Bachelor thesis is not to create a product, but rather to gain the experience that comes from doing so. I must say that this thesis really allowed me to better understand all the systems I worked with and see how much of what I learned in all the past years came together to provide useful and applicable knowledge for this specific domain. Also, the fact of having produced different demos definitely adds value to this experience. I think I was lucky enough to experience many benefits during this period. The main one being having learned to work with other people as well as independently on a multitude of different programs that will be of practical utility in my future. I believe the experience in an environment such that found at Disney Research prepares well for the future, as it is not possible to gain all the necessary knowledge to be proficient in a given profession just in a theoretic environment such as the classroom. I believe that practical experience is fundamental in seeing how all the pieces of the puzzle fit together.During these months, I had the opportunity to see how a team in a studio works together, collaborates and uses specific tools to get the job done. A lot of experience was gained in practical coding skills and problem solving, which are maybe two of the most useful takeaways from a thesis. As I desire to work in a game development studio after finishing my studies, I appreciate all the insights and facets I learned during my stay at DRZ. This definitely made me more prepared for what awaits me. I therefore take the opportunity to thank once again everyone at DRZ that shared a bit of their knowledge with me. *Thank you sincerely.*

Bibliography

[Borer, 2016] Borer, D. (2016). Intuitive Design of Simulated Character Controllers. Technical report, ETH Zürich.

[Buckland, 2004] Buckland, M. (2004). *Programming Game AI by Example*. Jones and Bartlett Publishers, 1 edition.

[Coros et al., 2010] Coros, S., Beaudoin, P., and van de Panne, M. (2010). Generalized biped walking control. *ACM Trans. Graph.*, 29(4):130:1–130:9.

[Gregory, 2014] Gregory, J. (2014). *Game Engine Architecture*. CRC Press, 2 edition.

[Hodgins et al., 1995] Hodgins, J. K., Wooten, W. L., Brogan, D. C., and O'Brien, J. F. (1995). Animating human athletics. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 71–78, New York, NY, USA. ACM.

[Johansen, 2009] Johansen, R. S. (2009). Automated Semi Procedural Animation for Character Locomotion. Technical report, Aarhus University.

[Kovar et al., 2002] Kovar, L., Gleicher, M., and Pighin, F. (2002). Motion graphs. *ACM Trans. Graph.*, 21(3):473–482.

[Lucas Kovar, 2002] Lucas Kovar, Michael Gleicher, F. P. (2002). Motion Graphs. Technical report, Michael Gleicher.

[Nicolas Heess, 2017] Nicolas Heess, Dhruva TB, S. S. e. a. (2017). Emergence of Locomotion Behaviours in Rich Environments. Technical report, DeepMind.

[Nystrom, 2014] Nystrom, R. (2014). *Game Programming Patterns*. Genever Benning, 1 edition.

[Stelian Coros, 2010] Stelian Coros, Philippe Beaudoin, M. v. d. P. (2010). Generalized Biped

Walking Control. Technical report, University of British Columbia.

[Vlissides, 1994] Vlissides, E. G. . R. H. . R. J. . J. M. (1994). *Design patterns. Elements of reusable object-oriented software*. Addison-Wesley Professional, 1 edition.

[Yin et al., 2007] Yin, K., Loken, K., and van de Panne, M. (2007). Simbicon: Simple biped locomotion control. *ACM Trans. Graph.*, 26(3).

[Yongjoon Lee, 2010] Yongjoon Lee, Kevin Wampler, G. B. e. a. (2010). Motion Fields for Interactive Character Locomotion. Technical report, University of Washington.