Synthesising character animation for real time crowd simulation systems in Unreal Engine

Luis Delicado Alcántara

MASTER IN INNOVATION AND RESEARCH IN INFORMATICS
COMPUTER GRAPHICS AND VIRTUAL REALITY

in the
FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)
UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) – BarcelonaTech

Supervisor: Prof. Nuria Pelechano
Co-supervisor : Prof. Carlos Andújar
Department of Computer Science

June 2018

Acknowledgements

I would like to express my gratitude to my supervisor Nuria Pelechano for the useful feedback and engagement through the development and writing process of this master thesis. Furthermore , I would like to thank the participants in my survey, who have willingly shared their precious time during the process of interviewing. I would like to thank my family and friends, who have supported me and helped throughout the entire process.

Abstract

Achieving realistic virtual humans is crucial in virtual reality applications and video games. Populating large virtual environments with crowds of virtual humans is a very time consuming task, not only in terms of real time performance, but also in terms of creating realistic looking and animated characters.

When it comes to simulation, these systems need to compute navigation, collisions and AI, but also need to apply the right poses to each character for each frame, so that the overall simulation looks real. In the case of virtual reality applications and video games, this computations need to be performed in real time, which is not simply 30FPS anymore, as new hardware requires higher frame rates (60FPS for stereo, and over 120FPS for most Head Mounted Displays).

Nowadays there are several software and game development tools, that are of great help to generate and simulate characters. They offer easy to use GUIs to create characters by dragging and drooping features, or making small modifications. Similarly there are tools to create animation graphs and setting blending parameters among others. Unfortunately, even though these tools are easy to use, achieving natural animation transitions is not straight forward and thus non-expert users tend to end up with animations full of artifacts.

In this master thesis, we have developed a framework to automate the process of generating graphs for animation synthesis. Our system is developed with Unreal Engine 4 and the main contributions include: the generation of graphs for animation synthesis embedded with the UE4 tools, a tool to ease the adaptation and configuration of new animations, a method to add different behaviours to the characters, and finally a method to generate automatically graphs for animation synthesis from any given set of animations.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

In recent years we have seen how the game industry is creating games where not only the main character has realistic movements, but also all the avatars of the NPCs (Non-Player Characters) are improving in both appearance and behaviors. In the past, we could observe that games would only use a handful of characters which very often had no behaviors at all (e.g. they would simply stand in a place with at most a basic idle animation, or had a cyclic animation such as walking following a pre-defined trajectory with poor synchronism between trajectory and animation). Nowadays, hardware improvements in graphics cards and the availability of popular game engines, we can find games applications that show many NPCs that make the environment highly realistic and dynamic. Some examples of this can be found in the games of Assassin's Creed [1] by Ubisoft where they show large agglomerations of people in virtual cities.

However, crowd simulation is still a challenging problem in real time applications like games, training or other types of VR simulations. Achieving highly realistic crowds, requires efficient algorithms for navigation around the environment, avoidance of obstacles, and animation synthesis that is consistent with the trajectories. This last point is by far the hardest. Obtaining realistic animation that follow the other behavior decisions in a natural manner can be very computationally expensive, and thus new algorithms need to be researched. The main goal of those algorithms should be to provide a good trade off between performance and plausibility. Moreover, it is very important to be able to integrate such algorithms with existing game engines, because that would contribute to both the gaming and academic communities.

Although there exists an extensive literature about the synthesis of character animations, those methods cannot be easily applied to multi-character simulation. There are many techniques that are designed to create realistic animations for a single character but they are very costly and can not be extended to large numbers of agents in real time applications.

One could think that current game engines, already solve most of these problems, but this is not really the case, unless you are an expert game developer. Game engines such as Unreal or Unity, offer tools with nice GUI where the user can drag&drop animations, and create transitions between them. But it is completely up to the user to determine what variables and values lead the blending of animations. Very often, the results of this blending are quite chaotic and full of artefacts. This was the main motivation for my master thesis: to research the possibility of creating automatically such animation graphs to ease the work of non-expert game programmers.

The initial motivation for this project was to enhance the visualisation of a larger crowd simulation project. This project deals with procedural modelling of cities [2] and the goal of our project was to provide better animations for the avatars in the simulations to enhance the realism of those virtual cities. Since the original project is being developed in Unreal Engine 4, we will also develop our animation synthesis project for this platform. This was also an important personal motivation for me, as I was interested in learning this engine.

1.2 Objectives

The goal of this project consists in developing a framework for synthesising character animation for real time crowd simulation using Unreal Engine (UE4). Even though, the result of this thesis will be included in the aforementioned project, we also want to keep it as generic as possible, so that it could be easily ported to other game engines or frameworks.

The first objective of this project was to study and evaluate how UE4 handles character animation. Even though it provides a very powerful tool to achieve natural looking animation, there are many steps that require manual intervention. This has as a consequence that only expert users can obtain those natural animation. However any beginner, will end up with animations full of artifacts due to wrong blending parameters, of bad transitions between states. Once we had identified those difficult manual steps, our next goal was to research alternatives to provide automatic ways to solve those issues with as little user interaction as possible. The final result of this work is a semiautomatic character animation pipeline. The use can specify some high level decisions, then

the pipeline automatically sets the graphs and parameters for synthesising animations, and finally the user can do minor modifications if needed using UE4 interface.

1.3 Document walk-through

The next sections are structured as follows. In section 2.1 we introduce the state of the art and different contributions related to our project. In section 2.2 we provide some background terms that we need to know before continuing. In section 3 we explain the key idea of what we have developed and details on how it works. Section 4 contains all the different steps that we carried out during the development of this project, providing details on the different approaches that we explored, and indicating the limitations of those that we finally decided not to include in our method. In section 5 we discuss the results that our framework obtains and some comparison against other approaches. Finally, section 6 contains the conclusions of the project and the future work.

Chapter 2

Background

2.1 Related Work

This project aims at generating a method to synthesise character animations for real time crowd simulation to obtain more realistic behaviours. Before starting with our project it is important to review which approaches are the most used in the state of the art for this field. This section shows several frameworks, with some of them not only focusing in character animation synthesise, but also in developing the full pipeline for planning character behaviour. Moreover, we will not only review approaches for real time applications, but also non real time applications for crowd simulation techniques where it is possible to achieve more realistic animations without worrying about performance.

Most of the previous works employs some kind of animation graphs for character animations synthesis. Those animation graphs can be used for structuring the animations in a way that we can the automatically find the closest (or the x closest) animations fast. The work by Pelechano et al. classifies animations based on speed and angle between velocity vector and torso orientation. This graph representation is then used to determine the most similar animation to the crowd simulation input. Instead of generating the new animation by blending between several animation clips, it simply selects the closest node and applies it directly using time warping and adjusting the torso orientation by rotating directly the skeleton back joints [3]. One important detail that this paper uses, and that we will use for our work, is that they update the root position of the character without considering the feet position on the ground. This is not simple to compute, because in many cases it might lead to unnatural simulations.

There exists other research approaches in this field that also use animation

graphs, but with their main goal being to control the foot position [4] [5] [6]. These approaches avoid the problems that can have the previous approach. This problem manifest as foot-slice and is caused because the animation goes faster or slower than the motion. This is why the previous approach can induce unnatural simulations. It avoids this problem, but is harder in computer complexity because usually needs to use two graphs, one for each foot.The figure 2.1 shows an example of graph [6].

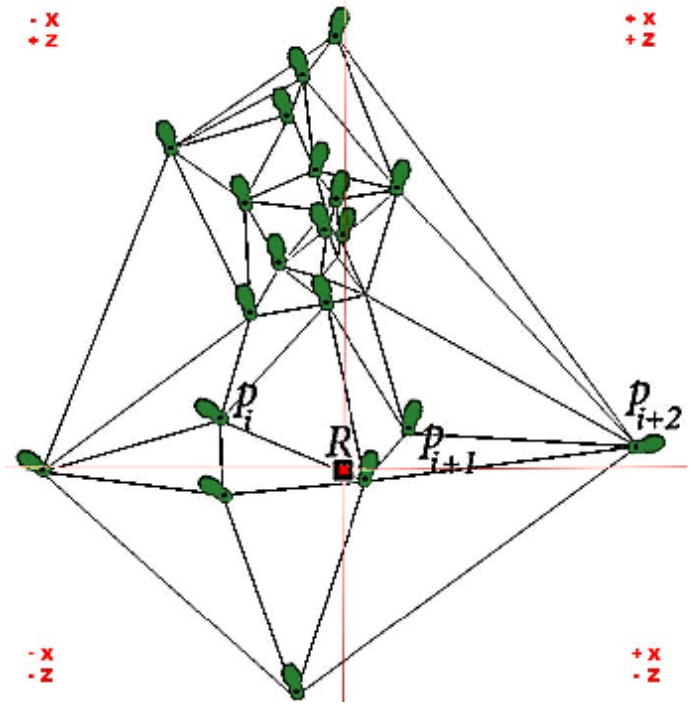


Figure 2.1: Example of graph to next step controlling foot position.

Other applications, focus on combining animation synthesis with collision avoidance [7]. They use the possible motions of the foot and detect which ones are able to continue the trajectory while avoiding the obstacles.

Recently, there have been some approaches based on learning which offer a completely new point of view. One of the latest methods uses a kind of neural network to find the correct pose for the next frame [8]. This method works with a very complex neural network that changes the weight of the nodes cyclically. Moreover, it uses a huge input for the neural network in which it needs the

future step positions, so a predictor is involved. This method provides good results and can be applied in a simulation of one or two characters in real time, but it can not support a crowd simulation.

Furthermore, there are more ways to generate animations. One of them that is very used in video games is using inverse kinematic, IK [9]. This method needs the position of some of the bones of the skeleton, usually the feet and the hands, and it calculates the rotations of the other joins following a set of rules(some bones do not allow some degrees). A recent example is presented by Ubisoft with the name of IK Rig that allows to use the same IK animations for any character or skeleton [10]. The figure 2.2 shows an example of motion with different terrains using IK [10]. The IK methods have a realistic results but they spend a lot of time.

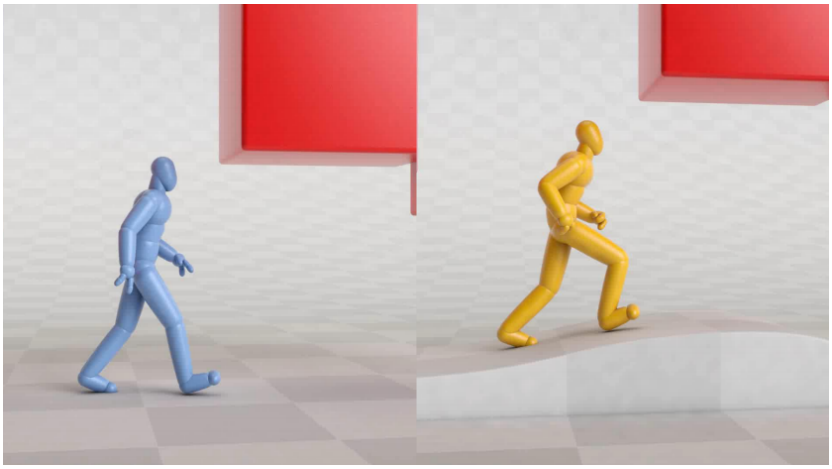


Figure 2.2: Example of IK motion in different terrains.

2.2 Preliminaries

This section explains some concepts that we need to know before starting the project and the explanation of some tools that we will use and the reasons why we decided to use them.

2.2.1 Character animation synthesis

This project is about generating automatically or easily a graph for synthesising character animations that provides a large variety of continuous motion from small set of animation. In the previous section we provided an overview of some papers that use this idea of synthesis the necessary pose of the character in run time [11]. In this section we will go into more detail to explain how works the character animation synthesis.

This process, as it is mentioned before, is about generate a variety of animations from a small set of animations. To do that, the methods use different kinds of interpolations between animations. The interpolation is applied to each bone and the result of the rotations can be calculate with different algorithms, like a linear interpolation between the animations involved or baricentric coordinates, like in some examples explained before [3]. Other important think is the method of classify animations. This is the part that selects which animations will be involved to synthesis the new animation. A usual method is generate a 2D parametric space where allocates the animations and then this space is divided using the animations as nodes of a graph. It usually is divided in triangles or quads, but it exists alternatives. An example of an animation classification framework is in the figure 2.3 [3].

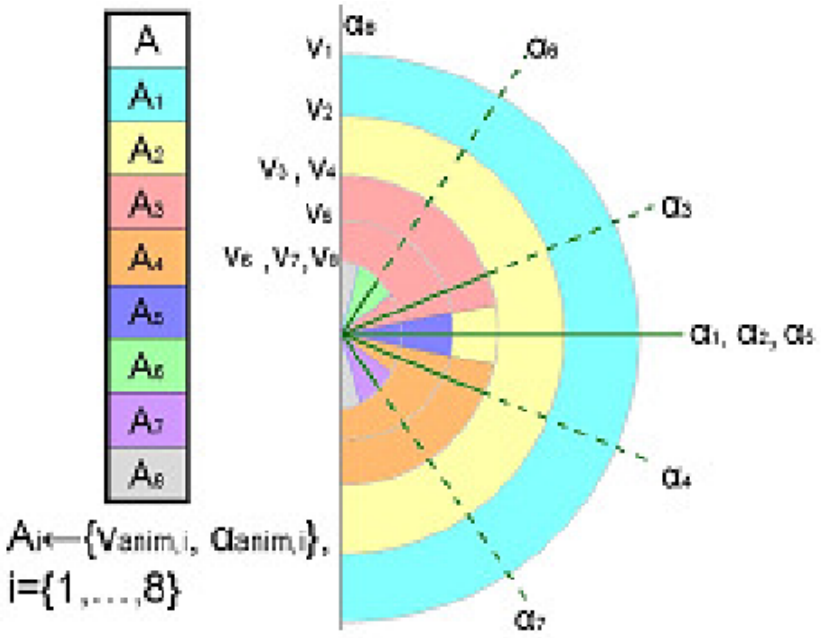


Figure 2.3: Circular Decision Framework.

One of the problems that this method needs to handle is the time warping cased by the blending animations with different time duration. Moreover, the animations need to be aligned to obtain a good result. To do that, the frameworks can use different strategies like use markers for alight, method that will be use in this project, or use more complex detectors.

2.2.2 Animation clip

The animation clip is a very generic concept, so its meaning can change depending on the context. For some people it can be any piece of animation and for other can be an animation with specific characteristics like time or behaviour.

The definition given in Unity is the most adequate one for our project: "Animation Clips are the smallest building blocks of animation in Unity. They represent an isolated piece of motion, such as RunLeft, Jump, or Crawl, and can be manipulated and combined in various ways to produce lively end results" [12].

In the case of this project, an animation clip is an isolated piece of motion than represents a step. A step is considered as an animation that starts with one foot on the floor and ends when it is again on the floor after lifting it. The

format used in the animations is FBX which are compatible with many game engines including Unreal Engine 4.

2.2.3 Game Engine

A game engine is a software development environment designed mainly to create videogames. The reason why there are so many different game engines in the industry, is because in the past every company would develop and use its own game engine. Sometimes they would even design a new game engine for a specific project. Nowadays, there are still a few private game engines, but some companies have recently made theirs free to use (with some commercial conditions), or even open source. The most popular game developing environments are currently Unreal Engine 4 [13], Unity3D [14] and CRYENGINE [15].

In this project we use Unreal Engine 4 to implement our ideas, but we could have used any other option, such as Unity3D or CRYENGINE. In this section, we describe how the different game engines work, what are the differences between them and why we use UE4 and not the others.

The first game engine that we considered was Unity3D developed by Unity Technologies. Until a few years ago, the versions went to Unity 5 with incompatibility between them, but in 2017 they launched started by Unity 2017.1 and kept updating version until arriving to the last 2018.1.2. It is a good engine to develop in 2D and in 3D but is the least powerful of the three game engines that we considered. The principal scripting language is C#, however, in previous versions you could use UnityScript which was basically their own interpreter of JavaScript, which is now deprecated [16]. One of the technical reasons why we decided not to choose Unity was that, although in recent months Unity Technologies has started to release their source code [17], it is not a open-source project since users are not allowed to change the code without paying a license. Users are only allowed to report bugs. This issue was a bit drawback for us, since we wanted to have the possibility of modifying components if needed. We could say that Unity is typically very popular among game developers, and researchers looking to easily create 3D content for VR applications, but when it comes to doing research in animation or simulation, sometimes Unity presents many limitations, as it makes it impossible to modify the core of the engine or to have access to certain information that is needed for the development of new algorithms.

The second engine that we considered was CRYENGINE developed by CRYTEK. This is maybe the least known of the three engines, although the games generated with it are very well known, like the saga Crysis or some of the Far Cry games. It has many versions with the latest being CRYENGINE V. Amazon bought a license to generate its own game engine based on it called Amazon

Lumberyard [18]. The principal scripting languages is C# and C++. A typical technique that more and more game engines are acquiring consists in having one interpreted language (in this case C#), for beginners or to develop small scripts, but using a compiled language (in this case C++) to create big games, since it provides higher performance. This engine is really quite similar to UE4 in performance and learning curve, and so either one could have been used for our project. However, if we had to give a reason to discard it, we can say that the community around it is much smaller than the UE4 community, which makes it harder to solve doubts and problems during development. Moreover, the first goal when CRYTEK created this engine was to have an engine to develop shooter games. This does not mean that you cannot make other kind of games or applications, but it increases the difficulty and makes this engine less versatile than others.

Finally, the third engine that we considered was Unreal Engine developed by Epic Games, Inc. As we have commented before, this was the engine chosen for the project and the main reason was the possibility of integrating it with another project that is currently being developed in UE4. It also has as many versions as the other two and the latest one is Unreal Engine 4.19, which is the one being used in the project. One of the differences between UE4 an the other two engines, is that not only it has incompatibilities between main versions like UE3 and UE4, but also the intermediate versions such as 4.18 and 4.19 can be very different in some parts and can be incompatible at least for parts of the project. The main languages are Blueprints and C++, which follow the same idea than before: using a interpreted and a compiled language. In this engine, they use their own language that are Blueprints Visual Scripting, which will be explained in the next section.

The main reasons therefore to work with Unreal were compatibility with the procedural crowd simulation project [2] [19], the possibility to develop code in C++, and the fact that it is open source.

2.2.4 Blueprints Visual Scripting

The Blueprints Visual Scripting used in UE4 is a scripting system based on connecting nodes to create gameplay elements. It is used to define object-oriented classes or objects in the engine [20]. There are many different kinds of blueprints.

The basic one is the Blueprint Class that allows to create functionalities on top of existing classes, but there are other types that are more specific, like the Level Blueprint that defines a global event graph for every level and each level of your project has one. One example of blueprint code is in the figure 2.4.

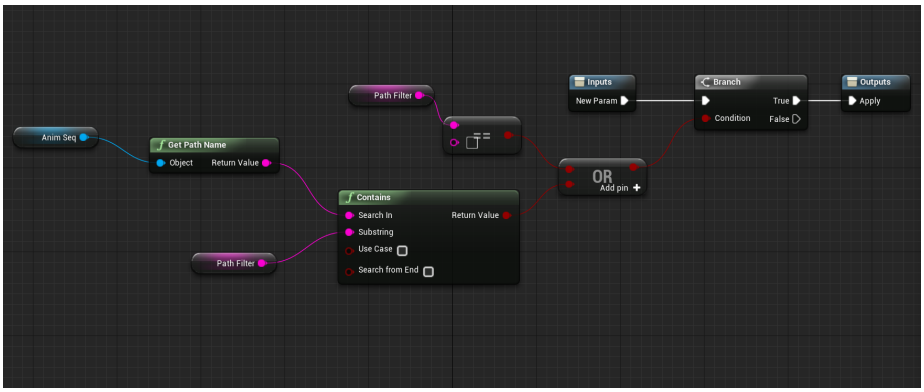


Figure 2.4: Example of blueprint code.

This language is really useful beginning people who have never programmed. To add some function or instruction it has a list with all the possible nodes that you can add separated classified by the field of the instructions. Moreover, to be faster, it has a search bar to specify the name function or part of it. Then, the method to connect nodes is so easy. Each node have the input and the output represented with circles at left and right respectively. The type of the variables are represented by colors, so connect nodes is as easy as connect colors between inputs and outputs. If we need to connect an input and an output with different types the interface generates a cast node, if is a simple variable, or apply the relevant function to connect these two types.

2.2.5 Mixamo

In the previous section we explained what an Animation clip is, but now will explain how to create them. There are different ways to create Animation clips for a character. One option is to use motion capture to obtain the animations, but it is not typically used for short movements such as single steps or jumps, but instead it is used when the clips are longer like a dance that is hard to recreate by hand. Other option is use programs like Maya Autodesk [21] or Blender [22], these are well known programs that allow the user to create an animation moving the skeleton of the character manually. These frameworks are usually not easy to use, so only experts can create natural looking animations by spending long periods of time manually polishing every aspect of the animation (posture, velocity and acceleration of joints, expressiveness, and so on).

Many animation artist and labs equipped with high quality motion capture systems, have created large databases of animation that are publicly available

on the internet. However, the main problem with those animations is that they are typically created for a specific character with its own skeleton, and this causes that often you can not use one animation from one character in another one (due to incompatibilities between skeletons). For that reason, we decided to use Mixamo [23]. Mixamo is an online service that offers a huge set of animation clips and a set for characters that are designed with the same skeleton so all of them can use all the animations provided. An example of these sets are shown in the figure 2.5. It is a good tool if you are not acquainted with the character design, and is a great help for obtaining big sets of animations. Moreover, this characters and animations are easy to import in Unreal Engine 4.

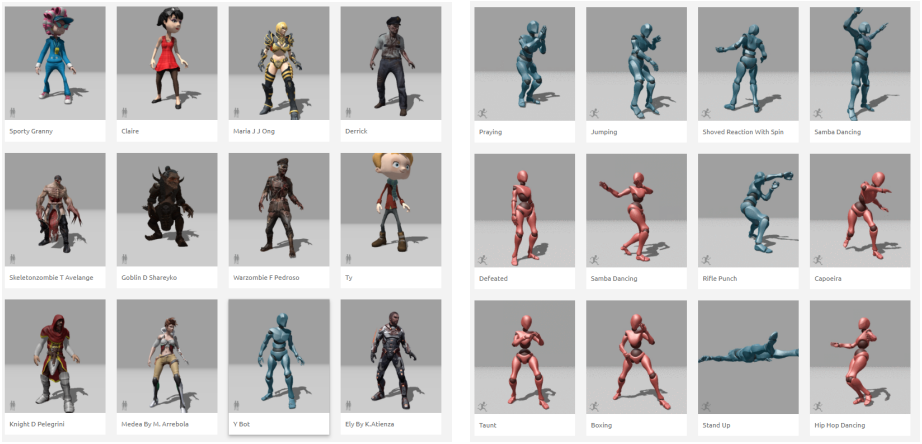


Figure 2.5: Example of Mixamo character and animation sets

Chapter 3

Methodology

In this section we explain how the development of this project is distributed in different parts. We describe the key ideas for the solutions proposed to each of the problems that were raised along the development of this project. We will also briefly described alternative solutions that were tested but latter on replaced by an alternative solution.

The development of this project followed an incremental approach. This means that we start with a simple solution, and then as the project advances, the framework gets extended to include additional parts. The global solution that we created is a framework that allows users to easily create animation synthesis graphs for character animations in Unreal Engine 4. Our goal is that the framework created will offer both good performance and realistic looking results for crowd simulation.

The first part of this project was to get familiar with the UE4 environment and to study the different kinds of tools that it provides to handle character animation. Then, using the powerful tools provided by UE4 we worked on generating manually an initial synthesis graph to determine the kind of animations that it could handle. This allowed us to study and evaluate its limitations and the complexity involved in setting a good graph with good transitions. From this part of the project we collected information regarding the parameters that control this graph, and how sensible the resulting animation was to any small inaccuracy in the manually created graph.

After this part of the project we had a good knowledge on the strengths and weaknesses of the given tool. The main thing that we noticed was how strongly the manual interventions could affect the output. So from this point on, we focused on identifying which parts of the tool could be made automatic or at

least semi-automatic (minimising user input to make the process of creating graphs easier and faster). In order to proceed, the next step to explore was the preparation of a small data base of animations clips. This task can be quite tedious because it needs to be done for each animation individually and doing it manually is not accurate. Therefore, to carry out this part, we searched for existing software and adapt it to our necessities.

The second part is the core of the project, and consisted on developing algorithms to automatise the generation of such graphs in UE4. To be more precise, the intention was to allocate the different animations in the space following some rules. To develop this part, we researched how we could programmatically allocate animations in the UE4 space, and then we designed the rules that will be used.

Finally, another part will was developed. The goal of this part was to improve the set of behaviours in the simulation. To do so, the idea that we present in this project, consisted on using more than just one graph and then develop an algorithm, to blend between them. We tried different possibilities to achieve a variety of behaviours. This part goes from allowing more kinds of motions for the characters to adding some kind of human behaviour to the character.

Chapter 4

Development

After explaining the background, the tools and the motivation of this project, this part will explain the full development of the project. This section explains the evolution of the project describing the different problems and solutions that we evaluated and tested. We also present the different improvements that we achieved, and discuss those alternatives that were tested but did not work as well as we expected.

4.1 Selection animation clip set

The first part is to study in depth how Unreal Engine works and what kind of tools it offers for solve our problem. Before starting to work with UE4, it is necessary to select the set of animation clips that will be used for the graph.

As is mentioned in the preliminaries section, the animations that we use, are from Mixamo and we select a small set of animations for the first approach that are contain between 10 and 12 clips with animations like idle, walk forward, run. This set includes animations with different speeds and degrees for turning left and right. As it is mentioned before, the huge set of animation clips and the number of different avatars that Mixamo offers, is a great help for people without knowledge in character design. However, we need to keep in mind some aspects when we use these animations in UE4. First, the skeleton that Mixamo uses for its avatars is different in terms of structure and nomenclature when compared to the default skeleton in UE4 so the animation clips from Mixamo cannot be used directly with avatars from UE4. Second, the Mixamo's animation clips have the root motion animation, that will allows us to move our character more

realistically and we will use and explain it later in the section 4.2.1. UE4 can use or lock such root animation, but the animations that it provides do not have the information of root motion. So, as the skelentons from UE4 and Mixamo are incompatible and Mixamos animations have the root motion information, which UE4 animations does not have, we will use only the animations an the skeleton from Mixamo's stack.

In the section 4.4 we extend the number of clips adding animations like side steps or rotations in place to solve problems that we encountered or to make the results more realistic.

4.2 Implementation using UE4 tools

Once we have the first animation set that we will use is time to research about what we need to develop our first automatic approach. What we want to do for it is to create a 2d parameterized space where every animation is a vertex and the space is triangulated. Then, at run time the simulation generates a parametric point in this space. With it, the triangle that encloses this point is selected. Finally, the pose of the character is obtained by blending between the animations of the selected triangle. Note that the manual step consists in placing the animation clips in the parametric space, and our goal is to do such allocation automatically.

The result of this process will generate the behaviour of the character and the displacement. To calculate them we need to see the different ways that allows us to move the character and how to generate the parameterized space in UE4 to blend the animation behaviour.

4.2.1 Displacement of the character

To displace the character there are two main strategies. First, you can apply the animation independently of the motion, which is a fast method and is provides an easy control over the direction and position at all times, because you set exactly the character in the position that you calculate. The problem with this method is that since you are controlling the behaviour and the displacement separately they are not synchronised and thus have many foot-slide problems, which creates the effect of very unrealistic simulations.

Second, you can use the root motion of the animations. If an animation has root motion it builds the motion of the root node of the skeleton around the space in addition to the rotations of the bones. This means that, for example,

if we use motion capture to obtain the animations it saves the displacement of the skeleton so it avoids the foot-slide. The problem is that it is harder to control because you are applying the animation and the root motion is fixed so the final character position is where the animation indicates. One example of a root motion from the documentation of UE4 is the figure 4.1 where the red line is the root motion of the animation [24]. This avoids foot sliding, as long as you are simply doing a play back of the original animation, but if any user control over the trajectory is needed, then problems arise.

Moreover, there exist approaches that offer a combinations of both. For example, use animations like walk and run with root motion but the rotation is controlled by the code [25]. This method was applied in the game Uncharted [26]. In the case of this project, we use the root motion of the animations, which as mentioned before, is provided by Mixamo.

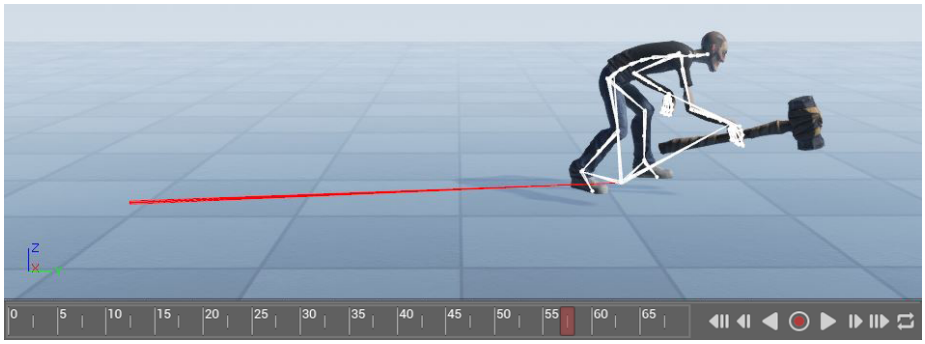


Figure 4.1: Example of root motion in Unreal Engine 4.

4.2.2 Blending of the animations

Once we know what animations we will use and what method will be used for the motion of the characters, is time to explore how we can create the 2D parameter space to blend the different animations. To blend animations, UE4 has two main forms that correspond to figure 4.5.

First, it offers some tools and function nodes in its blueprints visual scripting system or in C++. These functions do basically blending between two animations where the percentage used of the animation A is i and the percentage of the animation B is $1 - i$. It also provides functions that blend two or more animations but the total contribution does not need to add to 1. This method is not intuitive to generate the 2D space, because it needs to handle every clip

individually and the blends, which is not an easy calculation for this purpose, but could be useful in future improvements.

Second, UE4 has a component called Blend Space that is an element very similar to the principal idea that we want to implement. This system allows the user the possibility of generating a space in 1D or 2D depending in the number of parameters that it will use. This Blend Space is graphically represented as a grid and the idea is to assign a minimum and a maximum value for the parameters and subdivide the grid in each dimension. Then, the user can assign an animation clip to every intersection between each vertical and horizontal lines (in the case of 1D you have only horizontal lines and you put one animation in each line). After adding all the animations, it automatically triangulates the space (in the case of 1D this is not needed) and then it is applicable to a character or in a blueprint script if you want.

If we consider what we mentioned in the previous section about the root motion, both tools blend the behaviour and the root displacement of the character interpolating the positions of the animations that are being blended. For our first implementation we will use the second tool since it is very similar to our initial idea of the ideal approach, but in the section 4.4, which is about use more than one blend space, we will use both.

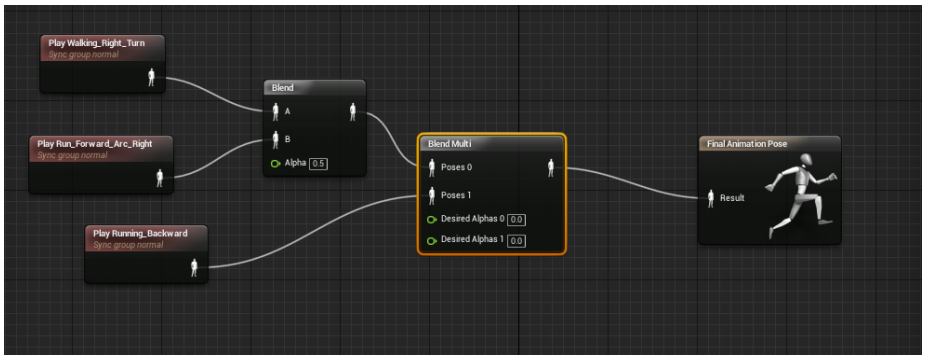


Figure 4.2: Example of blending blueprints code.

Figure 4.3: BlendSpace 1D interface in UE4.

Figure 4.4: BlendSpace 2D interface in UE4.

Figure 4.5: Different blend tools in Unreal Engine 4.

4.2.3 First implementation

After evaluating what we need for our first implementation, this section explains how it is done and what are the problems and improvements for this first design.

As it is explained before, it will use the root motion of the animations for the displacement and 2D Blend Space from UE4 to blend the animations. The first thing that is needed is activate the root motion in the settings panel for all the animations that the Blend Space will use. This seems obvious since there is a checkbox in the options of the animations to do so. But the UE4 interface is anti-intuitive because the animation viewer, when the root motion is disabled, shows the animation in a loop, starting in each loop step in the same position but moving in the space, but when the root motion is enabled it only moves the skeleton in place and does not move for the terrain, behaviour that is contrary to the expected.

After enabling the root motion, the next step is to create the Blend Space. To do so, it is necessary to understand the meaning of each parameter. As in the first case, it uses two parameters: one encodes the velocity, which only encodes the magnitude because it is a float and it can be negative to allow backward walking, and the second encodes the angle between the forward direction of the character and the velocity direction. The maximum and minimum values for the parameters are the following: the module of the velocity is an abstract parameter, which means that it is not a representation of an international system of units. In this first approach, we set the speed parameter with a maximum of 300 and a minimum of -300 for going backward. The angle between the velocity and the character position is determined based on the animations selected and it goes from -90 to 90, which means that the forward displacement has a 0 degrees.

Once all the elements are configured, the next step is to position the animations in the grid. This is done by hand and to make it easy, the grid is only divided by three intermediate lines since so it is easy to differentiate whether one animation is closer to 45 or to 90 degrees, but it is not easy if the difference between one angle and the following in the grid is too small. Once this is finished, you get a functional implementation, but given that it is done manually it is an error pruned process with room for improvements.

4.2.4 Problems and improvements

With the aforementioned implementation we have created a 2D parameterized space to blend animation clips, but also to compute the root displacement for the character interpolating between the root motions of the blended animations. If the aniamtions had been correctly created, this alone would avoids problems such as foot-sliding, and would achieve an overall natural looking animation. However, since many of the parts that are needed to generate the final animation, have been done by hand, the end result is likely to have artifacts. Moreover, there is another source of artifacts in blending, which is the case when one animation starts with the right foot on the floor and the other one with the left one, or vice versa. In the next sections we explain the approaches that we

proposed to solve such problems and the improvements that we achieved more thoroughly.

4.3 Foot Detection problem

One of the main problems when using blend spaces in UE4, which causes the most visible artefacts in the simulation, is the foot detection during the blending process. In this section we describe in detail the reasons for this error and we explain how to solve it in UE4. Moreover, in the second part we explain an improvement developed to eliminate this problem.

4.3.1 Blend animations with inverse foot cycle

This problem appears when it blends two animations that have inverted the foot cycles, which means that one makes the first step with the right foot and the second step with the left foot, while the others animation make the opposite order of steps. This happens very often, specially when the same animation is used for the characters turns applying a mirror function (option available with Mixamo). When Unreal Engine tries to blend them it aligns the first frames and the last frames without any offset or adjustment.

The usual result is an animation that looks as if the character would make small jumps all the time, or even worse, it appears to be floating as can be seen in the figure 4.6 where both feet are in the air but theoretically it is turning left.

Figure 4.6: Error of foot detection.

To solve this problem, it is necessary to find a way to align correctly the animations. To achieve this, UE4 has what they call 'sync markers'. This tool is very versatile since it can be used for many tasks. What it does is to annotate a frame with a label that it can be used to provoke some action or effect when the frame is played. For example, it can be used to activate a particle system when a foot touches the floor or to play some audio. For the case specific case of detecting a foot when UE4 blends two animations, that have some label in common, it uses those labels to align the animations if it is possible. What it does is try to align the frames that have markers with the same label name, that indicates that this frames needs to play at the same moment. In the next section we show an example where it fails to align the markers.

To add this markers, what the user needs to do is: in the animation view, go to the frame that will have the marker and put it manually. The markers added in the project are 'Left' and 'Right' when the animation touches the floor with the respective foot. Figure 4.7 shows an example of the marker bar for one animation. As in previous aspects of the animation synthesis process, this needs to be done by hand, and thus it is time consuming and prune to errors, specially for non-expert users.

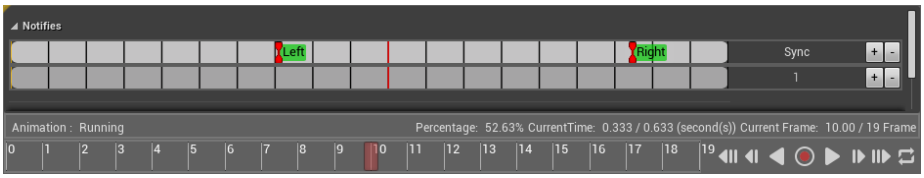


Figure 4.7: Manual Foot Detection.

4.3.2 Automatic Foot Detection

After studying how the sync markers work and how to are applied to avoid the problem of inverse foot cycles, we were not satisfied with this method and thus we studied possible ways to make this process fully automatic.

The result of the manual process after adjusting the markers many times can be really good. However, the main problem is that since it is done by hand it can be very imprecise and that a long amount of time to get it right. It is true that it is a pre-processing task and thus it only needs to be done once, but taking into account the time spent and the imprecise result, we decided to start investigating there exists some way to do it automatically or at least make it easier. UE4 does not have any tool that detect the steps of a animation automatically, but we found a blueprint script that even though is not fully automatic, it helps in the precision and the time consumed [27]. The final code of this approach, after the changes explained in this section, is in the appendix A.

What this script does is to generate a curve of movement for the bone selected and cuts it with an horizontal line such as $f(x) = a$, where a is a parameter that the user needs to select. To do that, first it calculates the relative position of the bone with respect to the root of the skeleton (since it can not take a world position of the bone). This position is calculated concatenating the transformations from the bone selected to the root. Then it selects one of the three components of the vector position that is selected in the parameters of the function. In this case it is using the Y axis and adds an offset. The bones selected are the named 'LeftFoot' and 'RightFoot' in the skeleton hierarchy corresponding with the ankles of the body. They are selected because it is the last big bone and the foot bones oscillate so much and generates blur in the curves. An example of the result curves is in the figure 4.8.

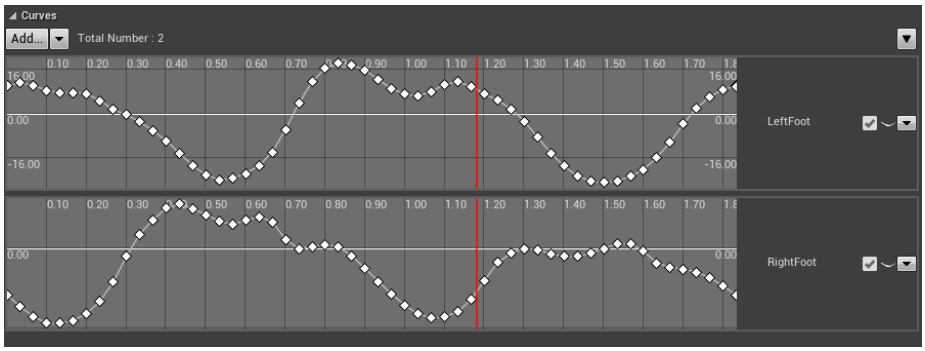


Figure 4.8: Bound position curves.

To detect the intersection points, in order to create the curve it applies an offset, it only needs to check when the curve has two frames where one has negative value and the other has positive. A marker is added for these points which can have two labels. If the first frame is negative and the second positive the label is the name of the bone concatenated with '_step_forward', if the frames values are inverted the label is the name of the bone concatenated with '_step_backward'.

With this information we can add sync markers that detect when the animation touches the floor and when it lifts the foot. The work that is needed to achieve this involves playing a little bit with the offset parameter in every animation for it to be more precise, and delete some markers that are generated as a consequence of the noise of the curves. This script makes the task easier although it is not an automatic method. Moreover, this solution sometimes generates animations that can not be aligned using the sync markers.

An example of this error is when it tries to blend the 'Running' and the 'Walking' animations. The result of the markers are shown in the figures 4.9 and 4.10. The problems come from the real behaviour of these steps. When people walk, the typical behaviour is having always one foot on the floor, while when people run, the typical behaviour is have sometimes both feet in the air. This is translated in the sync markers as in the 'Walking' animation, after one forward label goes the backward of the same bone, while in the 'Running' animation after one backward label goes the forward of the same bone.

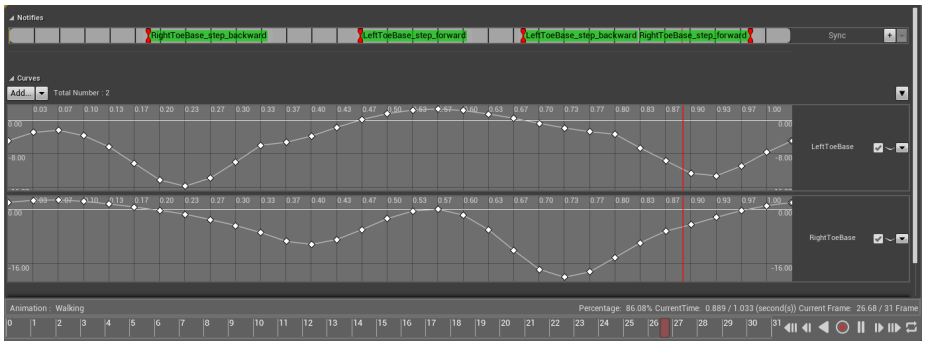


Figure 4.9: Walking animation sync markers.

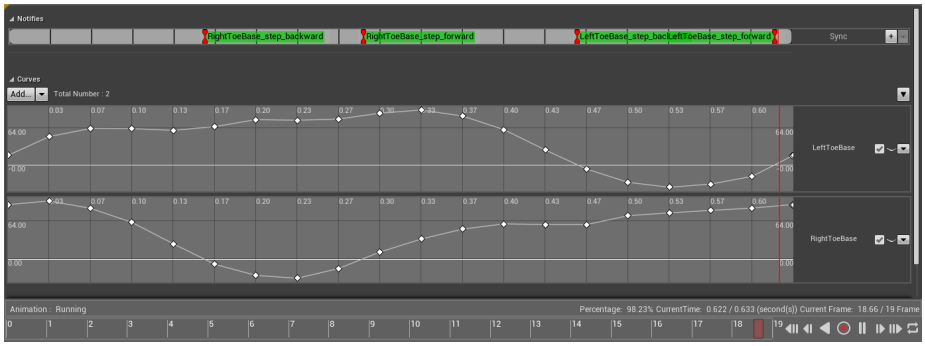


Figure 4.10: Running animation sync markers.

In order to solve this problem, following the idea of the foot detection, the solution consist in sorting the foot steps to know when it is a left step and when is a right step. The work that we did was change the original script to only detect when the feet touches the floor. The only required change is that now, we only need to detect when the curve goes from a positive to a negative value between frames. This is an easy change but it allows blend more animations than before. An example of a final result is shown in the figure 4.11. As is mentioned before, the final code is in the appendix A.

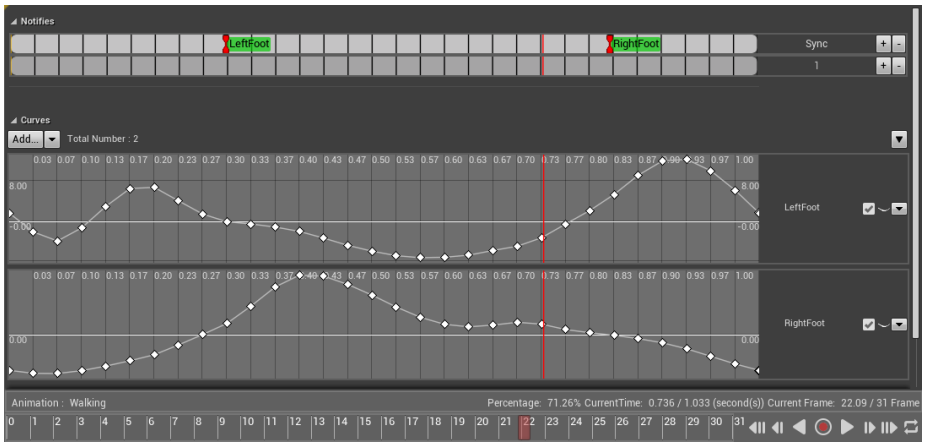


Figure 4.11: Walking animation sync markers final script.

4.4 Multi blend space

Having only a single 2D blend space, limits the graph of animations to depend exclusively on two parameters. With the current blend space, we can for example consider animations that walk or fun forward or with an angle, but we could not for example incorporate a step sideways animation. In order to have a richer range of animation it is thus necessary to increase the parametric space.

Once the 2D blend space was created, the next step was to research how we could increase the configuration of the graph, to allow for more behaviours, within the UE4 framework.

With the current 2D blend space, we could still add some more animations such as rotations in place when the speed parameter is 0 but the rotation parameter is higher or less than 0. The figure 4.12 shows these animations.

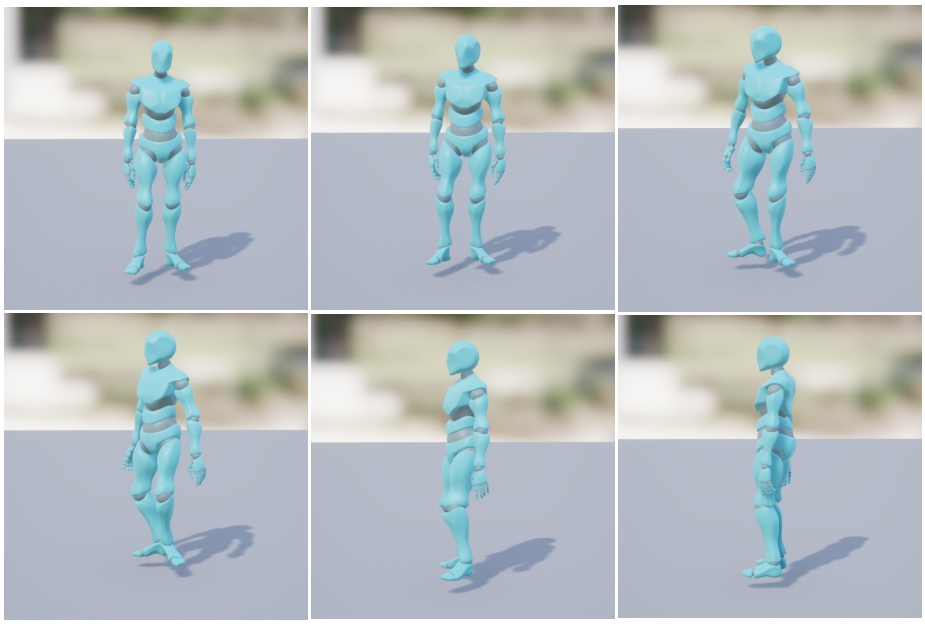


Figure 4.12: Rotation in place animation.

So as we can see, there is still some room to add animations with just one blend space, but in the end we find many limitations. Although the number of divisions in the grid can be increased, the number of animations that can be allocated in the blend space is limited by the two given parameters. This problem can be manifested in two ways:

- Some animations are incompatible with the behaviour of their neighbours if the movements are very different or simply the parameters chosen do not represent the behaviour of the animation.
- Some animations fall in the same place of the grid and that is not possible in a blend space of UE4.

In the next sections we explain further examples of this situations and how our proposed solution allows us to generate more behaviours and improve our simulations.

4.4.1 Ambiguity of animation clips

During the process of adding more animations to our system, we discover a problem with some common animations. This problem appears with more than

one animation but the first behaviour where we detected problems was with lateral walking. This was the first problematic animation that we found and the origin of the problem is that, with the chosen parameters, this animation can not be represented or, if we are more flexible, it leads to an ambiguity problem.

The problem is the current blend space assumes that the animation has always the root rotation looking in the same direction as the velocity vector. This translates in our 2D space as having the speed depending on the velocity of the character in the animation but the rotation is always zero because it is looking in the same direction. This means that the parameters from lateral steps and walking forward are the same. If we interpret the rotation as degree between root rotation and displacement vector it happens again. In that case it lead to problems with the turning animations.

In order to solve the problem, we started to research how other approaches solve this ambiguity. In this search we observed that some of the current approaches use two states to differentiate the situations. One example is in the paper *Phase-functioned neural networks for character control* [8], which as is mentioned before they use neural networks in their approach, but to use lateral steps a flag is needed to be activated, and for normal walking again the flag needs to be disabled.

Inspired by this idea we decided to generate a second blend space focused on lateral steps and the rotation parameter is represented as the angle between the displacement and the orientation. Then, we use the boolean blend function that UE4 has and we can control when to use one or the other. This function works very similar than the expleinde in previous sections. It allows to blend two animations, but instead of have a float input that determines the weight of each animation, it use a boolean to this, so the possibles inputs are 0 and 1 and it behaves as a switch that allows use one animation or the other.

This solves the problems observed, but we continued exploring alternative solutions. We then noticed that if we use a normal blend function instead of a boolean blend function we can have a third parameter ranging from 0 to 1. By adding this feature we can handle more kinds of movements, as opposed to switching switching only between two states. This provided the results that we expected, the motion was fluid without being a very different behaviour. As can be seen in the figure 4.13, we blend a lateral step with a turn step and the solution is a turn step but with the upper-body rotated a little bit to the front.

Figure 4.13: Result of blending two 2D blend space.

The main difference between this approach and a real 3D blend space is that we first blend six animations into two and then we blend again this two, so we are adding more weight to some animations that in a 3D blend space maybe they would not have.

Even so, with this approach we open the door to explore a method to increase the realism of the simulation and add more behaviours and movements to our framework. Some of these are explained in the following sections.

4.4.2 Character behaviours with 1D Blend Space

With the idea of blending between more than one blend space, we started to think what else could be done. One idea was to create more blend spaces depending on the behaviour needed for our character. This was particularly interesting to achieve animations with rare or unusual behaviours, for which it is difficult to find many examples on the internet, and creating them from scratch can be particularly difficult.

For example, if we try to generate a character with a drunk behaviour, maybe we find a walking drunk animation, a running drunk animation, an idle drunk animation and one turn left and right animation, but the number of animations will be small so the resultant blend space would have a bad behaviour. Another problem is that maybe we need to use multiple sources to obtain the full set. This is not tipycally a problem with a natural behaviour because the motions would be similar, but in the case of drunk behaviour the animations from two different designers can be very different and thus the result of blending between them can be very unrealistic.

Since the problem of obtaining a specific set of animation clips that satisfy our necessities is not easy, unless we specifically model it (which is hard for non-experts), we started to think what could be done with an small animation set for a specific behaviour.

As is commented, UE4 has two blend space, the 1D blend space and the 2D blend space. Until now the project is using only the 2D because the framework proposed for the initial solution needed two parameters. The idea that we propose is use a 1D blend space to add a behaviour to the character. As the 1D is only controlled by one parameter it has less kinds of motion and thus it needs less animations to achieve a good result.

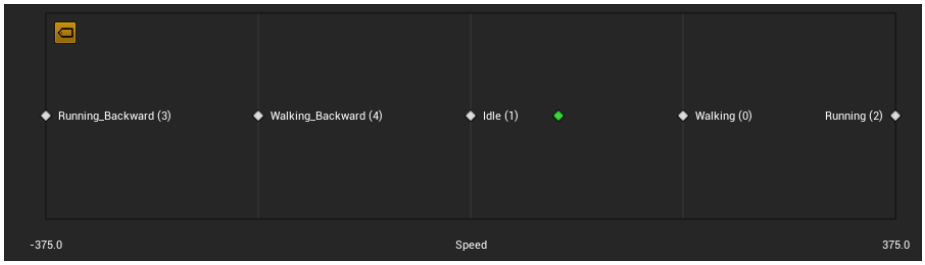


Figure 4.14: Example of 1D blend space.

For this kind of blend space we only use the speed as control parameter and the idea is to generate a 1D space with animations that goes forward and backward in a straight line. The set of this animations is smaller and easy to find because the biggest problem with the 2D space is to find rotation animations with different angles and velocities. To try it we created two spaces, one with a drunk behaviour that is shown in figure 4.15 and other with an injured behaviour that is shown in figure 4.16.

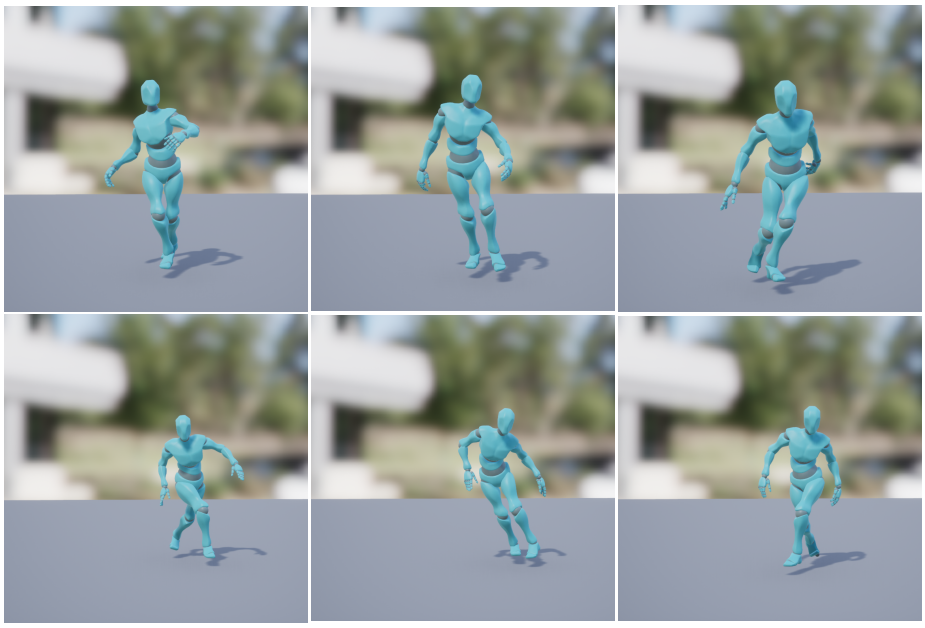


Figure 4.15: Frames from drunk blend space.

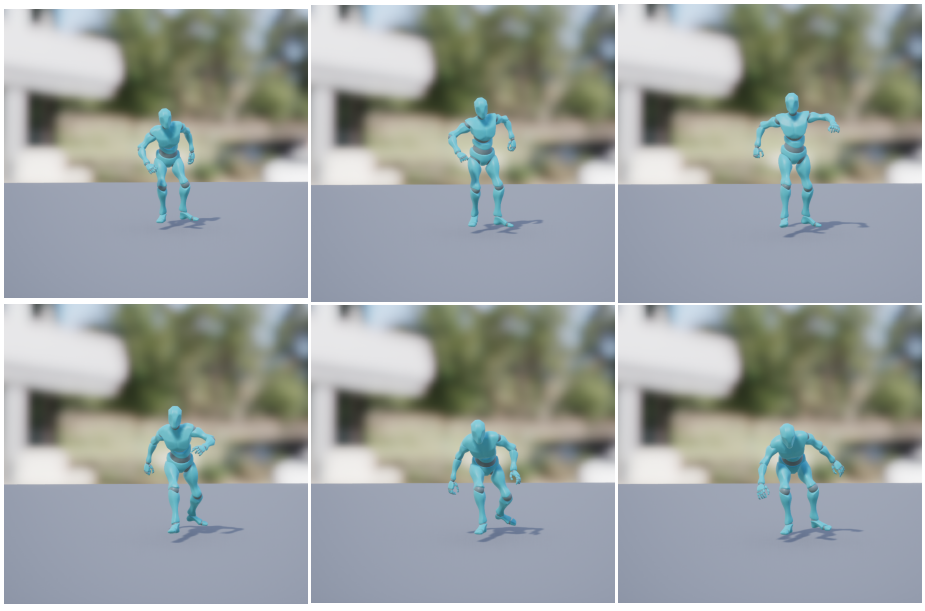


Figure 4.16: Frames from injured blend space.

Now we have motions with very different behaviours, but where the character only goes forward and backward and can not turn left and right. To expand the possibilities of our framework and solve this issue, the key idea was to blend the basic 2D blend space with the rare behaviour that we wanted to achieve from the 1D space. The idea of this is that the control of the motion is achieved with the 2D blend space and the rare behaviour is achieved with the 1D blend space. With this combination of blend spaces, we could obtain the behaviours that appear in figures 4.17 and 4.18.

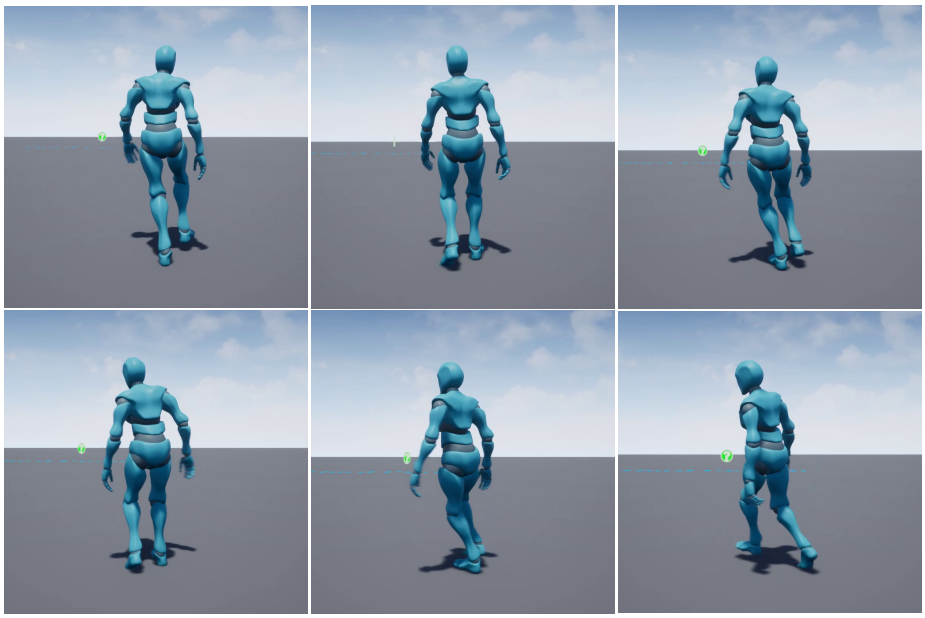


Figure 4.17: Full drunk motion.

Figure 4.18: Full injured motion.

With this blending between 2D space (locomotion directions) and 1D space (styles) we obtain the full motion of the different conducts and we can obtain different states if we change the blend parameter between the two blend spaces, for instance if you use 0.3 it will go less injured than if you use 0.5 because the base blend space take more weight. On the other hand, the control is less accurate since, for instance, if the rotation is a 45 degrees you are applying only the 70% in the final animation an the other 30% is a forward animation.

It is not easy to find out how the game industry makes their animations, or how they solve this kind of problems nowadays. However, some previous papers explain that an option is to use a simple animation or two animations that represent the drunk walking and combine it with inverse kinematic to generate the left and right turning animations from those walking animations [9]. That approach has similarities with our idea but the use of IK makes the computational time larger than in our technique. Furthermore, the game Uncharted, Naughty Dog Inc. uses only a forward and backward animations and, to turn left and right, it rotates the character manually [26], similarly to the technique explained in section 4.2.1. In that case, it uses some exceptions to make it more realistic but the problem of foot-slide appears, which can be avoided with our technique.

4.5 Automatic blend space

Up to this point, our work has focused on developing tools to ease the generation of blend spaces while increasing realism and variety of animations. So far, our work helps in the creation of the synthesis graph, but it still requires certain user intervention.

In this section we explain the method that we developed to automatically generate a blend space and how we still allow the user to tune it depending on our parameters. Moreover, it explains some possible problems that may happen, and other functionalities that it may have.

The main idea is to allocate automatically the animations in the blend space grid. Before doing so, it is necessary to execute the foot detection process explained in previous sections, on the set of animations that will be used in the graph. Once the set is processed, we need to find the correct parameters for the current blend space. In that case we start using the speed and the rotation of the animations as parameters of the blend space. Instead of allocating with symbolic values as before, which we assign the values only seen the animations (if one animation move faster it had more speed value), in this part we need to calculate the speed and the rotation of the animation.

To calculate the values of the parameters from an existing animation, we simply play such animation for one second (if the animation lasts less than one second it is run in a loop) and obtain the initial and the final transformation. UE4 has a specific function to obtain such information, but other frameworks also offers methods to obtain this data. From the initial and final transformation we can calculate the speed, since we know the passed between those two frames is one second and we can calculate how much the character moves subtracting the initial position to the final position and calculating the module of the result. Symmetrically, we can calculate the rotation using the initial rotation and the final rotation but in that case we use the z parameter of the result in Euler form that is the axis that traverse the character.

Once we have calculated the values of the parameters we can allocate the animation in the grid correctly. Initially the implementation of the UE4 for the blend space uses a grid, which means that we have a discretize space and thus we can allocate the animation clip in the exact position that we wish. To achieve higher resolution, we increase the divisions of the grid to $100x100$. With this dimension we decrease the positioning errors. After allocating all the animations, those that are not exactly in a valid point are displaced automatically to the nearest valid point. If an animation is allocated outside the grid, the blend space discards it. For that reason, it is a good idea to start with big maximum and minimum parameters that can be adjusted later.

After applying our method to the same set of animations that is used in the base 2D blend space the result is the following.

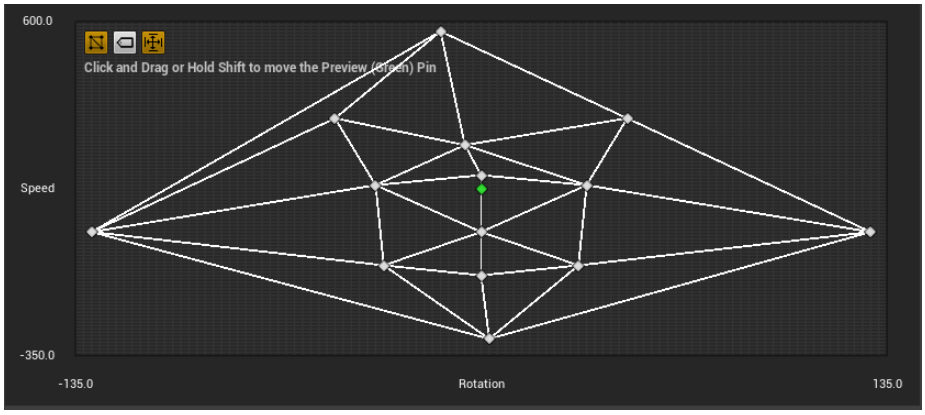


Figure 4.19: Apply automatic blend space.

The animations are located in the grid but some of them are not in the expected place. After analysing this result we discover some interesting information that can be observed using our method (which corresponds to the real behaviour of the animations), which are difficult to detect simply by observation. In figure 4.19 we can observe that the run forward animation being used is not moving they way we expected since it describe a curve with few degrees. Moreover, the rotations in place rotate so much in one second that are allocated really far away.

The problem of having outlier animations like the rotations in place is that the triangulation that UE4 does, joins vertices in the wrong way. In this example, UE4 joins the rotation in place with the run animation, which does not make much sense since they are very different animations and thus blending between them leads to awkward new animations. This triangulation can be observed in figure 4.19. One option that we tried consisted in playing the animations during only half a second instead of one second. By doing this we get the parameters from a half step. This option is done to observe if this rotation values is induced by some specific frames. It can happen if in one frame the rotation value is bigger than in the following frames, which emphasizes the value of the parameter just in that point, but maybe the value does not describe the trend of the animation. Reducing the time by half, the rotation value is reduced more than a half, but continues generating some wrong triangles. Note that this generates a similar space structure but it reduces the scale of the rotation axis so the maximum and minimum parameter will be changed. Figure 4.20 shows

the blend space with the same axis scale as in the previous case.

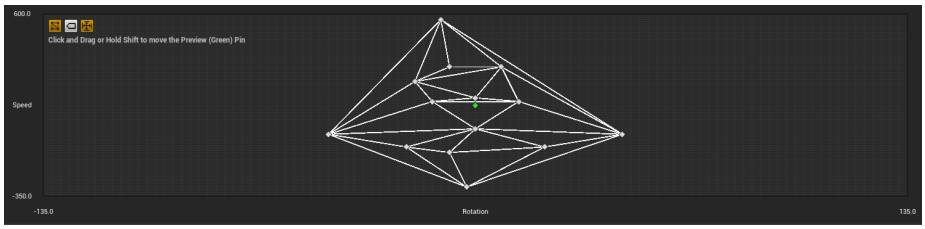


Figure 4.20: Apply automatic blend space with half second.

Another information that we can extract from our automatic allocations of animation clips, is what kind of motions are missing in our set. As it is explained in previous sections, it is not easy to find or to create a huge set of animations. With the information obtained, we can find specifically which animation clips we need to complete our blend space. We can also discard animations that, in the end, have similar behaviours. Moreover, this may help with the previous problem (large triangles between very distant animation clips). One of the solutions is to obtain the animations for some strategic points that leads to the generation of a better triangulation. To prove it, we use placeholders with the first automatic blend space to identify which animations are needed to obtain the behaviour that we want. Figure 4.21 shows the result with the placeholder.

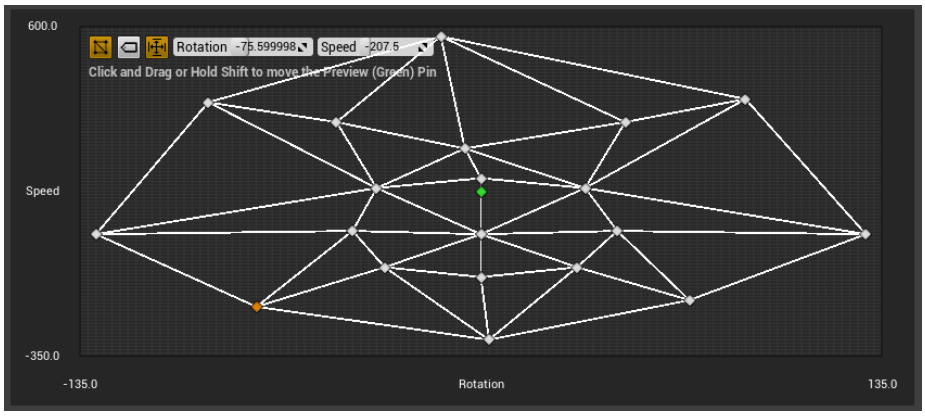


Figure 4.21: Apply automatic blend space with half second.

Taking this information into account, we find that we need to add six new

points to achieve a behaviour that is more similar to our idea. Since we can use one animation for every two points (because the points needed are symmetric), we only need three animations. Those animations are a turn forward left and right more or less running describing a close curve. Then we also need something similar, a turn left and right more faster than what we have that describes a close curve, but this time going backward. An the last animation needed is a turn left and right in place, but the turn needs to be slower that our actual animation.

One of the benefits of our method is that is easy to include in other framework and thus it is not mandatory to use UE4. With this idea we have the option to create our own generic blend space framework with the possibilities of using a 3D space with octahedron instead of triangles, or use other blend method similar to baricentric coordinates. The only requirement to use it would be to have or implement a blend space, and to be able to extract the root motion from animations, which is a typical functionality in other popular game engines like Unity.

On the other hand, as the speed and the rotation is obtained from the animation clip, the maximum and the minimum of these parameters will change between blend spaces that use different animations. This makes it difficult to blend between them and originates compatibility issues between different blend spaces. The problem is that the maximum velocity for a injured behaviour will be much more slower that the normal behaviour, so when the base motion is walking, the injured motion could end up using the fastest animation. To solve it, we can use more animations or scale the blend space that is similar to what we did by hand in the previous sections.

Chapter 5

Analysis of results

In this project we had two main goals: (1) The main one was generate a synthesizer for character animation to generate realistic simulations. And (2) the second was to use it for a real-time crowd simulation which adds more complexity as it requires computation to be simpler so that they can be computed in real time.

For these reasons we will measure two different aspects from the result. First analyse the realism of the character behaviour, and second, measure the performance of the framework in a crowd simulation.

5.1 Realistic result animation

One of the important goals of this project was to generate realistic simulations. As it is difficult to quantitatively measure the realism of one character, what we did is to run a comparison against other approaches. We will thus describe what kind of problems typically appear in other approaches that we managed to solve. We will also so the limitations of our method by comparing how other approaches solve some problems that we did not eliminate successfully.

In the figure 5.1 we can see the images of different approaches that are mentioned in this thesis and can be found in the bibliography of this project.

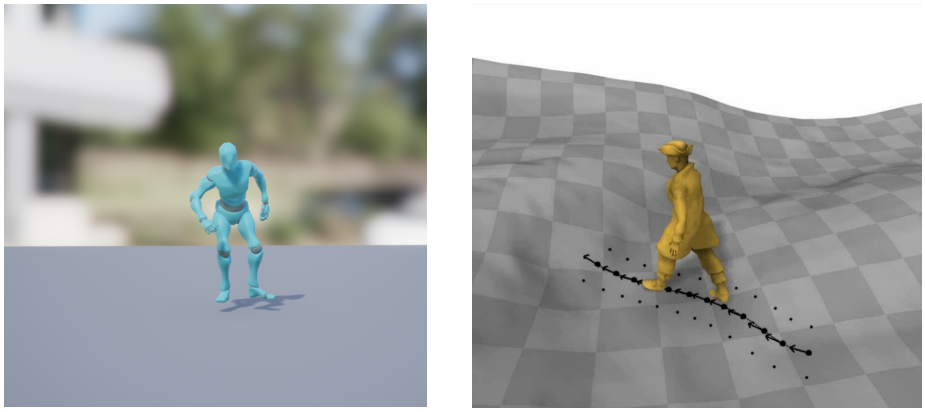


Figure 5.1: Examples different approaches. First example is our simulation. Second is the animation from Phase-functioned neural networks for character control" [8]

One of the most typical problems that appears in many approaches is the foot-sliding. This problem, as it is described in previous sections, happens when the character motion has different velocity than the captured one for the animation. What it causes is that when a foot is on the floor, instead of stay in the same point during a step, it slides through the floor. This problem is solved in our approach because we use the root motion of the animations for the motion of the character even while there is animation blending.

One problem that we observed in our simulation with some animations, was that if the animations had a center of gravity very different from the ones in the base blend space, then the position of the simulated character could appear tilted after a period of time. Fortunately, this artifact does not appear very often.

If we compare the result against one of the most realistic approaches in the state of the art that, which is "Phase-functioned neural networks for character control" [8], we can observe that although our approach does not have all the motions that it can do (such as turn backwards along a close curve or pass above obstacles) the basic animation clips for a character are very similar. This means that if both approaches contain the same set of basic walking, running and turning animation, we achieve very similar results from a qualitative point of view. Moreover, our approach can be used for a crowd simulation in real time, while theirs can only handle one or two characters in real time. So we can see that ours offers a good realistic behaviour with minimal performance cost.

5.2 Performance result crowd simulation

The next evaluation of results is based on measuring performance. Since the idea is to use it for crowd simulation, we need to know how it behaves in this kind of simulation. To analyse it, we measured how many avatars could be animated at the same time in a scene. To do so, we used a scene that only had a floor as an additional element. The avatars moved randomly around the scene and the parameter was updated every 3 seconds to change the direction of the avatars. With these preambles we runed the experiment. The idea was to measure the fps of the simulation while the number of characters kept increasing. To avoid UE4 not computing the characters that fall outside of the view frustum, an air camera was used for this experiment to fit all the characters at the same time as can be seen in figure 5.2.

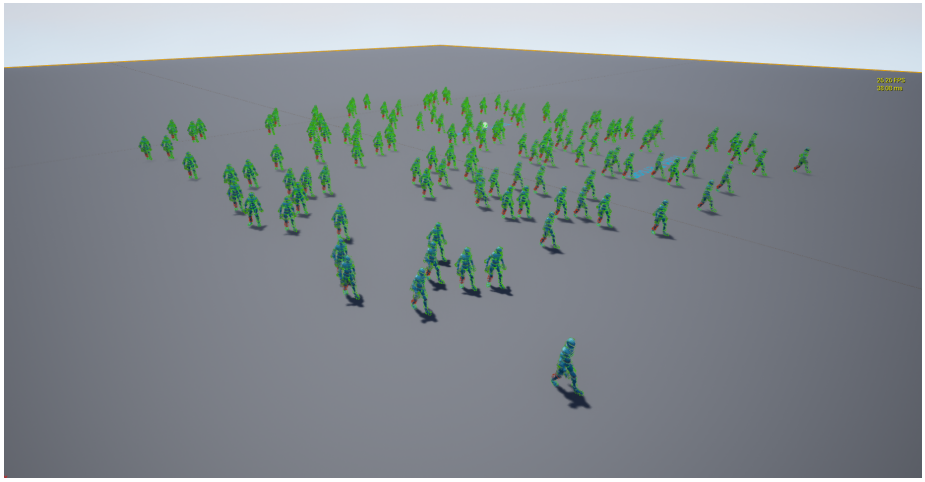


Figure 5.2: Example of an experiment execution.

The experiment started with one avatar and then increased the number by 10 every measured step. For every step, we computed the simulation with the original blend space, the blend space with both 2D blend spaces, with one 2D and one 1D blend space and with two 2D and one 1D blend space. The figure 5.3 shows the results for use only one blend space and use two 2D blend space and one 1D blend space. This is done to differentiate better the results since the other configuration results are in between of these two.

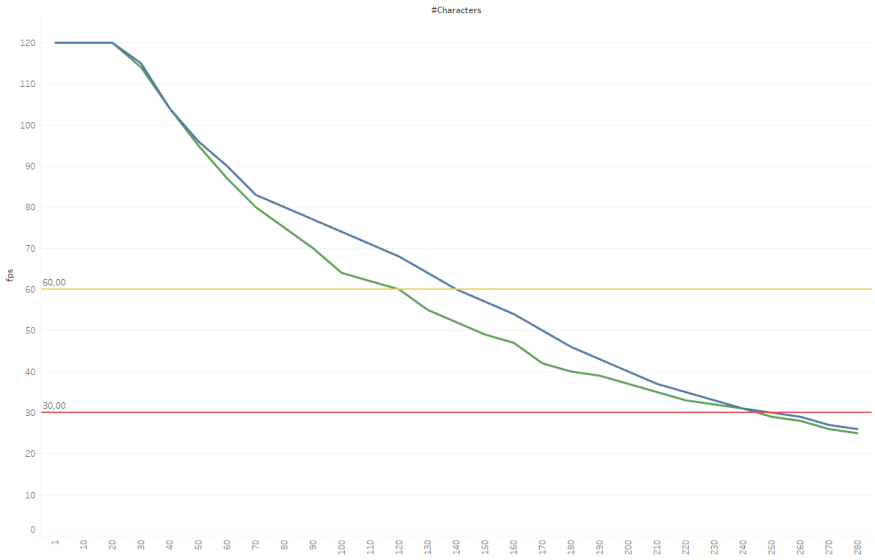


Figure 5.3: Results of the experiment. Line blue represent experiment with only one blend space. Line green represents experiment blending three blend spaces(two 2D blend spaces and one 1D blend spaces).

Figure 5.3 shows some interesting results. It only shows two lines for clarity purposes. The lines represents the extreme cases, in blue only one blend spaces and in green the case of three blend spaces. At the beginning the fps is constant at 120 because UE4 has limited the fps to this value. If we like to use it in VR applications the lowest rate we can use is 60 fps, which is achieved if we have in the screen less than 120 avatars with three blend spaces and 140 with one. In the case of using it in a real time application without VR the lowest rate is 30 fps and it can support 250 avatars approximately for both. The experiment, as it is mentioned before, has been tested with multiples configurations for the behaviour and the result is that the changes only take an oscillation with a maximum of 10 fps fps depending on the case. This experiment has some error rate because the fps is not fixed in time and it oscillates during the simulation.

As we expect, the cost of using only one blend space is lower than using three, since using three it needs to execute the process of using only one but three times and apply two blend functions to obtain the result.

Chapter 6

Conclusions and future work

The following sections explain the conclusion obtained with the different parts developed in this project and the results. Moreover, is explained the different ideas for continue working in this project.

6.1 Conclusion

In this project, we present a few tools and ideas to help in the generation of character animation synthesis using Unreal Engine 4.

Some of them such as the foot detection tool, which starts based on existing code, and the automatic blend space generator. Although there is room for improvements, our current framework can help the user to obtain better results.

The first, although is not automatic and it still needs some manual effort, it allows to increase the precision of the foot detection and to decrease the time needed for this task. With our solution, the user only needs to adjust a parameter and it generates the symmetric sync markers needed. Moreover, it gives the user the motion curve that helps in this adjustment.

The second is a simple method to generate blend space automatically, but although being a simple method, the structure generated provides good result compared to the current state of the art. Moreover, this method is easy to adapt to other kinds of parameters because it only needs to compute them tak-

ing the root motion of the animations. Furthermore, the result of this method gives an unexpected information about the real behaviour of the animations used and helps to identify the motions that are not well represented with the animation set chosen. So an additional functionality of our method could be to provide a realistic overview of the behaviors that can be covered with a given animation set.

The other part developed in this project consists on increasing the realism and the behaviour of the character in a simulation. The result of this method has a good performance, because it does not increase so much the computational time for each frame and thus allows to create different real behaviours in our simulations. For example, in the case of using one 1D blend space and one 2D blend space to simulate an injured character. Moreover, it decreases the number of animation clips needed, which decreases the time needed to do tasks such as modelling animations or searching for new clips. Furthermore, it solves, with a good result, the problem of only having 2D spaces in the UE4 blend spaces.

Finally, it is interesting to mention some conclusions about the development in Unreal Engine 4. It is a very powerful framework with which you can get very good results. As many people comment, and now I can also agree, it has a very slow learning curve and it is needed a large amount of time to have a good base knowledge of the framework.

The last part that I like to comment is about the coding in C++ in this framework. One of the possibles improvements is use this language in all the code but I like to remark a really interesting detail about why this is not as easy as it seems. When you create a project in UE4 you have the option of creating the base with Blueprint or C++. If you use the second, you will find the scripts of control made in C++ and some others. The interesting idea is that the animation scripts are created with blueprints and not with C++ so it can be noticed the difficulty of this task in C++.

6.2 Future Work

During this project many ideas for future work are opened. Some of this ideas are more advanced than other but there is room for improvements in all of them.

- For the part of foot detection, there are some algorithms that can be applied to achieve the automation of this problem. The main problem in this part is how to implement it using Unreal Engine 4, so improving our knowledge about it will be easier than implementing a new method.

- For the part of the blend spaces, the final idea is to implement our own generic framework disconnected from UE4. This will allow to us the possibility of change the game engine that we use or apply it in other kind of software. Moreover, it allows us to modify the triangulation algorithm that it has given some problems during the project, or delete the grid and allow to put the animations clip in any position of the space to delete calculation errors.

- Another future improvement related with the possibility to create our own blend space framework is, at the first instance, allows the 3D blend spaces, and in the future the possibility of multidimensional blend spaces. Of course this is only and idea and is not in our mind in the short time.

- For the part of automation of the blend space generation, we are thinking about more methods to develop this part starting by finding a better parametizations that can describe more accurately the real behaviour of a character using less parameters.

- Finally, one option that we are trying at the moment is to apply IK to the resulting animation to straighten the character that sometimes ends up twisted after some time (due to center of mass incompatibilities).

Bibliography

[1] UBISOFT: Francois Cournoyer. Massive crowd on assassin's creed unity: Ai recycling. `https://www.gdcvault.com/play/1022411/Massive-Crowd-on-Assassin-s`, 2015.

[2] Otger Roglà Pujalt, Núria Pelechano Gómez, and Gustavo Ariel Patow. Procedural semantic cities. In *CEIG 2017: XXVII Spanish Computer Graphics Conference: Sevilla, Spain, June 28-30, 2017*, pages 113–120. European Association for Computer Graphics (Eurographics), 2017.

[3] Nuria Pelechano, Bernhard Spanlang, and Alejandro Beacco. Avatar locomotion in crowd simulation. 2011.

[4] Alejandro Beacco, Nuria Pelechano, Mubbasir Kapadia, and Norman I. Badler. Footstep parameterized motion blending using barycentric coordinates. *Computers Graphics*, 47:105–112, 2015.

[5] Sean Curtis, Ming C. Lin, and Dinesh Manocha. Walk this way: A lightweight, data-driven walking synthesis algorithm. In *MIG*, 2011.

[6] Mubbasir Kapadia, Nuria Pelechano, Jan M. Allbeck, and Norman I. Badler. Virtual crowds: Steps toward behavioral realism. In *Virtual Crowds: Steps Toward Behavioral Realism*, 2015.

[7] Sahil Narang, Tanmay Randhavane, Andrew Best, and Dinesh Manocha. Fbcrowd: Interactive multi-agent simulation with coupled collision avoidance and human motion synthesis. 2016.

[8] Daniel Holden, Taku Komura, and Jun Saito. Phase-functioned neural networks for character control. *ACM Trans. Graph.*, 36:42:1–42:13, 2017.

[9] Franck Multon, Richard Kulpa, Ludovic Hoyet, and Taku Komura. Interactive animation of virtual humans based on motion capture data. *Computer Animation and Virtual Worlds*, 20(5-6):491–500, 2009.

[10] UBISOFT: Alexander Bereznyak. Ik rig: Procedural pose animation. `https://www.gdcvault.com/play/1022984/IK-Rig-Procedural-Pose`, 2016.

[11] Yi Lin. Character animation synthesis from 2 d sketches. 2007.

[12] Unity Technologies. Animation tab. https://docs.unity3d.com/Manual/class-AnimationClip.html, 2018.

[13] Inc. Epic Games. Unreal engine 4. https://www.unrealengine.com/en-US/what-is-unreal-engine-4, May 2018.

[14] Unity Technologies. Unity3d. https://unity3d.com/es/, May 2018.

[15] Crytek GmbH. Cryengine. https://www.cryengine.com/, May 2018.

[16] Richard Fine. Unityscript's long ride off into the sunset. https://blogs.unity3d.com/es/2017/08/11/unityscripts-long-ride-off-into-the-sunset/, 2017.

[17] Aras Pranckevičius. Releasing the unity c source code. https://blogs.unity3d.com/es/2018/03/26/releasing-the-unity-c-source-code/, 2018.

[18] Amazon. Amazon lumberyard. https://aws.amazon.com/es/lumberyard/, May 2018.

[19] Otger Rogla Pujalt. Procedural modeling of cities with semantic information for crowd simulation. Master's thesis, Universitat Politècnica de Catalunya, 2016.

[20] Inc. Epic Games. Introduction to blueprints. https://docs.unrealengine.com/en-us/Engine/Blueprints/GettingStarted, 2018.

[21] Autodesk Inc. Autodesk maya. https://www.autodesk.eu/products/maya/overview, May 2018.

[22] Blender. Blender. https://www.blender.org/, May 2018.

[23] Adobe Systems Incorporated. Mixamo. https://www.mixamo.com/#/, February 2018.

[24] Inc. Epic Games. Root motion in unreal engine 4. https://docs.unrealengine.com/en-us/Engine/Animation/RootMotion, 2018.

[25] Ue4 tutorial: Basic movement with root motion. https://www.youtube.com/watch?v=4xF7AK4UjMI, 2016.

[26] Naughty Dog Inc.: Travis McIntosh. Animation and player control in uncharted: Drake's fortune and uncharted ii: Among thieves. http://www.gdcvault.com/play/1012300/Animation-and-Player-Control-in, 2010.

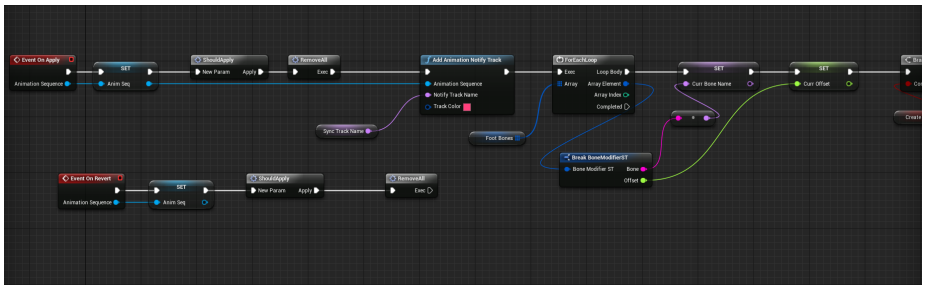
[27] Giuseppe Portelli. Automated foot sync markers using animation modifiers in unreal engine. http://www.aclockworkberry.com/automated-foot-sync-markers-using-animation-modifiers-unreal-engine/, 2017.

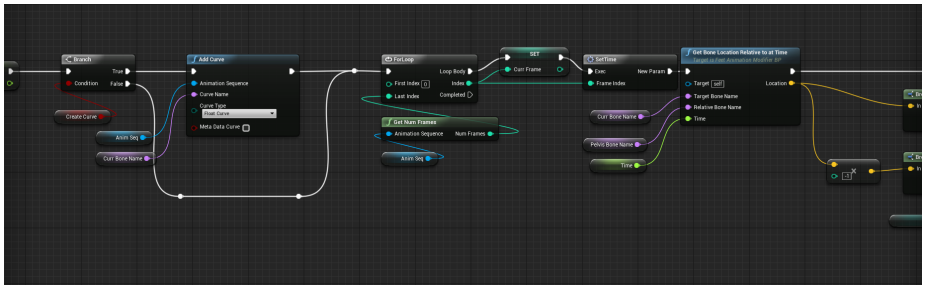
Appendices

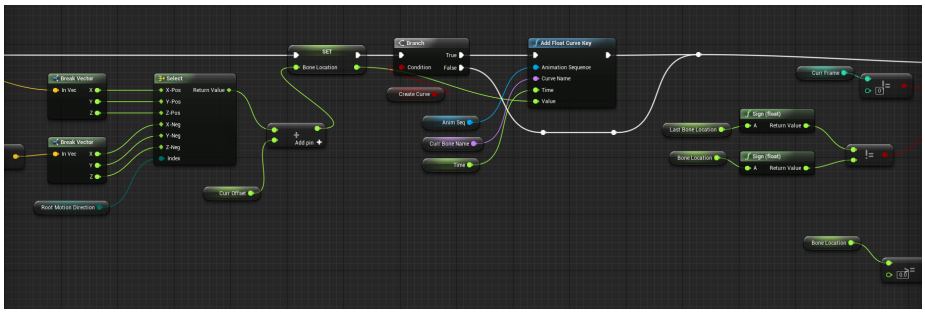
Appendix A

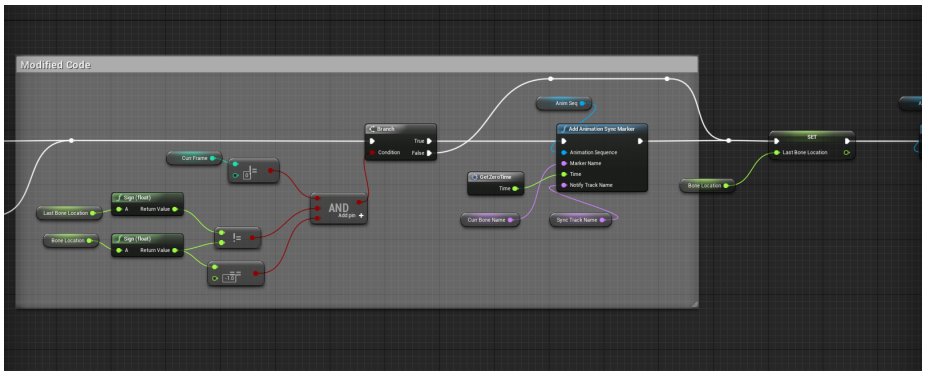
Code of Foot Detection

This is the code used in the foot detection part that it is based in other code from the UE4 comunity [27].






Break Vector
Select
SET
Branch
Add Float Curve Key
Create Curve
Add pin
Curr Offset
Root Motion Direction
Curr Frame
Sign (float)


Modified Code
Branch
Add Animation Sync Marker
SET
AND
Curr Frame
Sign (float)
Anim Seq
Curr Bone Name
Sync Track Name
Last Bone Location
Bone Location