João Carlos Santágueda dos Santos Claro

Licenciado em Engenharia Informática

Tool for Spatial and Dynamic Representation of Artistic Performances

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientadores : Fernando Pedro Reino da Silva Birra,
Prof. Auxiliar, Universidade Nova de Lisboa
Nuno Manuel Robalo Correia,
Prof. Catedrático, Universidade Nova de Lisboa

Júri:

Presidente:	Prof. Dra. Ana Maria Dinis Moreira
Arguente:	Prof. Dr. João António Madeiras Pereira
Vogal:	Prof. Dr. Fernando Pedro Reino da Silva Birra

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro, 2014

Tool for Spatial and Dynamic Representation of Artistic Performances

Àquilo que mais gostamos e que nos torna naquilo que somos.

Acknowledgements

As this document represents entering a new stage of my life, I would like to give out a few words of acknowledgment to show my gratitude.

First and foremost, I would like to thank my advisers, Fernando Birra and Nuno Correia, for their trust in me and their support throughout this dissertation's development.

Then, I would like to thank the Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa (FCT/UNL), for housing me for six years. I met great people here and had so many good times I will never forget, which helped me create much of the person I am today. I also have to thank the IT Department (DI) for providing me the much needed tools needed to develop this project.

I would also like to thank my colleagues for letting me experiment most of my work with them, helping me and laughing with me along the way.

I am forever grateful to my close friends, with whom many nights were spent after work, allowing me to relieve most of this stress inducing process called academic life. Thank you for sharing my life with you.

The most important thanks goes to my family, especially to my parents and brother. It is because of them I was able to get as far as I am today. I know many sacrifices were made, many hard times we went through together, but we proved we can do anything if we believe and never give up.

Finally I would like to thank coffee and *Coca-Cola* for being the best drinks in the world, Video Games for simply existing and Music for making the world go 'round.

Abstract

This proposal aims to explore the use of available technologies for video representation of sets and performers in order to serve as support for composition processes and artistic performer rehearsals, while focusing in representing the performer's body and its movements, and its relation with objects belonging to the three-dimensional space of their performances.

This project's main goal is to design and develop a system that can spatially represent the performer and its movements, by means of capturing processes and reconstruction using a camera device, as well as enhance the three-dimensional space where the performance occurs by allowing interaction with virtual objects and by adding a video component, either for documentary purposes, or for live performances effects (for example, using video mapping video techniques in captured video or projection during a performance).

Keywords: Video, three-dimensional reconstruction, three-dimensional modelling, motion capture.

Resumo

Esta proposta visa explorar a utilização de tecnologias disponíveis para a representação digital de cenários e autores das artes performativas de modo a servirem de apoio a processos de composição e ensaios, focando em especial na representação do corpo dos artistas e seus movimentos e na sua relação com objetos pertencentes ao espaço tridimensional das atuações.

A proposta visa o desenvolvimento dum sistema que permita representar espacialmente o artista e os seus movimentos, por processos de captura e reconstrução, assim como aumentar o espaço tridimensional onde o artista atua permitindo-o interagir com objetos virtuais e adicionando uma componente vídeo, quer para efeitos documentais, quer para efeitos de atuações ao vivo (por exemplo, através da utilização de técnicas de *video mapping* no vídeo capturado ou projeção durante uma atuação).

Palavras-chave: Vídeo, reconstrução tri-dimensional, modelação tri-dimensional, captura de movimentos.

Contents

List of Figures

List of Tables

Listings

1

Introduction

This dissertation addresses virtual representation of body movements and the surrounding space in a performance over time in the context of using software for artistic rehearsals purposes. This is a fairly recent concept, due to the rise of low cost motion capture cameras, and although there are already some experiments in this field it is currently under-explored.

In this chapter, we further detail the main motivations and context surrounding this dissertation, pointing out the different options and limitations in this field, and present the overall layout of this document.

1.1 Motivation

The field of artistic performances has been evolving over time, incorporating digital systems, making way for several applications to be created, either just to give a way for performers to study a routine or to give the viewers a richer experience with what is currently happening on stage.

The core of most performing arts is their routines, repeatedly performed procedures, and there are mainly two ways to learn them. The first way is through lessons, where a teacher can point out mistakes and give advice on how performers can improve their skills. The second way is through self-learning, which involves more time and effort. This is when digital information can come in handy by, for example, capturing performer's routine movements and simulating them on a virtual stage, making it possible by detecting all unwanted movements and correcting them on the spot, without any external help.

1.2 Context

Moore's Law[1] states that in the course of time the number of transistors on integrated circuits doubles approximately every 18 months, meaning that powerful electronic devices will become smaller in size over time. This rapid increase in technology allows for the creation of several capture methods where, as an example, performers can attach various small motion capture devices to their limbs, capturing their performance with good precision, while still making the performer feel comfortable. As an alternative, by using motion sensor cameras it is also possible to capture these movements in a non intrusive way, without the need to attach any peripherals to the performer, although the movements captured will probably be less accurate. The ability to record and analyze body movements can be very useful in several fields, such as artistic or theatrical performances, but can also be useful in other fields that require body movement analysis, such as sports.

By using 3D reconstruction methods, it is possible to recreate a stage where the performance can occur in virtual form. However, this process is not straightforward and requires complex algorithms and the use of very specific input devices, such as the Microsoft Kinect, a low budget motion sensor camera [1].

Motion sensor devices can also be used for gesture recognition. This means that by performing specific gestures, performers can trigger certain events on a virtual stage, such as materialize objects, stretch them, make them appear or disappear, and several other visual effects. This allows for a different play on traditional performances, mixing both real world and virtual world by enabling real world human movements to influence what happens on a virtual stage.

By blending these concepts we can accurately capture the performers' movements, the stage where they are performing and define their interactions with the stage elements.

1.3 Presented Solution

As mentioned before, there exists an inherent challenge in both capturing and depicting the routine movements in an accurate and precise way. Therefore, this project's goal is to create a software tool that allows users to interact with a virtual stage using only a 3D virtual model, making use of a specific input device, such as the Microsoft Kinect, for motion sensor purposes. This software will serve as a support for composition processes or rehearsals, where the user can focus on specific body movements done by either a real or a virtual performer.

The tool is developed in C++ using the openFrameworks platform [2], a library dedicated to creative graphical programming, specifically using the Kinect Common Bridge [3] add-on, which is a wrapper for the official Kinect Libraries. In order to accomplish the 3D graphical simulation our tool will communicate with Blender (section 2.4.1), a 3D

[1] http://www.mooreslaw.org/

modeling software with an internal game engine, through add-ons developed in Python language.

Our initial system design is divided into two main systems, the motion capture system and the 3D representation system, as represented in figure 1.1. The motion capture system focuses on capturing and interpreting the performer's movements, being capable of detecting performers in front of a camera and represent their performance in some data format in real time. It also allows for playing recorded animation files instead, that follow the same movement data format. The 3D reconstructing system focuses on representing the performer's body and the stage where the performance occurs in a virtual environment. The emulated 3D model is able to interact with any virtual objects in the virtual stage it is represented in. Video input is also used as part of the 3D representation system. When combining the two systems, a simulation can happen, in which a previously created 3D model mimics the performer's body movements, either detected through skeleton tracking in real time, or through animation files instead.

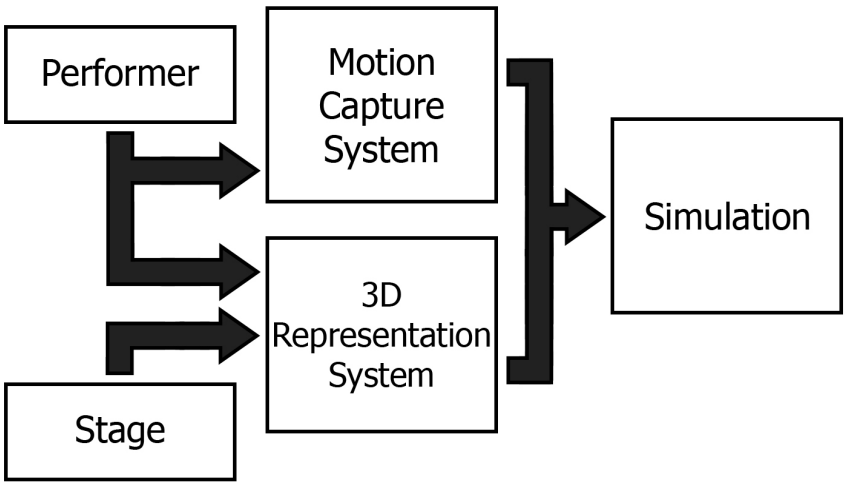


Figure 1.1: Diagram of the project's initial architecture.

1.4 Document Structure

This document is divided into five main chapters - introduction, related work, concept and initial prototype, final prototype and conclusions and future work.

The first chapter, **Introduction**, presents an overview of the dissertation, addressing several issues such as the main motivation and context for this dissertation, and our presented solution.

The second chapter, **Related Work**, focuses on systems and tools similar to the proposed solution, where some of their features or techniques are relevant to the solution.

The third chapter, **System Model and Features**, details the system model and how

each of the features is implemented. The initial steps of the development cycle are also defined in this chapter.

Then, in the **System Development** chapter, a final prototype is presented. All of the final intricacies regarding the development cycle are explained here.

Finally, the **Conclusions and Future Work** chapter presents the overall conclusions regarding this project, combined with evaluation aspects, followed by the future work possibilities regarding our work.

2

Related Work

This chapter presents concepts and techniques used in the development of the proposed solution. In the following sections, several tools, techniques and systems relevant to the solution will be presented in detail. This chapter is composed of three sections. The first section addresses digital tools used for representing artistic performances. The second section is dedicated to tools used for motion capture and finally the third section discusses the several options available in 3D simulation software.

2.1 Digital Tools for Artistic Performance Representation

This section covers already existent digital tools with a similar context to our presented approach that are considered relevant to the solution.

2.1.1 Virtual Reality Theatrical Performing

Artists in the world of traditional theatrical performing have always been looking for new ways to entertain the viewers. One way to give viewers a new and refreshed experience is through virtual reality, and some of the ideas explored so far include creating a fully virtual performer avatar who acts by recognizing a performer's actions, as seen in figure 2.1 [4], and even project holographic 3D stages [5], giving users a complete virtual theater experience.

Besides having the full traditional theatrical experience, a virtual stage enables the possibility of adding elements that would not be possible in the real world, for example, imaginary or non-human characters, impossible actions like "summoning magic",

or even simulating things that would be too dangerous to do on a real stage like pyrotechnics.

However, this transition from the real world to a virtual one is not seamless, mostly because of its need for pre-programmed actions. If these actions are to be captured by non intrusive motion cameras, such as the Kinect, they depend on the environmental conditions where the actions are captured, taking the risk of the system not working properly. Virtual performers or avatars also usually follow a script, acting based on pre-recorded actions, not allowing them to be spontaneous and capable of any form of improvisation.

Using gesture recognition, it is possible to simulate improvisation, making a virtual avatar act based on what actions are performed by a real world performer, giving the illusion of responsiveness [4]. In order to make the human-avatar interaction seem as spontaneous and human-like as possible, a large database of gestures and actions is needed. This requires an extensive list of avatar actions that need to be previously captured or pre-programmed, also requiring the performer to know every single gesture that makes the virtual avatar react, just to give the illusion that the avatar is acting on its own free will.

These approaches could also be combined into a mix between scripted sequences and inferred sequence. These last ones could be defined through machine learning by, for example, understanding the last movements executed by the performer and recognizing the sequence as a trigger for executing another action in stage. With this, any performer could define a sequence trigger, "teaching" it to the artificial intelligence system.

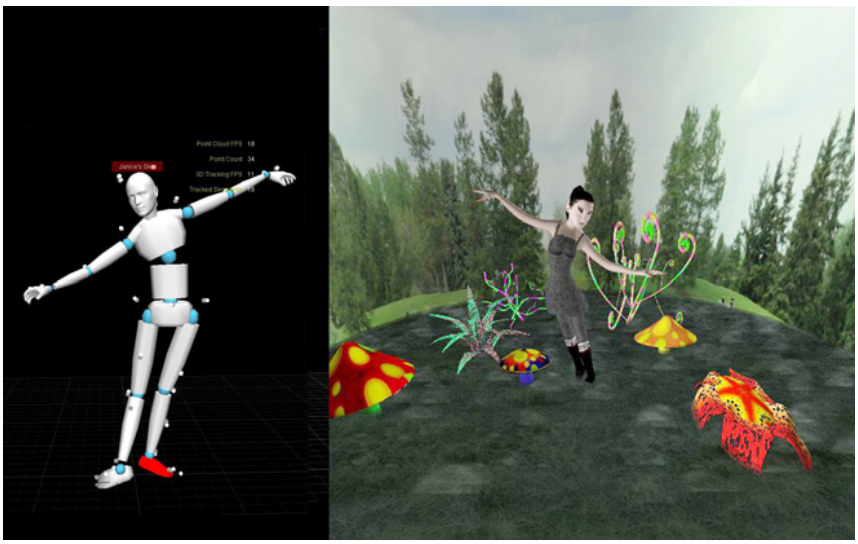


Figure 2.1: More elements are added to the virtual stage as a triggered response to the performer's actions [4].

2.1.2 Virtual Reality Dance Performing

As mentioned before in Chapter 1, some dances are based on routines, which define the performer's procedural body movements over a period of time. Along with the introduction of motion capture methods, new ways to perfect these routines started to appear. It became possible to record performances and perform a quick analysis of what movements were made, detecting and correcting all non intended movements.

This technique has previously been used to record dance routines for both solo practice or group practice. Baird *et al.* presents an example of how routines can be recorded and reproduced in digital form [6]. They designed a tool that uses a library of small movements that follow Laban dance notation, where users select which movements from the library to perform and capture them wearing a motion capture suit [7]. The tool takes the recorded movements and places them in a 3D virtual environment, where the user is free to rearrange the order of the recorded movements to its liking, while getting feedback through a virtual avatar performing the sequence of movements selected.

Chan *et al.* created a system for performers to improve movements in an interactive way, consisting on using a virtual avatar that can teach and perform dance routines, making users match their movements [8]. These dance routines must be previously captured, and are performed by a professional dancer wearing a full body motion capture suit, as seen in figure 2.2. When using the system, users must also wear a full body motion capture suit and just follow the instructions given by the virtual teacher. The user's performance is then compared to the teacher's and a score is given. This score is computed based on the euclidean distance of each of the joint positions, comparing the template posture and the student's posture, averaged throughout all of the frames. However, different body types will produce different results, for example, a person that is both short and large in stature will struggle enacting a position done by a tall skinny person. Because the score is based on an euclidean distance function, this difference in body types will not be that impacting, but enough to make a difference, not being equal to all body types. In the end, the point system is an interesting way of acknowledging how good the movements are performed, and by doing so, it encourages performers to improve.

There are also systems like ChoreoGraphics by Schulz *et al.*, that focus on solo and group performances. Wearing a motion capture suit, dancers can perform their actions guided by a musical track [9]. After recording, these performances can be divided into small steps, allowing the rearrangement of the same steps in whatever sequence. In addition, by combining several performances, it is possible to represent each one of them in a specific position on a 3D virtual stage, while performing their recorded sequences in synchronization, due to the same musical track tempo. By doing this, it is possible to analyze group performances as a whole, while still focusing on details of each individual performance.

Virtual reality has also been used for adding visual effects to the performance's actions. In project RAM visual effects are created based on what the dancer's position on

Figure 2.2: A performer using a full body suit for capturing dance movements for the virtual avatar to perform [8].

the stage is, and on their flow of movement. A tool was developed as a support for dance creativity, where visual effects are dependent on the skeletal tracking input [10].

Using the provided toolkit, the visual effects can be modeled and simulated on a virtual stage, based on a previously captured virtual performance, as demonstrated in figure 2.3. These virtual effects can then later be seen on the physical stage, when the dancers perform the rehearsed movements that trigger the modeled visual effects. This can be achieved using a standard optical motion capture system, a Microsoft Kinect, or Motioner, an open source system developed specifically for project RAM, consisting of 18 lightweight sensors attached to the dancer's body.

This system promotes creating and using different types of dance movements, where the viewers follow not only the dancer's movements, the visual effects that appear, but also how the effects interact around the dancer.

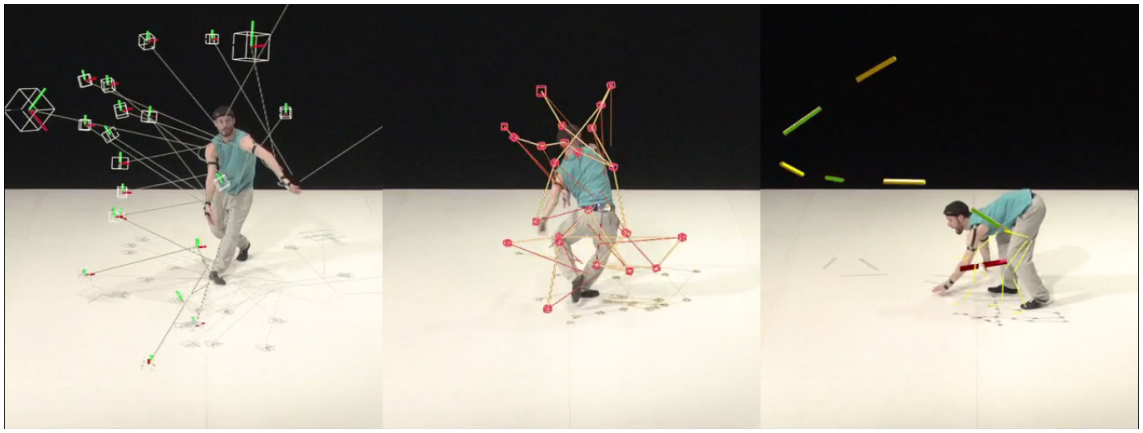


Figure 2.3: Project RAM's different visual effects created from the previous movements done by the performer [10].

2.1.3 Virtual Avatar Automatic Rigging

Virtual Avatars are composed of two separate structures, an external surface representation, also called skin or mesh, and the internal hierarchical set of interconnected bones, called skeleton or rig. Because the bone connections form a hierarchy, all transformations that occur on a parent bone node will propagate to all child bone nodes. For example, when animating a humanoid skeleton, if a thigh bone rotates then the lower leg and all of its children bones will also rotate.

A conventional rigging process requires manual input to make sure the body mesh follows the skeleton movements. This requires placing the skeleton joints inside the character mesh, while specifying which parts of the surface are attached to which bone. Because this process is time consuming and tedious, some automatic tools have already been created.

Ilya Baran *et al.* created a tool entitled Pinocchio that takes a character mesh, the corresponding skeleton and a motion of that skeleton as input, obtaining the character performing the given motion as the output, as seen in figure 2.4 [11]. The algorithm used consists of two phases: skeleton embedding and skin attachment. In the first phase, the skeleton's joint positions are calculated and placed inside the character. In the second phase bone weights are computed based on the proximity of the embedded bones, so that the algorithm knows how to apply deformations of the skeleton to the character mesh. To simplify the process, it is assumed that both the character and skeleton have approximately the same orientation, pose and proportions.

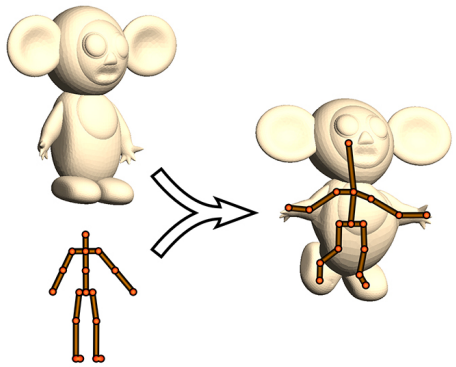


Figure 2.4: Using a static character mesh and skeleton as input, a moving animation is automatically created [11].

Several 3D modeling software options include tools specifically created for aiding in the rigging process. Rigify is an add-on for 3D modeling software Blender created for this purpose [12]. With it, it is possible to rig most biped characters, providing small individual armature parts, such as legs, arms or fingers. By connecting these individual parts in whatever way wanted, with just a button click the rig is automatically created. Rigify only automates the creation of the rig controls and bones, so the process of attaching the skeleton rig to a mesh must still be done manually. The created rig accepts only

rotational values, having a rotation node for each armature bone. When finally attached to a mesh, the effects of the rotation can be seen, as demonstrated by figure 2.5.

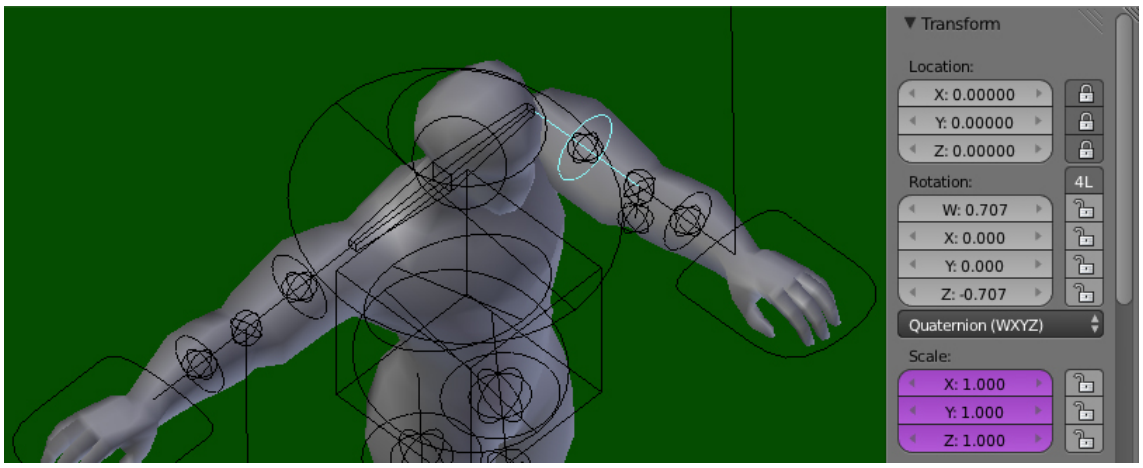


Figure 2.5: A rotation value was inserted in the selected node, rotating the model's arm. The node's position (Location), however, is locked and cannot be modified.

2.2 Motion Capture Cameras

As mentioned before, the main motion capture device used for this project is the Microsoft Kinect, so the main focus of this section is on the device's capabilities.

2.2.1 Microsoft Kinect

The Kinect camera was developed by Microsoft in cooperation with PrimeSense, being introduced to the public in June 2009 during the Electronic Entertainment Expo (also known as E3) and released in November 2010 [13]. Kinect consists of three different components working together:

- A standard color sensor, to retrieve RGB images;
- A depth sensor, comprising an infrared (IR) laser projector which shoots IR rays through the captured scene and a sensor that records them. The distance to the camera is then measured by the size and the position of the recorded IR dots;
- It also includes a built-in multi-array microphone to get audio information.

2.2.1.1 Color Stream Data

The Kinect camera has a default resolution of 640x480 at 30 frames-per-second, which can be increased, at the cost of a lower frame-per-second rate. This means that for high-resolution images more data per frame is sent, which makes it update less frequently,

while lower-resolution images update more frequently, but with some loss of image quality. These limitations are all hardware defined, because the of the infrared sensor's characteristics. The infrared image stream is a particular configuration of the color image stream as well, meaning that both these streams are supposed to function with the same resolution and rate. An example of the color stream data can be observed in figure 2.6(a).

The camera's color data is also available in different formats, able to be coded as RGB, YUV or Bayer formats, but only one resolution and format can be chosen at a time [14].

2.2.1.2 Infrared Stream Data

Infrared light is the electromagnetic radiation with lower frequencies than those of visible light. As a result, infrared light is used in industrial, scientific, and medical applications to illuminate and track objects without visible light.

The Kinect makes work of this feature, having a depth sensor which generates invisible infrared light to determine an object's distance (in millimeters) from the sensor [15]. An example of the depth and infrared stream data can be seen in figure 2.6(b) and 2.6(c).

In order to be able to infer depth from a scene, the infrared light is projected in a non uniform manner. Knowing this, the camera can match the patterns of dots to hard-coded images it has of the projected pattern. This technique is also called Structured light, and in Kinect's case, it is also combined with two other computer vision techniques [16].

The first technique is Depth From Focus, which is based on the principle that what is farther away will become more blurry. The second one is called Depth From Stereo, and it uses the parallax concept. The Kinect camera lens is astigmatic, having a different focal length in x- and y- directions. By projecting from one position and observing it from another, it is possible to detect the shift of the speckled pattern [17].

The Kinect's internal processor is then able to use this information to triangulate the three-dimensional position of the recorded points. Using this, it is possible to use depth data to track a person's motion or identify background objects [18].

Because the infrared image stream is a particular configuration of the color image stream, it is not possible to have a color image stream and an infrared image stream working at the same time on the same sensor [19].

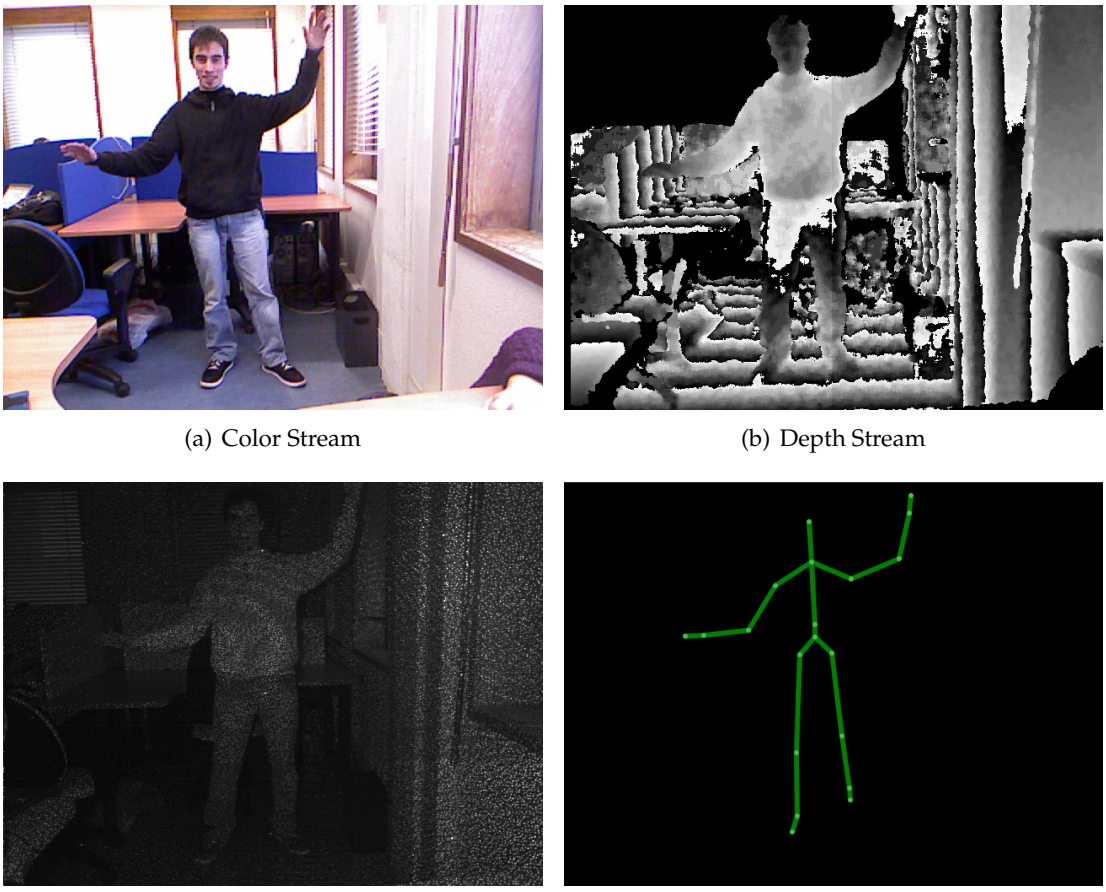
2.2.1.3 Skeleton Tracking

Skeletal Tracking is a feature that allows Kinect to recognize people and follow their actions. Using the infrared camera, it can recognize up to six users in the sensor's field of view, while up to two users at a time can be tracked in detail [20]. An internal application can locate the joints of the tracked users in space and track their movements over time, as demonstrated in figure 2.6(d).

This feature is optimized to recognize users standing or sitting, while facing the Kinect. To be recognized, users simply need to be in front of the sensor, making sure

the sensor can see their head and upper body. No specific pose or calibration action needs to be taken for a user to be tracked.

Because the human body is capable of performing an enormous range of poses which are difficult to simulate, the best way to recognize a human body's position is to have a large database of previously captured human actions [21]. Body parts are then inferred using a randomized decision forest, with over 500,000 examples of specific positions like driving, dancing, running or navigating menus.



(a) Color Stream (b) Depth Stream

(c) Infrared Stream (Enhanced) (d) Skeleton Tracking

Figure 2.6: An example of the Kinect basics: color stream data (a), depth stream data (b), infrared stream data (c) and the corresponding skeleton tracking (d).

2.2.1.4 Official Kinect SDK

In order to make use of the Kinect's features, Microsoft freely distributes the Kinect for Windows Software Development Kit (SDK) and Kinect Developer Toolkit (KDT), for anyone to use [1]. These kits contain drivers, tools, APIs[1], device interfaces, and code samples in programming languages such as C++, C# and Visual Basic, necessary for building

[1] Application Programming Interface

applications. This SDK is exclusive to Windows operating systems, but there are open source adaptations of the Kinect SDK libraries for other systems, such as OpenNI [22] and OpenKinect [23], for both Linux and Mac operating systems.

The first version of the Kinect SDK for non-commercial use was released in June 2011, already including functionalities such as retrieval of color and depth sensor data, skeletal tracking and audio processing capabilities. Along with version 1.5, which came out February 2012, support for new functionalities were added such as a seated skeleton tracking mode, more speech recognition languages and the Kinect Studio application, which allows users to record, playback and debug clips while interacting with applications which use Kinect input data. This application is particularly useful for offline testing, since a single Kinect Studio file includes all the available data streams, including depth and skeleton tracking data. In version 1.7 of the SDK, Kinect Fusion (mentioned in Section 2.3.1) was released, and since then, only one more version (1.8) was released for this device. A whole new device was released along with the 2.0 version, to go along with Microsoft's new home console Xbox One. The new Kinect features are specified in section 2.2.2.

Many of the code samples provided were experimented on, focusing on obtaining color, infrared and depth images, skeleton and face tracking and how the Kinect fusion actually works. The color, depth and infrared streams, skeleton and face tracking code samples are very straight forward, simply demonstrating how to use the corresponding data streams, and how to export those same data streams into image files.

2.2.2 Microsoft Kinect Version 2.0

The recently launched Kinect for Windows v2 sensor, released in Europe in September 2014, has the same basic hardware features as the previous Kinect, but improved their potential in all aspects [24]. Although it carries the Kinect name, it is a completely new device, not being backwards compatible with version 1.8 of the SDK, and also requiring Windows 8 as an operating system.

The sensor's new color camera can now capture and display data in full 1080p resolution video, which is a big improvement in quality from the previous 640x480 resolution. The new depth sensor allows tracking up to six people simultaneously, having an upgraded positional recognition of the skeleton's joints. The number of joints also increased to 25 joints per person, allowing for a more complete body tracking. Figure 2.7 demonstrates the depth sensor differences between devices. Facial tracking is now possible, whereas previously it was not, when using a Kinect for Xbox device.

A new Visual Gesture Builder application was introduced in this SDK version, where a user can create and customize new gestures to be recognized by the camera. These gestures are added to the already extensive list of gestures included in the SDK for this purpose. This version now allows building and publishing apps using the Kinect features in Unity's 3D development environment. It also allows for multiple applications to access

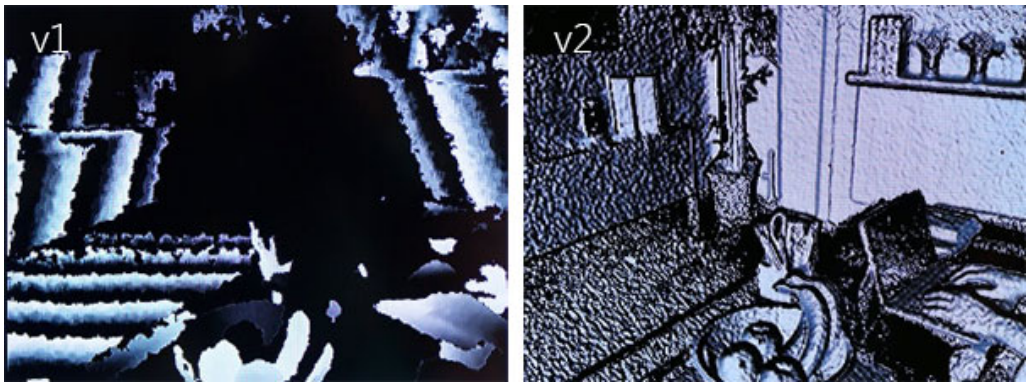


Figure 2.7: Comparison between the images provided by the depth sensors of the Kinect v1 (left) and the Kinect v2 (right) [24].

a single device simultaneously, not granting exclusivity to a single application.

Instead of using Structured Light, this new device's infrared sensor now uses the Time-of-flight technique [25]. Cameras using this technique are able to provide a real-time 2.5D representation of an object, since only a part of the surface can be seen by the camera. This object is illuminated with an incoherent light signal coming from the infrared emitter, which is reflected on the surface of the object, only to be recaptured by the infrared sensor. When this happens, not only depth values are acquired, but also the intensity of the reflected light signal allows for a better representation of the data received. With this technique, the device's response time is increased from 65 milliseconds to less than 14 milliseconds.

When comparing with Structured Light, by using Time-of-Flight it is not necessary to use Depth From Focus or Depth From Stereo anymore, since the 2.5D object representation and light intensity values achieve what these techniques are meant to do. Also, depth values at non-illuminated points have to be derived via interpolation, which means more computation time spent [26]. For Structured Light devices an initial calibration is also required to be able to map the 3D point values, because of the unknown light pattern which is emitted in a non uniform manner, as mentioned in section 2.2.1.2.

2.3 Motion Capture and 3D Representation Assisting Tools

Since we will be working specifically with Kinect devices for capturing motion data, a research was conducted on already existing tools that specialize in creating or manipulating motion capture data and 3D representation of human bodies.

2.3.1 Kinect Fusion

In March 2013, the first installment of Kinect Fusion was released with the official Kinect for Windows SDK version 1.7. Kinect Fusion is an application which provides 3D object scanning and model creation tools using the Kinect camera [27]. Users can take depth

images of a scene or an object with the Kinect camera, and create a detailed 3D model constructed in real time, being also capable of exporting the created 3D mesh in STL, OBJ and PLY formats.

Kinect Fusion takes advantage of a process known as Simultaneous Location and Mapping (SLAM) [28], in which it is possible to reconstruct a single dense surface model with smooth surfaces by integrating the depth data captured from the Kinect camera over time and from multiple viewpoints [29]. To do this, an algorithm known as Iterative Closest Point (ICP) is used [30]. As the camera moves around the scene, a different perspective is captured at every frame. The ICP algorithm then repeatedly rotates and translates the current frame until it finds the best match with the last frame's pose. By calculating how each frame relates to the others, it is possible to stitch these frames together into a single reconstructed voxel volume.

2.3.1.1 Simultaneous Location and Mapping Process

The SLAM process starts by receiving a depth frame from the camera. Usually this captured raw data is often noisy, so a bilateral filtering is applied to it, smoothing the noise in the data, but still maintaining sharp transitions between pixels, as we can see in figure 2.8 [31]. This filtered image is then scaled down twice, once to half-size and another for a quarter-size, creating three different scale images. These copies are very useful for the ICP alignment process, using the smaller copy for a quick rough alignment and the more detailed one for a more refined alignment.

In order to run ICP the scene must be rotated and translated and so, for mathematical purposes, the images are converted from a pixel based to a 3D coordinate based representation, resulting in a vertex map and a normal map for each of the three depth images. The vertex map contains a list of points where each point has 3D coordinates that represent the surface to measure. The normal map is also a list related to the same points, where each entry is the direction the surface is facing. These maps are always used together so that each vertex has a position and orientation.

To give the user some feedback on the scan's development, a Truncated Surface Distance Function (TSDF) volume (described in 2.3.1.2) is converted to an image. Finally, points are extracted from the more refined TSDF volume for the next round of ICP. This is done because the previous ICP alignment probably had a tiny amount of error, but if this were not to be corrected, errors would aggravate.

The SLAM process should occur 30 times per second, at the rate the depth frames are sent from the Kinect. If it is any lower than that, frames are going to be skipped, making it difficult for ICP to be successful. This process requires a large computing power and must always be running at full speed, and to do so, it is broken up into small chunks and run in parallel on the graphics card, taking advantage of the fast instruction processing speed done by the thousands of processor units that graphics cards contain.

(a) Original (b) Filtered

Figure 2.8: An example of bilateral filtering application: a picture with no filtering (a) and with bilateral filtering (b), smoothing the differences between pixels [18].

2.3.1.2 Truncated Surface Distance Function

To represent the vertex map and normal map that describes a surface in memory, an algorithm known as Truncated Surface Distance Function (TSDF) is used [32]. This function makes hand-held scanning on personal computers feasible, and allows for a continuous refinement of the captured model.

When scanning, the real world object is reconstructed within a virtual volume consisting of a grid of voxels. To each voxel is assigned a distance value and a confidence value based on its orientation, for example, a surface facing the camera is given a higher weight for being, most likely, more accurate than a surface at an angle.

These values are representative of an accuracy estimate for the voxel's distance and are generated as follows: By drawing a line from the camera through each vertex in the voxel grid, more voxels are going to be intersected. For every intercepted voxel near the surface, the distance value is updated, calculating the distance from the current vertex to the center of the intercepted voxel. This process is repeated for each intercepted voxel and for each vertex in the voxel grid. As the camera moves around, the TSDF volume voxels are continuously updated and refined, as demonstrated in figure 2.9.

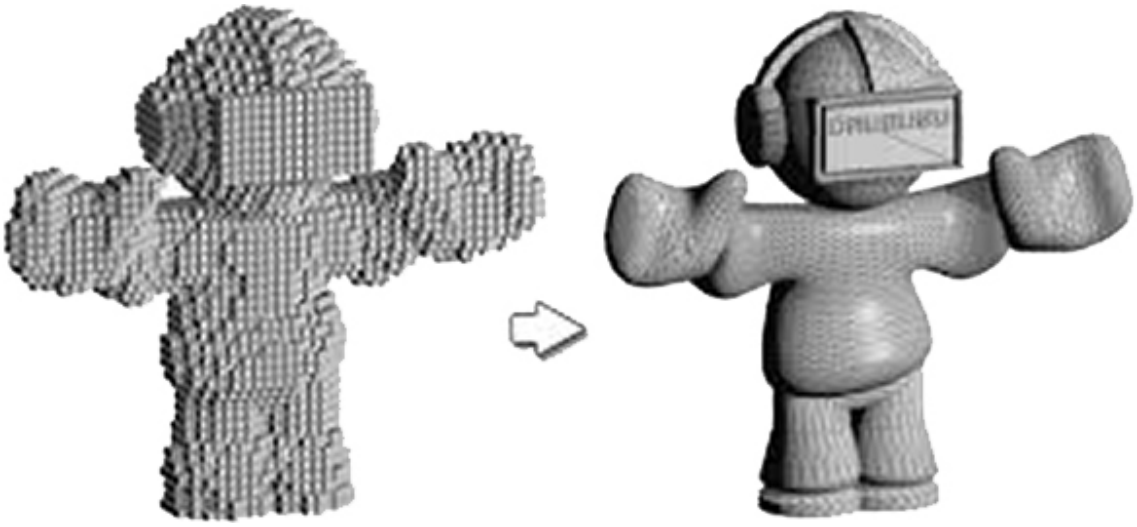


Figure 2.9: The model starts being unrefined, but after continuous updates a state of good definition is achieved [29].

2.3.2 Skanect

When searching for 3D reconstruction systems, Skanect was one of the first ones that was found [33]. Using low cost motion capture cameras like the Microsoft Kinect and Asus Xtion [34], it allows capturing full color 3D models, while also being capable of creating and exporting 3D meshes, similar to Kinect Fusion. These meshes can be later loaded in other 3D modeling software such as MeshLab [35] , Blender (Section 2.4.1) or Autodesk 3ds Max [36].

When starting Skanect, several options are presented. It is possible to scan a new mesh, load a previously scanned mesh or configure settings. When scanning a new mesh, Skanect provides feedback in real time, inside a bounding box, of what is to be scanned, also showing the color and depth images captured in real time by the camera. There are options to include a time delay when starting a recording, and also to limit the time of the recording. On pressing the record button, the mesh starts to be constructed, and as the camera is moved around the target, the mesh gets more refined.

What differentiates Skanect from Kinect Fusion is the ability to edit the recorded mesh before exporting it, as seen in figure 2.10. Skanect supports many mesh editing options, making it possible to reduce its number of faces, rotate and translate, remove unwanted mesh parts, color the mesh using the corresponding color data captured and also repair surfaces based on Convex hull algorithms [37].

2.3.3 Brekel Kinect

Brekel Kinect is a system very similar to Skanect, in the fact that it is also a 3D reconstruction system, capable of building and exporting 3D meshes using a low cost motion capture camera [38]. In addition, it allows recording skeleton tracking movements and

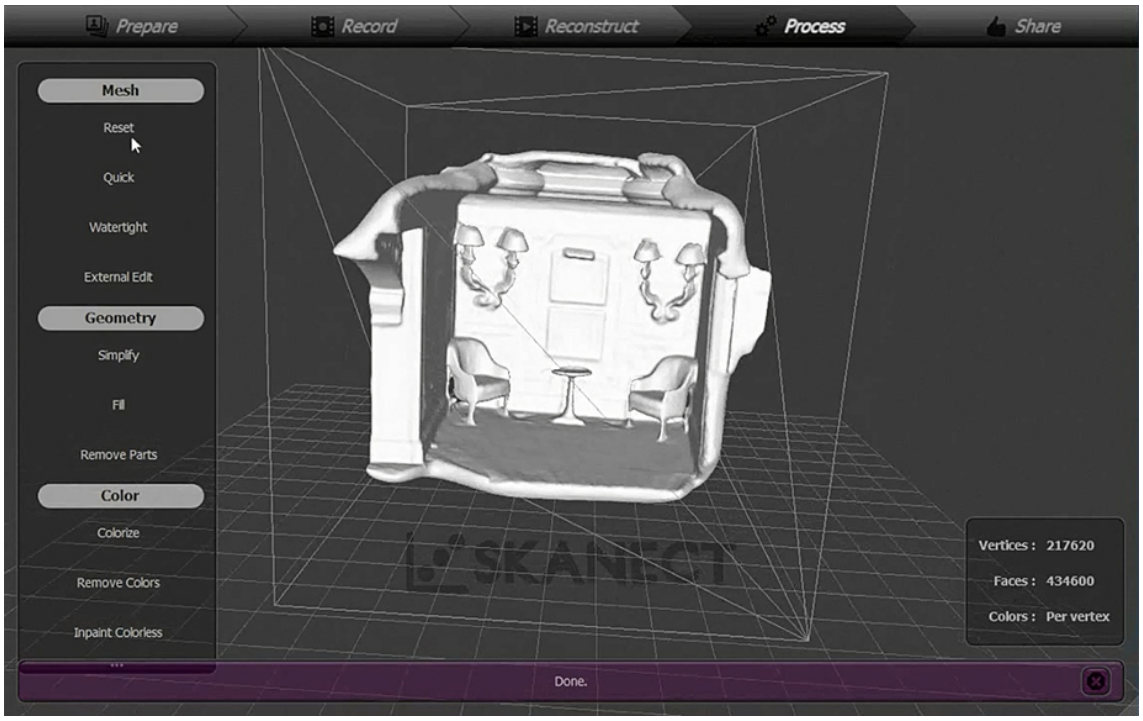


Figure 2.10: A reconstructed 3D mesh of a room using Skanect [33].

export them in BVH (Biovision Hierarchy) format, providing skeleton hierarchical information as well as motion data, or stream it into Autodesk MotionBuilder [39] in real time.

These features are all bundled in the free version, however, there are three more paid versions which specialize in other features such as enhanced point-cloud recording, face tracking or enhanced body motion capture. Unlike Skanect, that is built to be simpler and more user friendly, Brekel allows for a more advanced tweaking of settings, as seen through the menu options in figure 2.11. For example, when reconstructing a scene using point-cloud, it is possible to make small tweaks to the machine's performance by specifying if the processor should work in single or multi-threaded mode, while also controlling the minimum and maximum depth to capture and also the size of all points captured. This performance aspect can all be monitored through the frames-per-second counter and CPU usage percentage graphs.

Using the Kinect's skeleton tracking feature, it can also generate a BVH animation file, which can be used in any 3D animating tool. Being originated from the Kinect, this BVH file contains the right amount of bones, and their hierarchy, for creating animation rigs to use with the Kinect captured data.

2.3.4 NI Mate

NI Mate is a small but powerful piece of software, developed by Delicode, which uses real time capture data from a low cost motion capture camera such as the Microsoft Kinect or

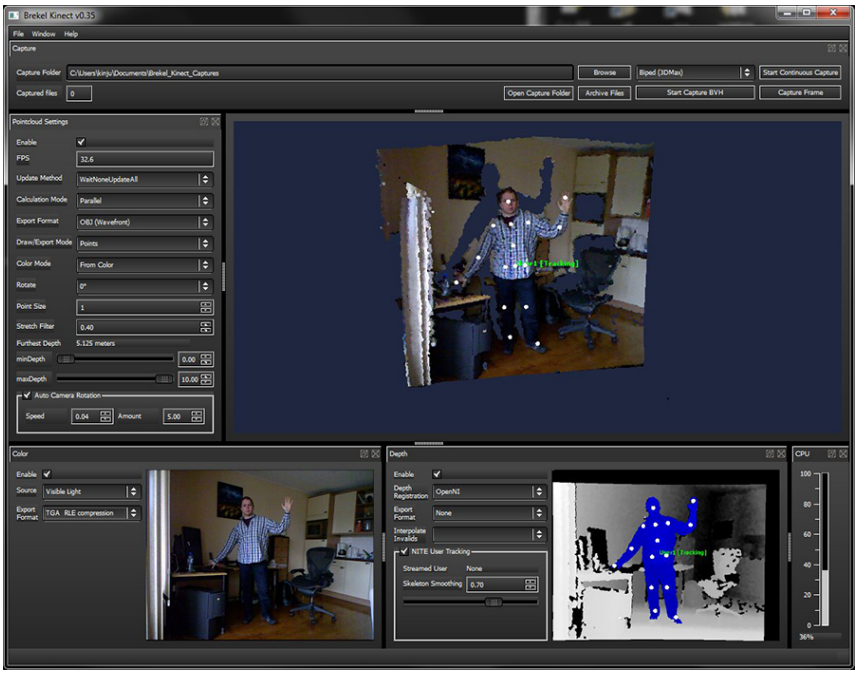


Figure 2.11: Brekel's main menu. We can observe the computed point cloud originated from the depth and color images captured [38].

Asus Xtion [40]. Its main feature is the real time skeletal tracking, while also providing additional add-ons for easy integration with other systems, and some sample files and tutorials for fast learning purposes.

What makes this system interesting are the plug-ins provided to use in conjunction with several 3D modeling softwares such as Autodesk Maya [41], Unity [42] or Blender (Section 2.4.1). By rigging a skeleton, and associating Kinect's skeleton joints in NI Mate to the 3D model joints in the 3D modeling software, it is possible to animate a 3D model in real time with real human movements, making it easier to give animations a reliable human-like behavior, as demonstrated in figure 2.12.

2.4 3D Simulation Engines

Because this project will have a virtual representation of the user, the need for a 3D simulation development tool is essential. This tool will have to be capable of creating a virtual stage, create or import a human-like rig, and also have a way to integrate with the Kinect camera data. In the following subsections, we will be discussing these aspects, considering the different available tools which integrate with the Kinect.

2.4.1 Blender

Blender is a free open-source 3D computer graphics software developed by Blender Foundation used for creating 3D assets to be used in several media such as animated films or

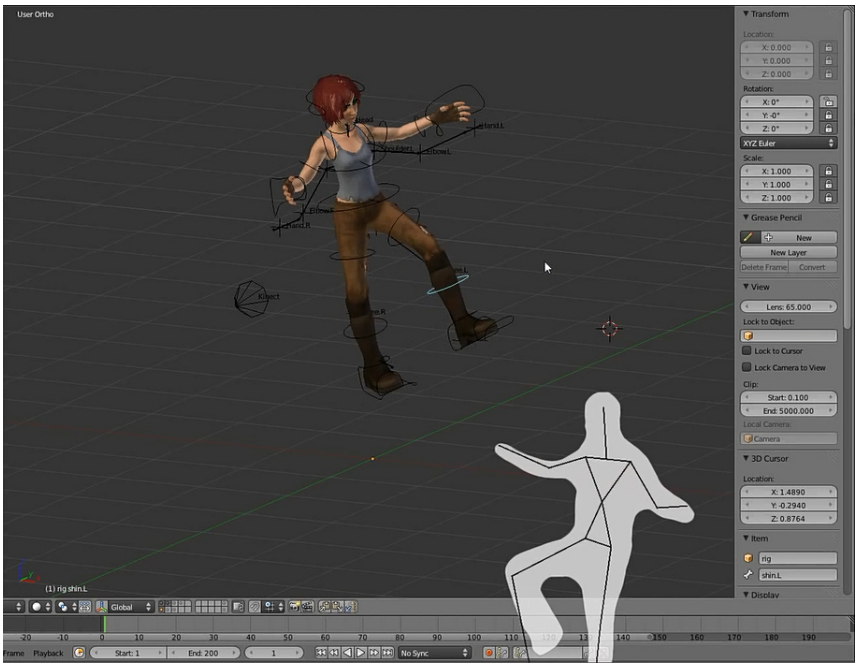


Figure 2.12: One of the developers at Delicode using NI Mate to animate an existing rig in Blender [40].

even video games [43]. It includes a built-in game engine that works differently than the conventional Blender engine.

When using the standard Blender engine, all images and animations are only rendered once, meaning they cannot be modified, but because the game engine renders in real time, all objects in the scene can be considered dynamic, simplifying the creation of interactive 3D applications or simulations. The game engine also has other features such as collision detection, dynamics engine and programmable logic. This logic uses Python programming language and, through Blender's API, allows scripting for tool creation and prototyping, game logic, and other custom tools.

Blender Foundation announces approximately every two years a new creative project to show case the tool's potential and encourage new and innovative applications done in Blender. These projects include short films such as "Elephants Dream" (2006) [44], "Big Buck Bunny" (2008) [45], "Sintel" (2010) [46], "Tears of Steel" (2012) [47], but also small video games like "Yo Frankie!" (2008) [48] and "Sintel The Game" (2010) [49].

There are already several experiments to incorporate real time motion capture directly into Blender, taking advantage of Blender's support for external add-ons. Some examples include László Sátori's work, who used Blender's engine to create projections that respond to a Hungarian dance group's movements captured by Kinect [50], and Technical University of Ostrava's work "Kinect & Blender" [51], in which a small application developed in C# uses the Kinect API to get the skeleton tracking points and transmit them over UDP[2] through the Python add-on in order to use them as an input for controlling

[2]User Datagram Protocol

an armature in Blender, as demonstrated in figure 2.13.

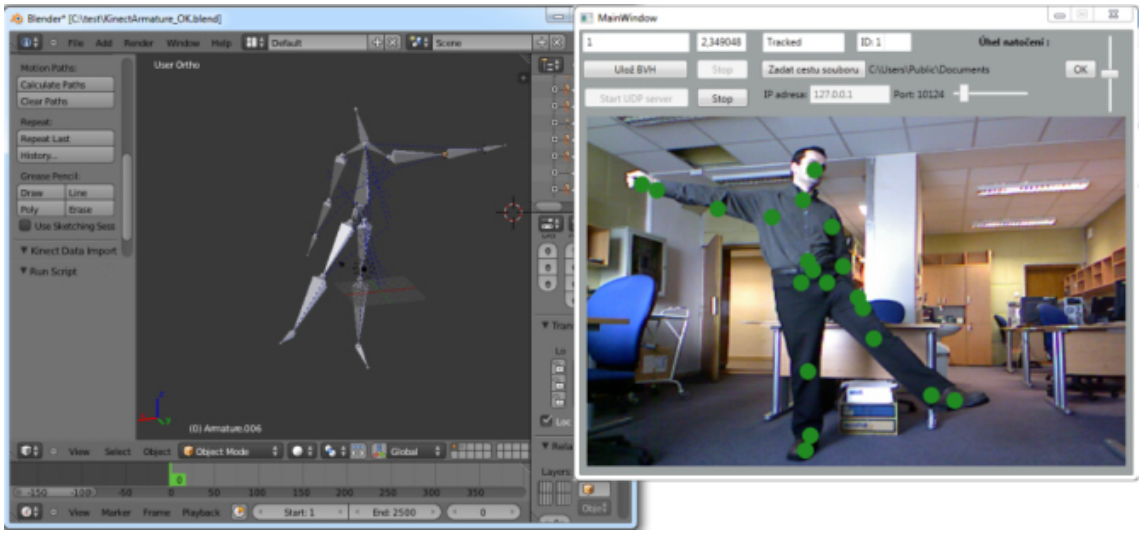


Figure 2.13: Kinect & Blender: demonstrating the skeleton tracking capabilities [51].

2.4.2 Unity

Similarly to Blender, Unity is a 3D computer graphics software which specializes in creating products for specific platforms [42]. Unity supports a wide range of platforms including operating systems, such as Windows, Linux and MacOS, most mobile systems, such as Windows Phone, Android and iPhone, but also home consoles such as the Sony Playstation, Nintendo Wii U and the Microsoft Xbox systems. It even was chosen as the default SDK for Nintendo's Wii U console, shipping the Unity Pro version free of cost along with each Wii U developer license [52]. It also allows for created content to run in most web browsers, through the Unity Web Player application.

Unity is currently in version 4.5.5, and it has two versions available: There is a free Unity version for noncommercial use, and also the Unity Pro, which license is available for a fee. The most notable differences between versions are that most CPU and GPU optimizations, video playback and streaming tools, and tools for rig creation are only available in the Pro version [53]. Using the free version, most users resort to using external rig creation tools to achieve this last one, but most of them require a money fee, which could be a setback for educational purpose development.

As for existing external tools which support Unity, the most commonly used is Zigfu [54]. This tool's development kit functions as a wrapper for external devices, more specifically, the Kinect camera. It was created as a way to develop Kinect based applications using JavaScript, and it eventually expanded to work with Unity. Using this, Unity built projects, for both desktop or browser use, can access and work with real-time Kinect captured data. The recently launched Kinect v2 SDK also advertises that it now includes API support for Unity Pro, but, at the time of writing this document, the projects and samples

using this feature were still in private testing, so details regarding this matter are subject to change [55].

2.5 Discussion

As demonstrated in section 2.1, there are several ways to capture a performance. For example, ChoreoGraphics [9] and the system developed by Chan *et al.* [8] use a traditional full body motion capture suit, but other capturing methods are starting to appear. Low cost motion capture cameras like the Kinect can be used to capture performances the same way that other more expensive systems do, at the cost of a lower fidelity of the movements tracked. Also, more and more sophisticated motion capture peripherals are being created specifically for motion capture purposes, like the Motioner used in project RAM [10].

Having a method to capture movement, a way to visualize this captured data is also needed. The system must give the user some sort of visual aid to help understand what is being captured by the cameras in real time. All systems mentioned in section 2.3 achieve this in some form, for it is an essential feature to have, by having a virtual avatar in a virtual stage.

In short, there are many different types of features already implemented in this field such as gesture and sound recognition, performance recording and matching, and many other real to virtual world interactions. The transition from virtual to real world, however, is not perfect. There is still the need to "humanize" all virtual movements, make them feel natural and not forced, while also guaranteeing the reliability when capturing all movements performed.

For this to happen, some features in this field, such as skeleton tracking and the 3D reconstruction methods, should be further improved for a more accurate model of the performer simulating real performances in a virtual stage. Our proposed tool gives the user a way to capture and represent a human digitally, both in his physical body and in the movements it can do, while also visualizing its interactions with non physical objects inside a virtual stage. It also opens the way for future features such as 2 person local, or remote, collaboration using multiple cameras, or even incorporate different captured objects, not specifically human-like, into a virtual stage.

Back in 2011, when the Kinect was first released, there were no other low cost motion capture systems available. This innovative piece of hardware certainly impacted the world of motion capture and animations, and it allowed for the creation of numerous new tools that were never thought of before. The Kinect SDK is constantly being updated, improving the existing features and integrating new tools, and with the most recent version having much more tool support, more and more tools will be created in the future.

Most of the researched tools mentioned in section 2.3 focus on skeleton tracking and

some way of motion capture, either by recording the skeleton animation or by integrating it directly into an external 3D modeling software. Some of them are also capable of, similarly to Kinect Fusion, digitally reconstruct a 3D model and allow exporting the reconstructed mesh, which can be later used in 3D modeling software such as Blender or Unity.

As for the researched options in 3D development software, mentioned in section 2.4, when applied to a project such as our own, Blender is probably the best choice. Its best trait is the accessible integration of external Python language plugins, and being Open Source, it could be ultimately modified to operate in any way desired. Because creating a rig with specific characteristics is an important step in our project, Unity is sadly not an immediate option, since its free version is incapable of doing so without external help. External rig creators could also be an option, but, out of the available free ones, many are automated, not creating rigs with as many bones as the Kinect's tracked skeleton. Because of this, Blender's Rigify plays a big part in the development of our prototype, since it can freely create rigs for a specific set of skeleton bones connected with a specific hierarchy, in our case, the one provided by the Kinect camera.

3

System Model and Features

In this chapter, we will start to delve into the fundamentals of building a prototype, taking in mind the notion of motion capturing, 3D model animation and 3D spatial representation, to be used in an artistic performance context. The goal of this chapter is to document in detail the system model and how each of the features is implemented, explaining all of the rationale behind each decision.

3.1 Data Acquisition and Representation

The core feature of this project is the capture and representation of data portraying body movements. This can be split into two phases, the first one being the *input*, represented by the chosen data type to be captured, the other being the *output*, represented in a virtual body on a virtual stage capable of acting according to the captured data. To achieve this, several tests were executed with the camera at our disposal, the Kinect for Xbox (Model 1414). The captured data is then to be injected in a virtual model, or a rig, inside a 3D game engine tool.

3.1.1 Data Acquisition

The first step executed for building the prototype was obviously getting the Kinect camera to display information, giving the user a visual cue of the data being captured by the camera. It was already decided that the code would be written in C++, as a direct result of using the openFrameworks toolkit. So, for an easier integration, it was also decided to use the Microsoft Visual Studio 2012 as the IDE[1] of choice. As such, openFrameworks

[1]Integrated Development Environment

would facilitate developing code capable of having a graphical representation, while also using the Kinect Common Bridge (KCB) add-on to ease the process of obtaining information from the Kinect camera.

The KCB add-on simply functions as a wrapper, providing a simpler way to fetch information from the Kinect camera. It also integrates several additional libraries to help manipulate most commonly used data streams. An example of this is the capability of obtaining not only the global rotation values, but also hierarchic values from each joint automatically. Because of this, it is logical to say that, at this level, the tool's limitations in capturing data are the same as the camera's.

Most openFrameworks add-ons provide one, or more, code samples for the user to understand how these additional tools can be used. This was also the case for the KCB add-on, as one of the skeleton tracking examples was studied for constructing the base line for our tool. This code sample provides all the necessary basics for capturing body movement, for it displays what the camera is capturing in a color and depth stream, side by side, and can detect and display users on screen as can be seen in figure 3.1.

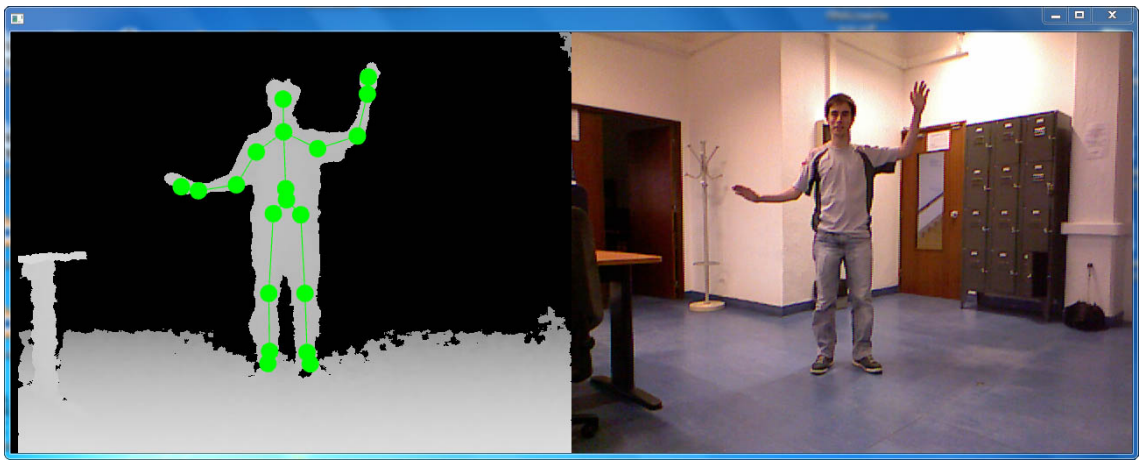
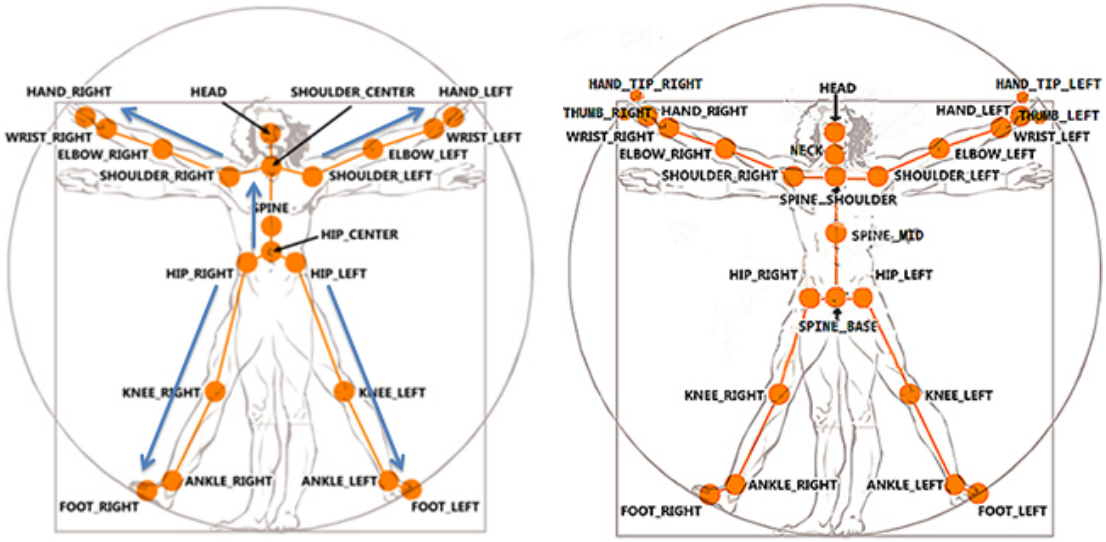


Figure 3.1: Kinect Common Bridge's Skeleton example graphical display.

After analyzing the information captured by the Kinect camera it was understood that each skeleton is represented by twenty nodes (figure 3.2(a)). The new Kinect version 2 has an improved skeleton tracking system, with the new skeleton having twenty-five nodes, with new nodes placed in the neck area and on the hands. These nodes bring more consistency to the whole skeleton's orientation, since it can now easily determine how the arm is rotated using the thumb's position, and also allows hand gesture recognition.

These skeleton bone nodes contain position and rotation values in relation to the camera's position, which can be later used when injecting data in the virtual rig. All nodes are connected in an hierarchic manner, *i.e.*, all geometric transformations executed on the parent node will propagate to all child nodes attached, where in our case the HIP_CENTER node is the root of the entire skeleton. A tree representation of the skeleton's hierarchy can be seen in figure 3.3. The position bone node values are stored in a simple *Vector4* structure, which is capable of storing four *float* variables, representing *W*,



(a) Kinect for Windows v1.8 (b) Kinect for Windows v2

Figure 3.2: Kinect skeleton node positions and orientations [56].

X, *Y* and *Z* values respectively. Only the *X*, *Y* and *Z* coordinates are really used, as they represent the distance in meters from the origin, at the center of the camera's field of view. The rotation bone node values, however, are stored in a *NUI_SKELETON_BONE_ORIENTATION* structure, which when represented in C++ code goes as follows:

Listing 3.1: Kinect SDK v1.8 *NUI_SKELETON_BONE_ORIENTATION* structure

```
typedef struct _NUI_SKELETON_BONE_ORIENTATION {
    NUI_SKELETON_POSITION_INDEX endJoint;
    NUI_SKELETON_POSITION_INDEX startJoint;
    NUI_SKELETON_BONE_ROTATION hierarchicalRotation;
    NUI_SKELETON_BONE_ROTATION absoluteRotation;
} NUI_SKELETON_BONE_ORIENTATION;
```

From the variable names and types it can be presumed that this structure gives access to the index position of the starting and ending joint, and also some kind of rotational values in an hierarchical or global orientation. With this, it is now known that it is possible to adjust what node data is sent, either hierarchical or absolute values, depending on how the rig interprets the injected data. This *NUI_SKELETON_BONE_ROTATION* structure contains exactly that, a rotational matrix of size 4 and a rotation quaternion. By using KCB, it is also possible to automatically convert this rotation data into Euler angles, for the sake of adaptability.

Euler angles work with three rotational axes (normally defined as *X*, *Y* and *Z*), in an hierarchical order, where any three-dimensional object can be freely rotated upon. This combination of successive rotations can be decomposed in three angles, each representing an axis-angle rotation.

Quaternions expand this concept, working with four dimensions, one real dimension

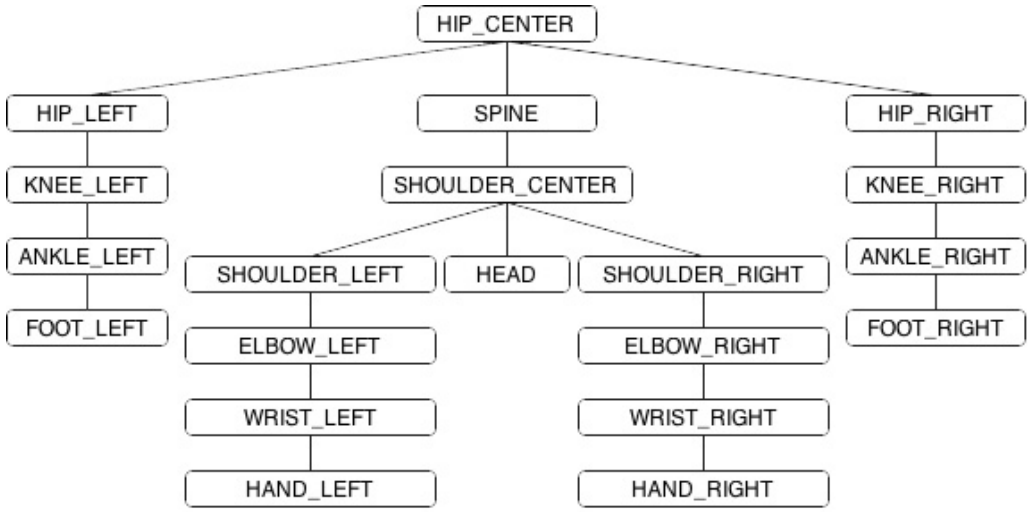


Figure 3.3: A tree representation of the Kinect v1.8 joints' hierarchy.

and three imaginary dimensions. Each of the imaginary dimensions has a unit value of $\sqrt{-1}$, while being mutually perpendicular to each other, and are defined as i, j and k. The following equation 3.1 represents a quaternion in axis-angle notation, where for any quaternion q, a is the angle of rotation and x, y and z are the vector representing the axis of rotation.

$$q = \cos(a/2) + i(x * \sin(a/2)) + j(y * \sin(a/2)) + k(z * \sin(a/2)) \tag{3.1}$$

However, for Kinect version 2, most of the APIs were modified, so the structure that stores the joint rotation values is now represented as follows:

Listing 3.2: Kinect SDK v2 *JointOrientation* structure

```
typedef struct _JointOrientation {
    JointType JointType;
    Vector4 Orientation;
} JointOrientation;
```

This structure only has two variables, where *JointType* is an integer enumerate value representing the index position of the current joint, and *Orientation* is a rotation quaternion representing the absolute joint rotation values. Since the hierarchical rotation values are not represented, in order to convert a global rotation G of joint b into a hierarchic rotation H, we need to get the global rotation G of the parent joint p, and multiply its inverse as follows:

$$H^{(b)} = G^{(p)-1} * G^{(b)} \tag{3.2}$$

3.1.2 Data Representation

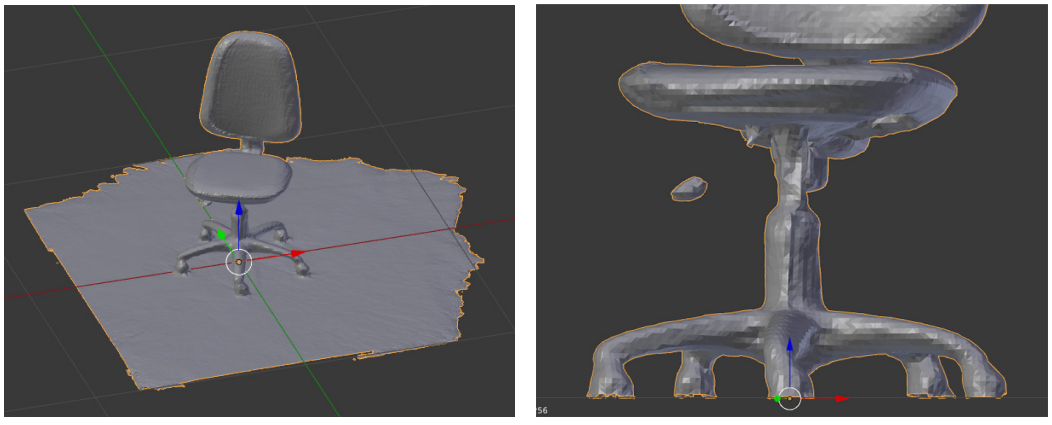
Having found a way to acquire motion capture data, there is a need to somehow interpret and represent that information in a visual manner. Using a virtual avatar it is possible to

emulate those captured movements in a 3D environment, so the next questions that need answering are "What is our environment?" and "How can we create an avatar?". Our answer to both questions is found using 3D simulation engines.

As mentioned in section 2.4.1, the Rigify add-on can create humanoid rigs using a simple mesh and an armature that resembles human bones in structure. From this, the main idea is to create a mesh, to represent the avatar body, and create an armature, to represent the bones, so the animation rig can be created. Because we are using the Kinect camera, and the only known skeleton model created by the Kinect is the one mentioned in figure 3.2(a), it is only logical to create an armature that has the same amount of bones as this one.

In a way to try to immerse the user in a more realistic experience, the idea of reconstructing a 3D avatar mesh of the performer came to life. With the Kinect camera at our disposal, this process is simplified, since the official SDK includes the Kinect Fusion tool (section 2.3.1), which allows to create an OBJ file containing a 3D representation of the captured user. Some experiments were carried on to test the potential of this tool. The first attempts were on inanimate objects, with later attempts being on human bodies. For the inanimate object experiments, an office chair was chosen for not being too small of an object and having some intricate parts that would be interesting to reconstruct.

The inanimate object reconstruction experiences presented several things to consider. One thing in particular was obvious, as the floor was always getting captured and reconstructed along with the object, as demonstrated in figure 3.4(a). This happens because it is required that the target stays stationary during the reconstructing process and, because gravity affects all objects the same way, they usually stay on an elevated surface, or on the floor. As a result, there are surfaces which are impossible to reconstruct, where the camera does not reach, such as the bottom part of the chair wheels, or the sole of the shoes of human bodies, which lie on the floor.

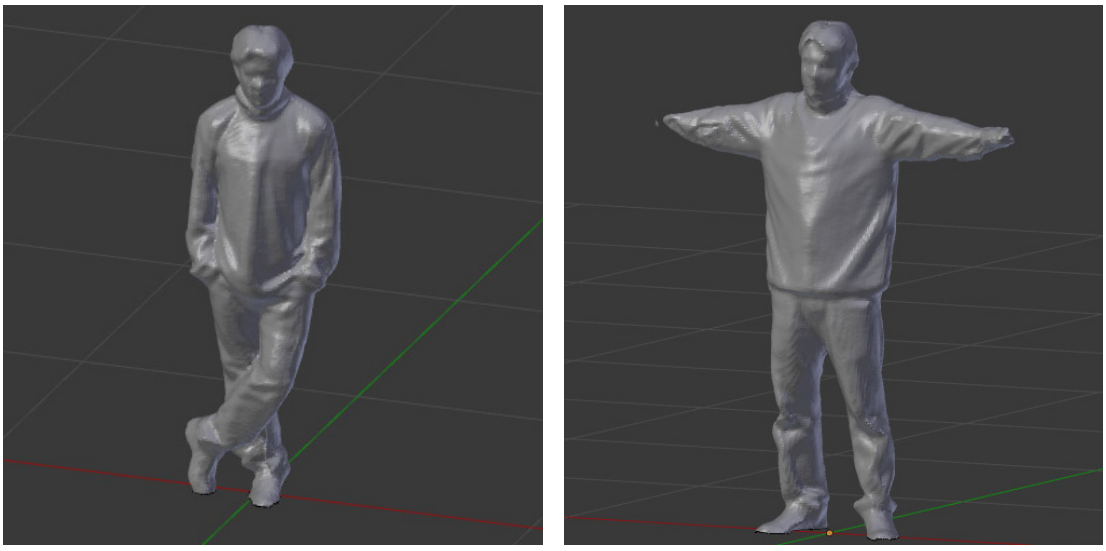


(a) Floor capture (b) Chair handle reconstruction

Figure 3.4: Results of inanimate object reconstruction: the floor is captured along with chair (a), a small metal bar linking the handle and the main chair pillar is missing (b) and a final successful chair reconstruction (c).

This implies that, later in the project, since we are going to rig the reconstructed human bodies for animation purposes, the floor will have to be edited out of the created OBJ file, for it does not belong to the human body. The chair lift handle also was not accurately recreated, for it was missing a small metal connector with the main chair pillar, seen in figure 3.4(b). This detection mishap probably happened due to its small volume, being too small for the Kinect Fusion to consider it as part of the object being reconstructed.

Attempts on human bodies had similar problems to the inanimate object attempts, as the floor was also being reconstructed. The clothes the target is wearing are also an aspect to consider. The Kinect Fusion algorithm does not differentiate between body and clothes, so it stitches them together in the same mesh. Later, when we animate these bodies, this factor can make the model seem fatter than it is, or worse, by having the limbs too close to each other they can be stitched together, not accurately representing how the human body limbs are connected. One such example is having the arms staying too close to the torso, stitching them up together in the mesh, as seen in figure 3.5(a).



(a) Casual pose reconstruction (b) Partial T-pose reconstruction

Figure 3.5: Results of human body reconstruction: full scale body reconstruction in a casual pose (a) and T-pose attempt ends in a partial reconstruction of the arms due to the camera distance to the target (b).

Each of the reconstruction attempts took about 3 minutes to completely create the mesh without any cracks or missing parts. However, this is not optimal for human body reconstruction. The optimal pose, and also more convenient, for accurately reconstructing human bodies for animation purposes, without having body parts incorrectly stitched, is having the model standing straight with arms wide open, making a T-pose, exemplified in figure 3.5(b). This requires the target to be in that same pose for that amount of time making it too tiresome. The resulting meshes also are not optimized, having a high number of triangles defining the mesh. This can possibly be harmful later on, when applying Blender Game Engine physics to the whole body, forcing it to make

much more calculations per second. Due to these complications, it was decided to use a simple free 3D model available online as our human mesh.

After loading the OBJ file containing the desired model into Blender, and knowing the skeleton composition coming from the Kinect camera, it is possible to set up an armature that will support the triangle mesh model. However, Rigify does not automatically perform this operation, meaning that the armature has to be previously aligned with the mesh for it to detect what skeleton bone belongs to which body part of the model. Only then can the armature be attached to the mesh, creating another set of armature pose modifiers with appropriate constraints, having the bone joint position and scale locked, but being able to change rotational values. These constraints are also configured to be contextualized, depending on the skeleton bone joint. A quick example of this is having the knee joint unable to bend forwards, even if the rotational values received define it as such.

Having an armature rig, we can finally decide what type of rotational values are to be injected into the joints. After some research and deliberation between using Euler angles or quaternions, we came across a common three-dimensional phenomenon which can occur when using Euler angles. The Gimbal Lock problem happens when two rotational axes of an object are pointing in the same direction, as demonstrated in figure 3.6 [57]. These two axes, who are overlapping, now work as just one, losing one degree of control over the rotation of a three-dimensional object. This problem is inevitable, for it happens in all Euler angle hierarchical combinations. With quaternions, this problem is non existent, making it the obvious choice for our rotational values.

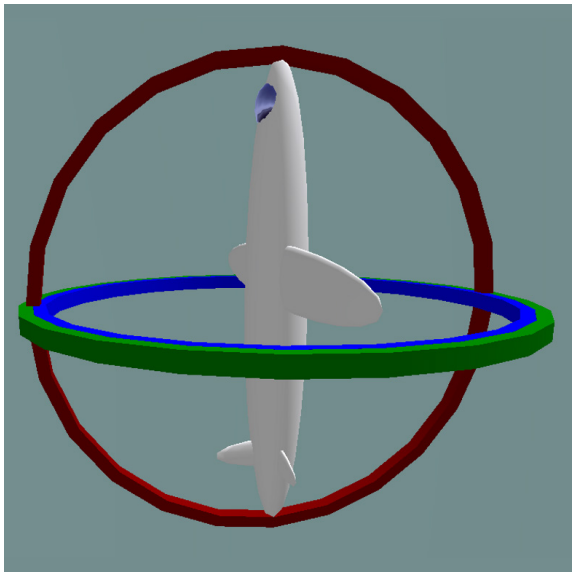


Figure 3.6: Gimbal Lock example, two axes are pointing in exactly the same direction [57].

3.1.3 Initial Data Simulation

At this point, we already have the two elements necessary to create a prototype, for we have the skeleton rotation nodes from the Kinect camera, and a way to create 3D body models to animate in Blender's environment. All that is left is to link the two. For that, a simple UDP[2] connection is established, in where the C++ code which fetches the Kinect information sends it through the connection as information gets captured, as long as there is a skeleton detected. In Blender, a simple Python script is loaded in the Game Logic layer, which receives the information from the established connection. It then redirects the given values to the correspondent armature pose modifier generated through Rigify, which it would theoretically make the 3D model move accordingly. Unfortunately, because of the order Blender executes the different layers, the Game Logic layer comes before the Scene layer, which makes the Python script execute first than the armature constraints, overriding them.

By running the Blender Game Engine, we come across our first big hurdle. It seems that even after receiving the values correctly, the body would not move. This was caused by a Blender Game Engine standard, in where all objects are rendered only once, when they are first created. A quick workaround was devised, where we would force only the body object to re-render every frame. This has a higher, but necessary, strain on the tool's performance, for the entire model triangle mesh has to be reloaded as many times as the information is received, which is expected to be 30 frames-per-second. If we were to use a 3D model generated with Kinect Fusion, which has a higher triangle count, and by adding more objects to the scene, there is the possibility that the tool's performance quality would decrease.

The second, and more relevant, hurdle came from the Kinect camera itself. When testing the different node rotations, one by one, it was becoming apparent that some of the nodes did not seem to react properly to the information received. As we are using an hierarchical approach, when the root node is not working as intended, the adjacent nodes are going to also not work properly, which is problematic.

In an attempt to circumvent the incorrect rotation problem, a different approach was considered. Instead of using rotation values received directly from the Kinect, we tried to inject a full BVH animation file into our tool. These BVH files contain information referring to the skeleton hierarchical composition, node starting locations and offsets between them, while also containing rotational values that affects each of nodes for each frame.

Developed alongside with the prototype, MoCapPlay is a BVH file visualizer, *i.e.*, it loads BVH animation files and displays its skeletal representation in an OpenGL[3] environment, as demonstrated in figure 3.7. The program was then modified to convert the raw Euler rotation values from the BVH file into quaternions, and send them through an UDP connection for our Blender Python script to receive, making the 3D model move in

[2]User Datagram Protocol
[3]`http://www.opengl.org/`

unison with the MoCapPlay skeleton.

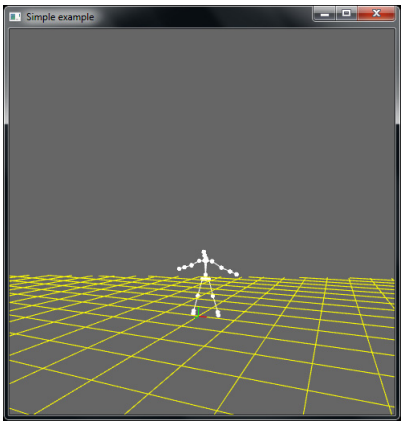
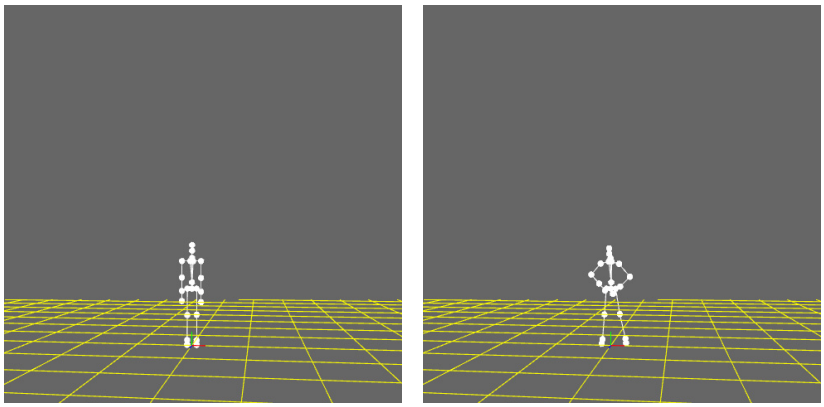


Figure 3.7: MoCapPlay's graphical environment.

The most distinguished aspect from the previous approach is the skeleton composition, for it is not guaranteed that the skeletons included in the BVH files have the same hierarchy, or even the same amount of bones, as the Kinect skeleton. The most common way to compare skeleton compositions is by analyzing their rest poses. A skeleton is in rest pose when every bone has no rotation in their local space, as demonstrated in figures 3.8(a) and 3.8(b). This means that the number of skeleton bones and their positioning are what can diverge between skeletons, with the most common skeleton variations being having the arms wide open (T-pose) or pointing down (Relaxed pose).



(a) Rest pose (b) First frame

Figure 3.8: Displaying a skeleton's rest pose and its first frame of animation, taken from the *jumpkick.bvh* file, in MoCapPlay.

The main sample BVH file used for testing purposes, appropriately named *jumpkick.bvh*, included an identical skeleton structure to the Kinect skeleton, and its animation had a dancer doing some spins, ending up its performance with a jump kick, as illustrated in figure 3.9. This file, and several others used for testing, were downloaded from the Carnegie Mellon University Motion Capture Database [58].

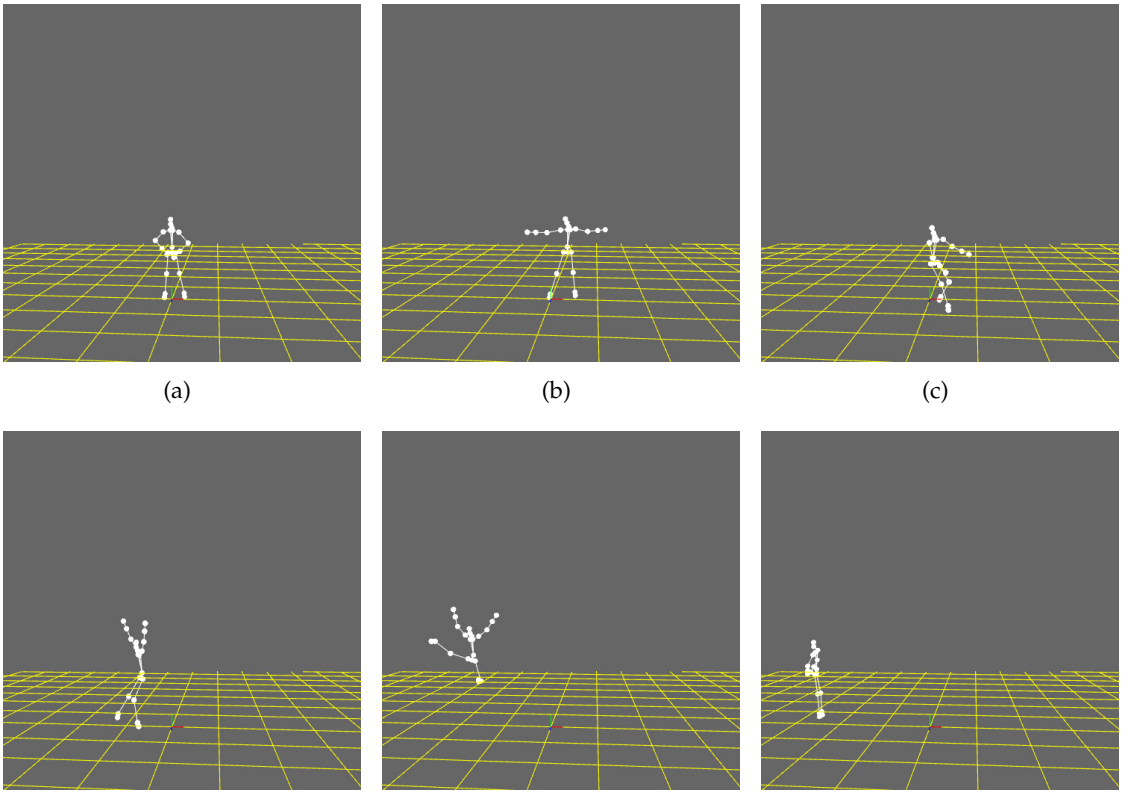


Figure 3.9: Still frame shots of the *jumpkick.bvh* animation file.

The first tests had some interesting results. Firstly, it appeared that the chest node would never rotate properly, even when injecting the rotation values manually in Blender's properties editor. This was caused by a Rigify fault where, when creating the armature pose modifiers of the rig, the hierarchical composition of the armature is ultimately changed. When this happens, the armature hierarchy is then divided into two totally independent sections: an upper section, which contains all bones from the chest up, with its root being the $SHOULDER\_CENTER$ node; and the lower section, which is everything below the chest, having the HIP_CENTER node as its root, as illustrated in figure 3.10. This results in the upper section root node not being altered by any rotation involving only the lower section root node, and the other way around as well. One such example is demonstrated in figure 3.11, where the chest node does not properly rotate, changing the way the model should be posing.

To resolve this problem, it was necessary for the $SHOULDER\_CENTER$ node to accumulate all the rotational values propagated through the HIP_CENTER, $SPINE$ and $SHOULDER\_CENTER$ nodes, in that specific order. So, we took advantage of the quaternion multiplicative and associative properties, and performed a simple multiplication, exemplified in equation 3.3. Assuming $Q^{(c)}$ is the quaternion obtained from the $SHOULDER\_CENTER$ node, $Q^{(s)}$ is the quaternion obtained from the $SPINE$ node, and $Q^{(h)}$ is the quaternion obtained from the HIP_CENTER node, representing the

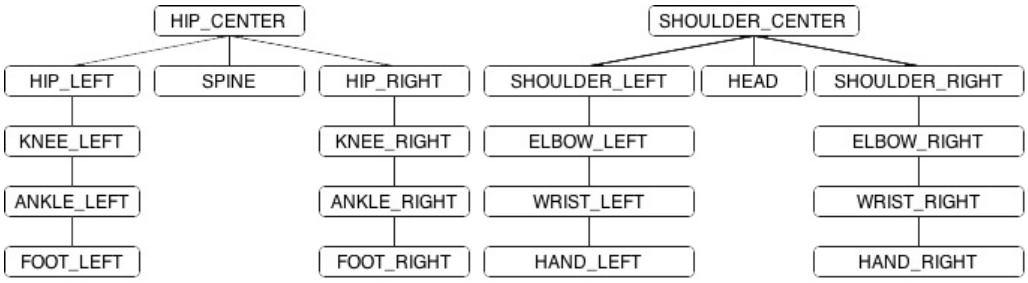


Figure 3.10: A tree representation of the Rigify created rig joints' hierarchy.

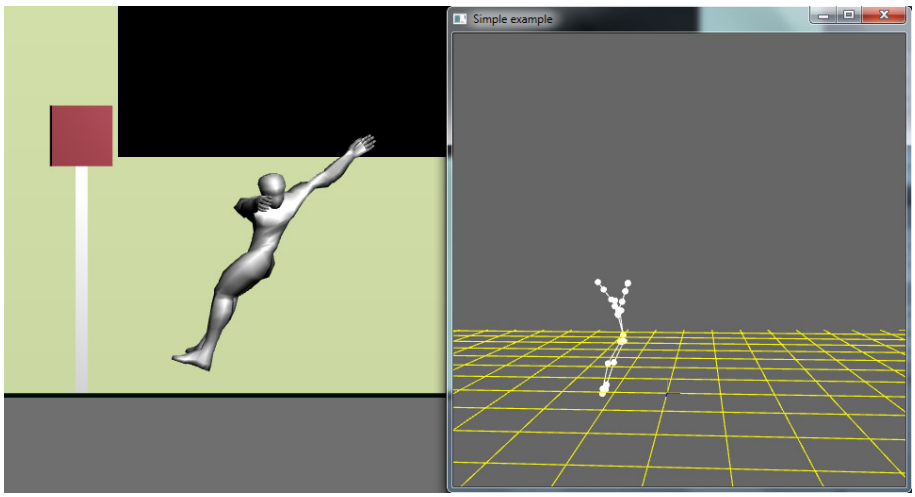


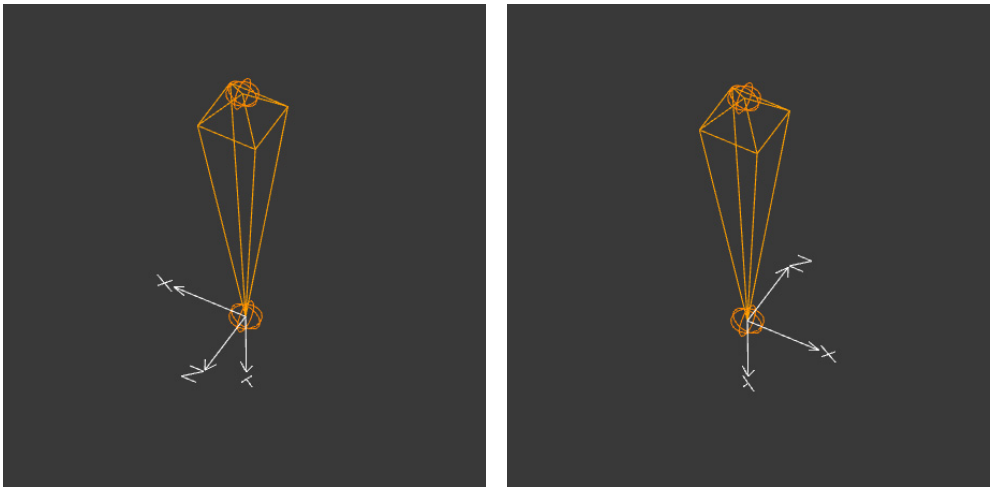
Figure 3.11: The 3D model skeleton does not match the BVH skeleton because of the faulty chest node values.

complete hierarchical structure for quaternion $Q'^{(c)}$:

$$Q'^{(c)} = Q^{(c)} * Q^{(s)} * Q^{(h)} \tag{3.3}$$

Secondly, while some of the bones rotated as intended, some seemed to be facing the wrong direction. This happened because the joint from the BVH file and the corresponding joint in our armature functioned in different coordinate systems, having different orientations. An example of the difference in orientations is represented with figures 3.12(a) and 3.12(b). These armature bones look similar, but their orientations axes are placed differently, which changes the way these bones react to rotations. If we were to apply a rotation of 90 degrees in the X axis on both bones, the they would rotate in different directions. If bone (a) was a bone from the BVH file skeleton and bone (b) was a bone from our armature rig, we would have a faulty representation of the same rotation values.

When faced by this problem, the most common solution is to express one of the objects in the same coordinate system as the other. [59]. Let us assume that $M_{i \leftarrow j}$ is the rotation that converts the position of a point in a coordinate system j into its position in coordinate system i. If $P^{(i)}$ is the position of a point in coordinate system i, and $P^{(j)}$ is the position of the same point in coordinate system j, then we have $P^{(i)} = M_{i \leftarrow j} * P^{(j)}$.



(a) (b)

Figure 3.12: Representation of armature bones with different orientations.

Then, if $Q^{(j)}$ is a rotation in coordinate system j, we want to find a rotation $Q^{(i)}$ in coordinate system i that when applied to a $P^{(i)}$ would produce the same results as if $Q^{(j)}$ were applied to $P^{(j)}$. This relation between points is represented by $Q^{(i)} * P^{(i)} = M_{i \leftarrow j} * Q^{(j)} * P^{(j)}$. Substituting $P^{(i)} = M_{i \leftarrow j} * P^{(j)}$, this expression becomes $Q^{(i)} * M_{i \leftarrow j} * P^{(j)} = M_{i \leftarrow j} * Q^{(j)} * P^{(j)}$. By simplifying this equation, we are left with $Q^{(i)} = M_{i \leftarrow j} * Q^{(j)} * M_{i \leftarrow j}^{-1}$.

This same principle can be applied to our problem, forcing the local orientation of the current node to match the orientation of the corresponding joint in the rig armature when in rest pose. Suppose a quaternion $Q^{(f)}$, with the coordinate system originated from the BVH file data, and assuming $M_{r \leftarrow f}$ as the alignment rotation that converts the representation of coordinate system f, representing the file joints data, into its representation in coordinate system r, representing the armature rig joints. Equation 3.4 shows how the final expression for correcting the rotation values, for any quaternion $Q^{(r)}$, with the coordinate system from the corresponding armature rig joint.

$$Q^{(r)} = M_{r \leftarrow f} * Q^{(f)} * M_{r \leftarrow f}^{-1} \tag{3.4}$$

By using these corrections, all joint rotations coming from the BVH file are thus correctly displayed on both the MoCapPlay display and our 3D model in our virtual stage. The final alignment quaternions defined for all nodes in the *jumpkick.bvh* file are disclosed in table 3.1. These values were obtained by comparing each BVH bone to each rig bone, one by one, and determining which rotation would have to happen so that they would have the same coordinate system, giving an close approximation of the movements performed.

Table 3.1: Alignment quaternions used for *jumpkick.bvh* file. All joint names are the Kinect's equivalent joints, already referenced in figure 3.2(a).

Kinect joint equivalent	**w**	**x**	**y**	**z**
SHOULDER_LEFT	0.500	0.500	-0.500	-0.500
ELBOW_LEFT	0.000	1.000	0.000	0.000
WRIST_LEFT	0.000	1.000	0.000	0.000
SHOULDER_RIGHT	0.500	0.500	0.500	0.500
ELBOW_RIGHT	0.000	1.000	0.000	0.000
WRIST_RIGHT	0.000	1.000	0.000	0.000
KNEE_LEFT	0.000	1.000	0.000	0.000
ANKLE_LEFT	0.000	1.000	0.000	0.000
KNEE_RIGHT	0.000	1.000	0.000	0.000
ANKLE_RIGHT	0.000	1.000	0.000	0.000

3.2 Simulation Environment

As previously mentioned, the simulation environment chosen for this project to run on is the Blender Game Engine. In this section we explore the wide range of possibilities provided by this engine, and how we can apply our ideas into it.

For this prototype a simple stage was created, seen in figure 3.13, where our 3D model is able to move around and interact with objects. This performing stage is composed by a solid floor and walls and a pole with a cube on top, which we used to do some tests for the physics engine. All these objects are configured so, when running the physics engine, they have different behaviors, as a sample of what can be done with the physics engine. The initial test consists of running the *jumpkick.bvh* file, having the 3D model knock down the box placed on top of the pole when performing its animation.

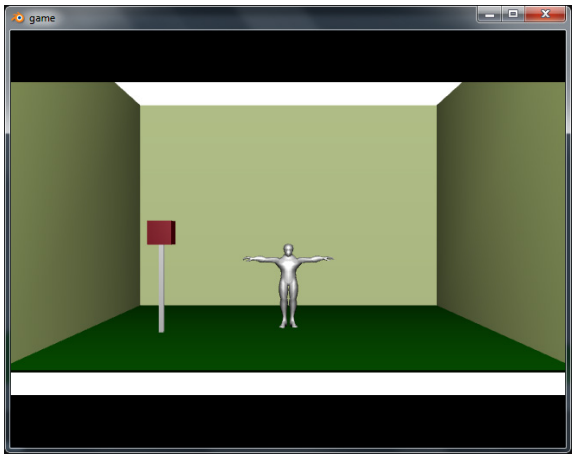


Figure 3.13: Prototype of the virtual performing stage created in Blender.

Already mentioned in section 2.4.1, Blender has its own built-in Game Engine that allows creating 3D applications or simulations. The engine allows simulating content within Blender itself, but also allows exporting a binary run-time to run in Windows,

Linux and MacOS. This Game Engine has a specific method in the way that it renders the scenes. Instead of rendering the scene only once, forcing objects to stay static, the rendering process works in real-time, allowing modifications in the objects belonging to the scene. This property gives way to many different possible interactions, such as integrating a real-time physics engine or including dynamic textures in objects.

The Blender Game Engine uses a system of "Logic Bricks" to manipulate the objects running in the engine. These "Bricks" are divided into three categories: (1) sensors, (2) controllers and (3) actuators.

Sensors are the starting point of any action, functioning as a trigger for all connected controllers. These send a positive pulse whenever a trigger event occurs, such as a keyboard press, a timer going out or even when a collision between objects is detected. A negative pulse is sent whenever the sensor is deactivated. The controllers are the logic barrier between the sensors and the actuators, which specify the conditions for which they operate. These conditions can be boolean operations between sensor pulses, user written boolean expressions or Python scripts. The actuators are the ending stage, as they receive the signal from the controllers to finally execute an action on an object. The set of possible actions includes editing object properties, constraints and position and orientation, sending signals to objects, add objects to the scene, play sounds and more.

For simulating the physics engine, Blender uses an external library named Bullet [60]. This library features several components such as 3D object collision detection, soft body dynamics, and rigid body dynamics. Using this it is possible to define how objects interact with each other, how they should react on collision, and control their mass and weight to appropriately simulate gravity.

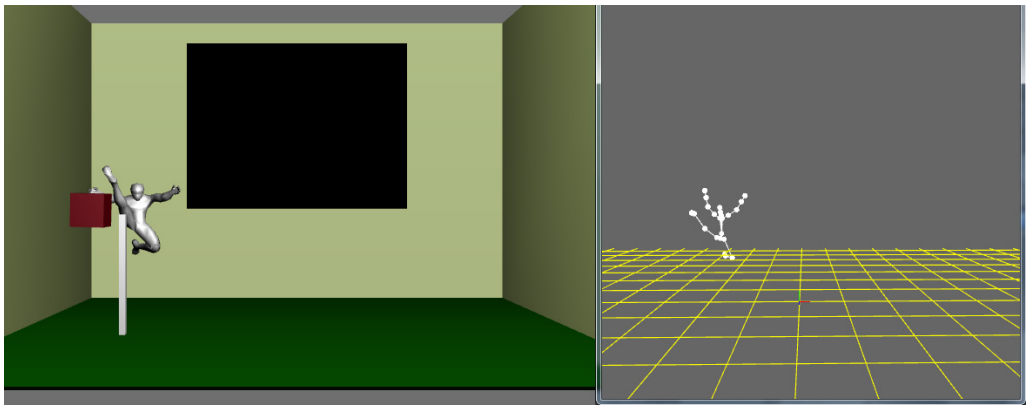


Figure 3.14: Using MoCapPlay to animate the 3D model to knock down the red box.

A good example of how we use this engine is with the *jumpkick.bvh* animation file. We define our body mesh as a dynamic object, that does not need any additional rotation applied to it when colliding with an object, since we are updating the mesh in real-time, not requiring that additional computation. Meanwhile, the box is a simple rigid body that simply reacts with the gravity and when collided with other objects. Then, when executing the animation, the performer will collide with the box, forcing it to move from

the pole and fall down to the floor, as shown in figure 3.14.

To further expand the possibilities for our tool, the option to play a video was added into the tool. This video player can be used for different purposes, such as projecting a video that would contribute to the performance, acting as an expansion of the stage itself. It could also be used as a training aid, possibly projecting a dance routine that the user would emulate, hopefully improving their performance.

Blender allows for special types of textures to be used on objects, and so, one of such textures was used on one of the virtual stage's objects. Our virtual stage's back wall holds a 2D panel object with a special dynamic texture type, which will gather data from any chosen image or video file format covered by the FFmpeg codec framework, and display it accordingly [61]. As so, the user is able to choose any available AVI[4] file and play it on a loop, as demonstrated in figure 3.15.

Figure 3.15: Video being displayed on the wall in the back of the stage while the performer dances.

3.3 Discussion

While defining the tool's main features, we came across various obstacles. Setting up the Kinect camera was not a trivial process, all because the Kinect support for openFrameworks is fairly limited. Luckily, the Kinect Common Bridge add-on facilitates this integration with actual code samples and documentation. From there on, getting the data wanted is very straightforward.

Analyzing the skeletal composition of the Skeleton Tracking data collected with the Kinect was very interesting. The internal algorithm is able to detect bodies on camera without any problems, having only some limitations related to the camera's depth perception limits and when inferring not visible nodes' positions and rotations. One example of this is when the body is sideways, because most body limbs cannot be correctly

[4] Audio Video Interleave

discerned from all the nodes clumping all together. The camera is also not able to distinguish when the body has its back turned to it, creating a skeleton as if it was facing forwards.

Several attempts were made using mesh reconstructing software to generate real life objects in a virtual environment. We can observe that the reconstructed chair object can be considered good in quality, but not perfect, as demonstrated by the missing handle, and the ground getting captured as well. The attempts on reconstructing a real person showed that this method is not optimal, for several reasons. The time it takes to accurately capture a watertight mesh, with no cracks, all this while the model is in a T-pose, can become very tiring. Also, the high triangle count in the reconstructed meshes deems this method not usable in this context, unless for the cost of a higher processing strain when animating the rig. We could reduce this triangle count in a processing step after the mesh acquisition, but to keep things simple, a simple free 3D model available online is used in the default rig for all further steps.

The rigging process was probably the most complex part of the development process. The way the quaternion rotations function is not very intuitive, requiring thorough analysis on each and every armature bone. Building MoCapPlay was a way to fix the rotation problems, and possibly some more underlying problems. We then discovered two Rigify armature flaws, one where certain armature bones needed some alignments in their coordinate systems to make them rotate in the right direction, and the most important of them all, the overall hierarchical structure of the Rigify armature. Even after fixing this, the seemingly wrong values received from the Kinect were still a problem, which will be discussed in section 4.2.

Learning to work with Blender was also an interesting working experience, especially the physics engine. The integration with the Python scripts for the data capture was fairly simple, with the only really important complication being the fact that all modifications executed through these scripts will override all defined properties performed through the graphical editor, such as the armature constraints example. Other than that, it worked just as expected, and it certainly was satisfactory when the 3D model knocked down the red box when running the *jumpkick.bvh* file in MoCapPlay for the first time. The dynamic video texture, however, feels like it needs to give the user access to more control options outside of using Python scripts to control it. As it stands, it can only play, pause, stop and resume playing the loaded video. All of these actions could only be triggered through specific sensors or Python scripts.

4

System Development

For this chapter, we will be presenting the final product originated from the system's development cycle. The following sections will present the project's overall architecture and implementation details, that were not mentioned before in this document. More focus will be given to the various skeletal compositions that appear throughout this project, namely the skeleton armature received from the Kinect and the armature created in Blender with Rigify, and how they interact with each other.

4.1 Architecture

From figure 1.1, representing the initial plan for this project's development, we detailed both initial system blocks, the Motion Capture System and 3D Representation System, as seen in figure 4.1. It is still divided into two major sections, with the intention to represent both the performance and also the stage in a final simulation.

For that, the Motion Capture System focuses on using a camera device to capture a body's skeleton data which, when interpreted, can be sent in real time to the Motion Capture Representation module, or stored in an animation file database for later use. The 3D Representation System uses input from a device that can create 3D meshes from real life objects in order to represent the performer's body. This is used for constructing the 3D Avatar Model which will emulate the performance from the Motion Capture System module. As discussed in section 3.1.2, the current methods available for creating this mesh do not give an optimal final result for the final simulation, so the 3D avatar will be the same for every performance. The system's other focus relies on representing the 3D Stage Model where the virtual performer will stand. This stage will include the 3D avatar, the stage objects themselves, some kind of video integration, all being subject to

the simulation environment's physics engine. The Simulation block is where the two systems are combined, taking all information passed out through them, such as the 3D stage and its properties, the performer's movements and the 3D avatar, emulating them using the internal engine in real-time.

The machine that was used for this prototype had an Intel Core i7-3630QM processor running at 2.40 GHz, 8 GB of RAM, an Intel HD Graphics 4000 graphics card, with Microsoft Windows 7 Professional as its operative system.

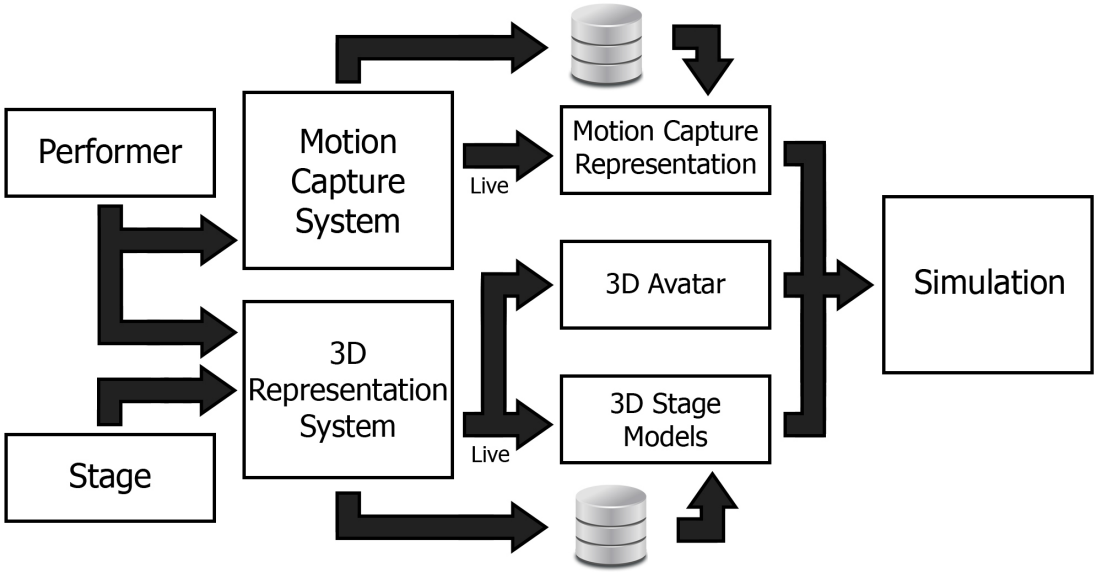


Figure 4.1: Diagram of the project's final prototype architecture.

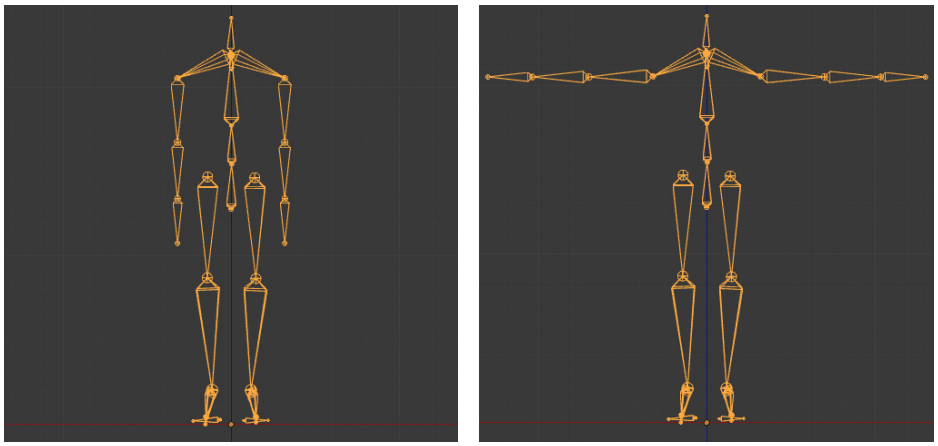
4.2 Data Input System

After successfully animating our 3D rig through MoCapPlay, by means of interpreting a BVH animation file, we decided to return to the apparently erroneous Kinect skeleton armature values issue mentioned back in section 3.1.3. The rig used with the BVH animation was created from an armature composition, inspired by an armature obtained through the Brekel Kinect application, already mentioned in section 2.3.3, which is able to take the Kinect directives and save BVH animation files. This BVH file contained the main bone hierarchy that the Kinect expects when sending the position and rotation values.

However, since we are required to rely on Rigify to create our rig, we could only construct an armature using the representative limb armature templates included in Rigify. With this restriction, we tried to build an armature as close as possible to the one obtained from the Brekel application. When finalized, the only difference in structure between this armature and the Kinect captured skeleton is the lack of the left and right hip bones. Even so, these small bones only propagate small rotations to both legs, which are taken

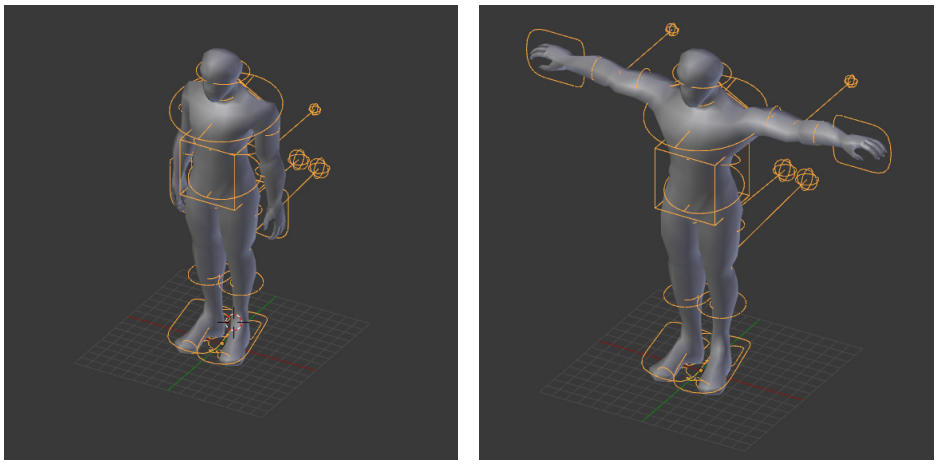
into account for hierarchical and mathematical purposes, but are not significant enough to display changes when animating the 3D model.

The Rigify created armature is then parented to a 3D model, automatically recognizing and assigning an armature part to a body limb. As mentioned earlier, instead of a reconstructed mesh a sample 3D model was used. This model had much less polygons than a reconstructed mesh created from Kinect Fusion, which will mean less computations when running the Blender Game Engine's physics engine. The original mesh was in a relaxed pose, and was then modified for us to have both a relaxed pose version and a separate T-pose version. The same process was executed with the armatures, modifying the original armature to match each model accordingly. Each of the armature compositions can be seen in figures 4.2(a) and 4.2(b), while the final rig result of the models can be seen in figures 4.3(a) and 4.3(b).



(a) Relaxed pose (b) T-pose

Figure 4.2: Final armatures used for creating a rig with Rigify.



(a) Relaxed pose (b) T-pose

Figure 4.3: Final rigs created with their respective models and armatures.

As we converted the values acquired from the Kinect camera, from matrix into quaternions, a pattern started to emerge. Because it is known that the skeleton captured from the Kinect has an hierarchical structure, it would make sense that when posing in a rest pose, the value from any quaternion Q would also be close to identity, or $Q = (1, 0, 0, 0)$. However, this was not happening.

Instead, some of the skeleton joints were presenting different results. Particularly the hips, both shoulders, both upper arms and both thigh nodes all displayed different neutral stance rotations, with all of them being different between themselves as well. By applying the same method used for correcting the BVH file skeleton orientation, the Kinect joint rotation values can be equally corrected, as demonstrated in equation 3.4.

Since the rest pose for our Blender armature is in fact the T-pose, we had to account for the additional pose rotations as well. This mean that, when in T-pose, each one of these rotations has to be nullified, *i.e.*, transformed into a identity quaternion. Knowing this, in order to get the correct rotations a few tests were conducted. Let us assume a quaternion $R^{(k)}$, resulted from a skeleton joint captured by the Kinect camera while performing a T-pose. Also assume its coordinate system k is the same as coordinate system r from its corresponding rig joint, not needing any alignment. Because it is known that any quaternion Q multiplied by its inverse Q^{-1} results in an identity quaternion, if we store this quaternion R value for later use, we can nullify any following rotation quaternion $K^{(k)}$ captured by the Kinect. With this, the expression that corrects the excessive rotations between the Kinect captured joints and the Rigify created rig joints is represented by $K^{(r)} = K^{(k)} * R^{(k)-1}$. All correcting quaternion rotations applied to nodes are displayed in detail in table 4.1.

As already mentioned in section 3.1.3, the Rigify chest node problem determined that this specific node required value dependencies passed on through the bone hierarchy. We can expand this characteristic for any quaternion, *i.e.*, if a quaternion Q_1 has a dependency value on another quaternion Q_2, the interaction between both quaternions is represented by the expression $Q_1 = Q_1 * Q_2$.

By combining these three proprieties, for any quaternion $Q^{(k)}$ captured from the Kinect, having its corresponding alignment quaternion A, rest pose correction quaternion R and dependency quaternion D. Equation 4.1 represents how these quaternions are combined, resulting in the final joint quaternion $Q^{(r)}$ that will be injected in our rig.

$$Q^{(r)} = A * Q^{(k)} * D * R^{-1} * A^{-1} \tag{4.1}$$

4.3 Final Data Simulation

After defining how the rigged model created in Blender works and determining how the information is sent from the Kinect, there is still a need to find a way to standardize the information connecting the two ends. It is not safe to rely on always having the same type

Table 4.1: Kinect to T-pose rotation conversion table, in quaternions. Figure 3.2(a) displays where each node is placed, according to the node names.

Kinect Node	Alignment Quaternion				Rest Pose Correction			
	w	x	y	z	w	x	y	z
HIP_CENTER	–	–	–	–	0.000	0.000	1.000	0.000
SHOULDER_LEFT	0.707	0.000	0.707	0.000	0.500	0.000	0.000	-0.866
ELBOW_LEFT	0.707	0.000	-0.707	0.000	0.981	0.000	0.000	0.195
WRIST_LEFT	0.707	0.000	-0.707	0.000	–	–	–	–
HAND_LEFT	0.707	0.000	-0.707	0.000	–	–	–	–
SHOULDER_RIGHT	0.707	0.000	-0.707	0.000	0.500	0.000	0.000	0.866
ELBOW_RIGHT	0.707	0.000	0.707	0.000	0.981	0.000	0.000	-0.195
WRIST_RIGHT	0.707	0.000	0.707	0.000	–	–	–	–
HAND_RIGHT	0.707	0.000	0.707	0.000	–	–	–	–
KNEE_LEFT	0.000	0.000	1.000	0.000	0.924	0.000	0.000	-0.383
ANKLE_LEFT	0.000	0.000	1.000	0.000	–	–	–	–
FOOT_LEFT	0.000	0.000	1.000	0.000	–	–	–	–
KNEE_RIGHT	0.000	0.000	1.000	0.000	0.924	0.000	0.000	0.383
ANKLE_RIGHT	0.000	0.000	1.000	0.000	–	–	–	–
FOOT_RIGHT	0.000	0.000	1.000	0.000	–	–	–	–

of input, for different systems may differ on what information is sent. For that, a simple XML file loader was implemented. Each XML file corresponds to a certain type of input device, containing information regarding each of the bones, how they are connected and their rotation properties. It is assumed that the joint information is always sent with a hierarchical structure in mind. All XML examples demonstrated below have the Kinect camera as an input system.

Each file has a simple structure, starting with a *skeleton* tag, followed by three group sub-tags:

- <*nodes*>, representing the expected receiving input nodes and their hierarchy dependencies;
- <*alignments*>, representing the rig's joints which require alignments in their coordinate system;
- <*rotations*>, representing the rig's joints which require correction for achieving the T-pose identity quaternion.

Each one of these sub-tags include several tags on their own. The <*nodes*> group represents how many nodes we are expecting to receive from the input system, allowing including several <*node*> tags. Each one of these contain an *id*, and optional *parent* and *bone* attributes, representing order in which they are processed, the *id* of the parent and to which rig bone they are referring to, respectively. Listing 4.1 exemplifies the chest node rotation problem, which has a specific set of dependencies between them, already mentioned in sections 3.1.3 and 4.2.

Listing 4.1: Code excerpt of the <*nodes*> group in the Kinect XML configuration file.

```
<nodes>
  <node id="0" />
  <node id="1" parent="0" bone="hips"/>
  <node id="2" parent="1" bone="spine"/>
</nodes>
```

The <*alignments*> groups allows having as many <*alignment*> group tags as required to represent the alignment quaternions. These <*alignment*> tags must include attributes *id* and *name*, representing the number of the joint we are aligning and its armature rig bone. The following listing 4.2 illustrates how the left arm area joints are affected by the alignment quaternions.

Listing 4.2: Code excerpt of the <*alignments*> group in the Kinect XML configuration file.

```
<alignments>
  <align id="4" name="shoulder.L">
    <rotation w="0.707" x="0" y="0.707" z="0" />
  </align>
  <align id="5" name="upper_arm.fk.L">
    <rotation w="0.707" x="0" y="-0.707" z="0" />
  </align>
  <align id="6" name="forearm.fk.L" >
    <rotation w="0.707" x="0" y="-0.707" z="0" />
  </align>
</alignments>
```

The <*rotations*> group can include <*bone*> group tags to represent the joint's corrections to produce the identity quaternion. Each <*bone*> tag must include *id* and *name* attributes, also representing the number of the joint we are aligning and its armature rig bone. Listing 4.3 demonstrates how the left arm area joints are corrected in relation to the T-pose, after being affected by the alignment quaternions.

Listing 4.3: Code excerpt of the <*rotations*> group in the Kinect XML configuration file.

```
<rotations>
  <bone id="4" name="shoulder.L">
    <rotation w="0.500" x="0" y="0" z="-0.866" />
  </bone>
  <bone id="5" name="upper_arm.fk.L" >
    <rotation w="0.981" x="0" y="0" z="0.195" />
  </bone>
  <bone id="6" name="forearm.fk.L"/>
  <bone id="7" name="hand.fk.L"/>
</rotations>
```

As seen in the examples above, both <*alignment*> and <*bone*> tags can include as many <*rotation*> tags as necessary, which in turn represent a quaternion rotation, with its w, x, y and z values.

There were two XML files created in total throughout development, one for each version of the Kinect. For the Kinect v2 version, the SDK requirements are completely different. One example being it requires Windows 8, while also conflicting with the v1.8 SDK. For that, we were forced to develop a separate tool version, that handled the new Kinect APIs, which also changed. Since our tool is accessing the Kinect APIs, we managed to test out some skeletons in a different manner. By using Kinect Studio, we were able to load files that were already created, presenting recorded performances in front of the Kinect camera. This application allows "emulating" a Kinect camera in other applications using the Kinect APIs simply by pressing a single button, playing the captured data from the loaded files, without really needing the camera device. This is fundamental because it provides a way to record and play the captured data as much as necessary. This method of analyzing data is also less tedious and more efficient than having to stand up every time we wanted to test out a scenario.

Most of the Kinect Studio files used for testing were acquired from footage of a theatrical play by the name of *"Este corpo que me ocupa"*, by João Fiadeiro. An example of how the captured data is represented in Kinect Studio can be seen in figure 4.4, and figures 4.5(a) and 4.5(b) represent an enactment of the recorded footage.

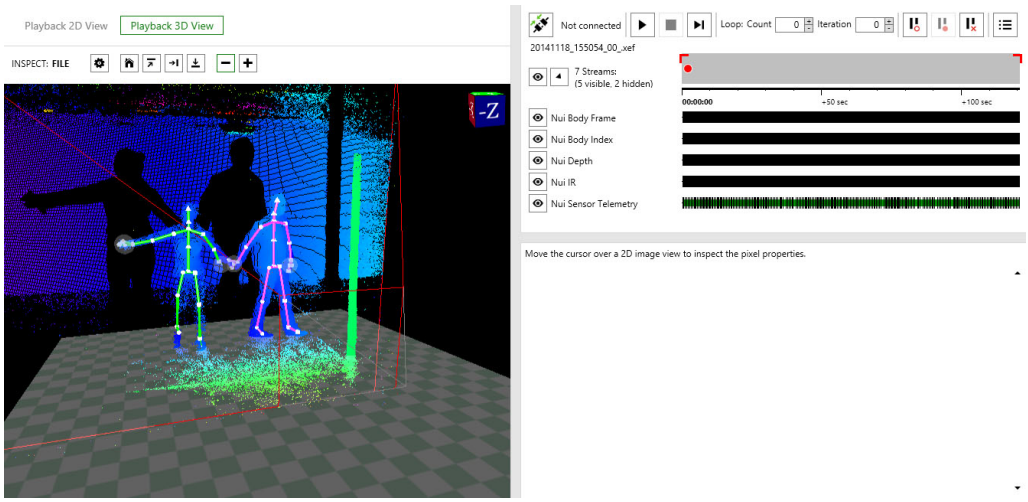
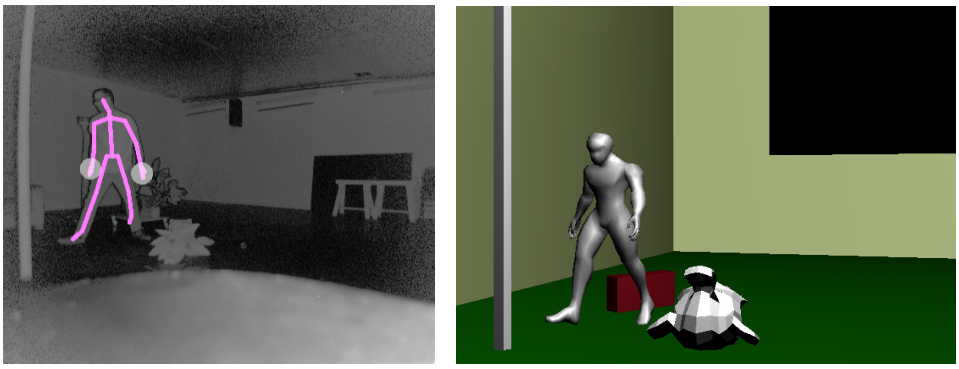


Figure 4.4: Kinect Studio's window display, after loading a file containing Kinect stream data.

Because most of the APIs changed, including the skeleton tracking data, the previously build XML file had to be changed. Firstly, we had to take in account the new nodes, and their dependencies. Then, we had to recalculate each of the joints' alignments, so that it matched the rig. Lastly, we had to find the new differences between the rig and the new skeleton's rest pose. All this resulted in a completely new file, which enabled us to work with any Kinect v2 skeleton stream data. On the other hand, this new skeleton stream data led to some interesting results, which will be presented in section 5.1.



(a) Kinect Studio stage display (b) Blender stage display

Figure 4.5: Representation of a theatrical play recorded with Kinect Studio (a), enacted in a Blender created stage (b).

4.4 Discussion

As it is shown in the sections above, all previously identified problems are now resolved. With this, our project's final prototype is complete, accomplishing the outlined goals. Using a Kinect camera, the performer's movements are interpreted and displayed on a virtual stage, represented by a 3D model of the performer. However, our system is ready to interpret any kind of real-time skeleton tracking input, as long as there is a XML file to correctly represent the new input data. The video file loader can also serve different purposes. By projecting the video on the virtual stage it could function as an extension of the performance itself, complementing the user's routine, or as a training aid for the user, mimicking the routine displayed on the video.

Out of the previously defined goals, making a rigged model was probably the one in which there were more doubts, since there was no previous knowledge of working with Blender, much less knowing the tools to make a static model animate correctly. It is safe to say that most of the early development cycle time was spent on making an initial animated model. The end result is a rig that can correctly display Kinect skeleton data, which was our objective from the start.

Part of the time spent making the model animate correctly was caused by the problem of the seemingly incorrect Kinect rotation values. These inaccurate values are also evidenced by the MoCapPlay and the *jumpkick.bvh* animation file, which needed some corrections as well. By studying how we could use quaternions to correctly represent the rotations from the Kinect in our rig, we were able to find out the corrections necessary for each of the skeleton nodes, and map them on a XML file. This file was our solution in order to standardize the tool's skeleton data input method. This way any camera capable of performing skeleton tracking can be mapped to a XML file, and correctly animate our rigged model. Two XML files were created, one for each Kinect device. However, since the Kinect v2 SDK is not a final stable version, it is subject to changes, which can lead to later corrections in its XML file.

5

Conclusions and Future Work

In this closing chapter we will present some final considerations concerning this dissertation and its development cycle. First comes a section presenting an evaluation of our system's prototype and results, followed up by the overall conclusions regarding our tool. Finally we will talk about possibilities of future work surrounding this project's subject matter.

5.1 Evaluation

After developing and witnessing how the system prototype ended up working, we reached some conclusions. The following paragraphs were written regarding what was determined when comparing all of our system's data input methods.

Constructing MoCapPlay was a very important step in the development of our system. It allowed us to understand how information could be sent to our rig, in order to animate it. However, the skeleton presented in the *jumpkick.bvh* file is very similar to the Kinect skeleton, which worked for our advantage, since our rig is prepared for to exclusively receive Kinect skeleton data. This could be a problem for different files, with different skeleton bone compositions. In order to be optimized for any BVH animation file, some adjustments would have to be made to the whole system.

The available Kinect documentation lacks information about how each of the nodes' rest positions is orientated, which led to more calculations on our side. This point aside, the documentation has enough information to understand how the data can be reached and how to use it in our virtual avatar. One thing to notice is that when capturing skeletons, the distance of the performer relative to the camera matters, if only to ease the

computations for the skeleton tracking algorithm. If the algorithm tracks part of a skeleton, for example, when not exposing the entire body for being too close to the camera, all undetected node positions and rotations are going to be inferred. This can possibly lead to incorrect data, so it is recommendable that the capturing space is unobstructed. The camera is capable of detecting skeletons in a range of 0.5 to 4 meters, but after testing we determined the ideal distance to the camera is about 2.5 meters for it to correctly capture all of the skeleton nodes.

The Kinect skeleton tracking algorithm is also not defined for capturing people with their backs turned, or standing sideways. If the performer is required to rotate its body, it is recommendable to stand facing the camera, letting the algorithm detect all of the nodes, and only then start rotating. With this, even if the body is sideways, since all correct node positions and rotations were previously calculated, the inferred node data will be close to an optimal representation of a sideways skeleton. This problem could be minimized in the future if, instead of one camera, we used more cameras with different placements, and triangulate the captured skeleton rotations. When applying the rotation values to the rig, there is still a noticeable stiffness of the bones, which can be related to how the armature was constructed. Since the Kinect skeleton detection depends on the user's body type and its distance to the camera, it can vary in terms of bone size and positioning. So, for our prototype, we had to settle for a middle ground, constructing a standard sized skeleton to use as armature.

The new detection capabilities of the Kinect v2 camera, already mentioned in section 2.2.2, bring new potential to what can be made with it, especially in the gesture and face tracking field. The improved depth sensors can also help in reconstructing possible stages for the performers to act. Our system could then be capable to recreate a complete stage, including objects that could be virtualized onto the stage for the performer to interact. However, since the available Kinect v2 SDK is still not the final release version, its APIs are preliminary and subject to change. This is evidenced by the complete lack of documentation regarding this version. The new joint additions, notably the new thumb joints, change how the overall arm orientation is calculated. By verifying the the thumb's position relative to the arm, the new arm orientation is then defined. Also, the overall skeleton tracking algorithm still has difficulties in instantaneously figuring out where the thumb is located, making the overall arm orientation values fluctuate, which makes it very difficult to stabilize the arm in order to analyze the joints' orientation.

5.2 Conclusion

The final result of this dissertation presents a tool capable of simulating a virtual performer doing a routine on a virtual stage, enabling the use of a reconstructed 3D model of the user if they so desire. With that said, the user is allowed to freely perform their routine in front of a camera, which in turn will recognize each body limb due to skeleton tracking properties internally implemented by the camera. This skeleton information is

then processed and converted to values accepted by our rigged 3D model, specified by specific XML files. These XML files are our way to standardize the values to be injected in our rigged 3D model, in order to be able to receive any type of input, which include BVH animation files. As a way to further aid this routine practice, the tool allows playing video files in the virtual stage's background.

When comparing with the studied similar systems specified in section 2.1, our tool is different in the way that it functions within a fully functional physics engine, which allows interacting with objects in a much more intuitive and ordinary manner. There are systems that work in a similar way [4], but instead of reacting on triggers set up on the objects, the triggers are in the poses the dancer performs. This requires a large database of poses to be used for comparison, which the user must also know about, making it more complex than a simulation of a real life physics engine. The XML file loader is what differentiates our tool from other existent tools, such as NI Mate [40]. Our system is capable of receiving data from any capture device, not being limited to the Kinect, and is capable of representing it on any 3D simulation engine, not just Blender.

By falling back on a non intrusive motion capture method, namely the Kinect camera, the quality of the motion tracking is not going to be anywhere near perfect. It can, however, provide some ideas as to how the body should stand and what rotations can a human body perform. The specific model used for testing this tool had some limitations, particularly the inability to correctly capture a body turned sideways or even distinguish between the front and back of a user standing before the camera. If it were a motion capture suit or some kind of sensors attached to the body limbs, such as the ones used in project RAM [10], the values would probably be much more reliable.

As for the choice of 3D engine to present our results, the decision to use Blender was certainly a success, due to the different object properties available in the internal game engine and Rigify being such a powerful tool. This add-on made the process of rigging a model so much easier, almost in an intuitive way. The ability to individually rig certain preset skeleton parts enables for some really interesting possibilities, due to its building block approach. Our approach for the animation rig used a skeleton model based of a capture performed with Brekel (section 2.3.3), which also uses Kinect as a default capture device. The inclusion of the background video player was a goal we also pursued. Resorting to Blender's engine, we were able to display video using a special texture type. This allowed for a much more reliable training method, granting the user the choice to follow a guideline provided by the selected video.

Also mentioned in section 4.4, most of the time spent on the development cycle was on correctly converting the values received from the Kinect into rotation values for our 3D model. To help figuring out exactly what values were needed an additional tool was developed. This tool, MoCapPlay (section 3.1.3), loads a BVH animation file, which follows a certain skeleton hierarchy and its rotational values along the course of time. These values are then converted into quaternions and injected in our 3D model, animating it.

Overall, we consider the project's final result is successful, even with its flaws. We

managed to create a system which is capable of using any type of input device to simulate animated movement on a virtual environment, which was the main goal from the beginning.

5.3 Future Work

We will conclude this document with some suggestions of directions to take from this project's development.

From the project's current state, it is safe to say that some of the current features were not completely implemented, and could be improved upon. One such feature is a video player control panel. Actions found normally on any media player such as pause, stop, play, forward and rewind were not implemented due to time constraints. The same happened to sound control actions such as volume up, volume down and mute. Camera control options could could also be implemented, giving the option to pan around the stage and maybe zoom in and out.

The option to record performances could be also an future improvement. These recordings could be a video recording of the performance, saved into a video file. Or maybe a skeleton animation recording, saved into a BVH file, even though there are already tools that are capable of converting Kinect data into BVH files.

In terms of technology constraints, better capture methods could be used. Instead of just one camera device, more could be used at the same time, calculating a single skeleton from different points of view. This option would eliminate most of the problems with detecting bodies not standing facing the camera. If we were to use a motion capture suit, or a specific intrusive motion capture device optimized for body movements, better data could be obtained. However, these options are out of the project's budget, so they were not considered, but they are still options to consider.

An interesting feature that could be investigated is related to the 3D models used in Blender. A system that could detect and distinguish between different users in front of the camera, and to those users associate a certain 3D mesh reconstructed from the performer, could be a major improvement in simulating reality. With this, and finding a way to instantly rig and change the performer model in real time when a body was detected in front of the camera, it would improve the user experience.

Bibliography

[1] Microsoft. *Kinect for Windows*. Last Access: Apr. 2014. URL: http://www.microsoft.com/en-us/kinectforwindows/.

[2] *openFrameworks*. Last Access: Apr. 2014. URL: http://www.openframeworks.cc/.

[3] J. G. Joshua Noble. *Kinect Common Bridge*. Last Access: Apr. 2014. URL: https://github.com/joshuajnoble/ofxKinectCommonBridge.

[4] Q. Wu, P. Boulanger, M. Kazakevich, and R. Taylor. "A Real-time Performance System for Virtual Theater". In: *Proceedings of the 2010 ACM Workshop on Surreal Media and Virtual Cloning*. SMVC '10. Firenze, Italy: ACM, 2010, pp. 3–8. ISBN: 978-1-4503-0175-6. DOI: 10.1145/1878083.1878087. URL: http://doi.acm.org/10.1145/1878083.1878087.

[5] Y. Fei, D. Kryze, and A. Melle. "Tavola: Holographic User Experience". In: *ACM SIGGRAPH 2012 Emerging Technologies*. SIGGRAPH '12. Los Angeles, California: ACM, 2012, 21:1–21:1. ISBN: 978-1-4503-1680-4. DOI: 10.1145/2343456.2343477. URL: http://doi.acm.org/10.1145/2343456.2343477.

[6] B. Baird, O. İzmirli, and A. Joshi. "Using Motion Capture to Synthesize Dance Movements". In: *Proceedings of the Thirteenth Biennial Symposium on Arts and Technology at Connecticut College*. New London, CT, USA, 2012.

[7] C. Griesbeck. *Introduction to Labanotation*. Last Access: Jan. 2014. URL: http://user.uni-frankfurt.de/~griesbec/labane.html.

[8] J. C. P. Chan, H Leung, J. K. T. Tang, and T Komura. "A Virtual Reality Dance Training System Using Motion Capture Technology". In: *IEEE Transactions on Learning Technologies* 4.2 (2011), pp. 187–195. ISSN: 1939-1382.

[9] A. Schulz and L. Velho. "ChoreoGraphics: An Authoring Environment for Dance Shows". In: *ACM SIGGRAPH 2011 Posters*. SIGGRAPH '11. Vancouver, British Columbia, Canada: ACM, 2011, 14:1–14:1. ISBN: 978-1-4503-0971-4. DOI: 10.1145/2037715.2037732. URL: http://doi.acm.org/10.1145/2037715.2037732.

[10] Y. C. for Arts and M. InterLab. *Reactor for Awareness in Motion (RAM)*. Last Access: Jan. 2014. URL: http://interlab.ycam.jp/en/projects/ram.

[11] I. Baran and J. Popović. "Automatic Rigging and Animation of 3D Characters". In: *ACM SIGGRAPH 2007 Papers*. SIGGRAPH '07. San Diego, California: ACM, 2007. DOI: 10.1145/1275808.1276467. URL: http://doi.acm.org/10.1145/1275808.1276467.

[12] N. Vegdahl. *Blender Extensions: Rigify*. Last Access: Apr. 2014. URL: http://wiki.blender.org/index.php/Extensions:2.6/Py/Scripts/Rigging/Rigify.

[13] Microsoft. *Kinect for Windows Sensor Components and Specifications*. Last Access: Jan. 2014. URL: http://msdn.microsoft.com/en-us/library/jj131033.aspx.

[14] Microsoft. *Color Stream*. Last Access: Jan. 2014. URL: http://msdn.microsoft.com/en-us/library/jj131027.aspx.

[15] Microsoft. *Depth Stream*. Last Access: Jan. 2014. URL: http://msdn.microsoft.com/en-us/library/jj131028.aspx.

[16] K. Liu, Y. Wang, D. L. Lau, Q. Hao, and L. G. Hassebrook. "Dual-frequency pattern scheme for high-speed 3-D shape measurement". In: *Opt. Express* 18.5 (2010), pp. 5229–5244. DOI: 10.1364/OE.18.005229. URL: http://www.opticsexpress.org/abstract.cfm?URI=oe-18-5-5229.

[17] J. MacCormick. *How does the Kinect work?* Sept. 2011. URL: http://pages.cs.wisc.edu/~ahmad/kinect.pdf.

[18] R. Vision. *How Kinect and Kinect Fusion (Kinfu) Work*. Last Access: Jan. 2014. URL: http://razorvision.tumblr.com/post/15039827747/how-kinect-and-kinect-fusion-kinfu-work.

[19] Microsoft. *Infrared Stream*. Last Access: Jan. 2014. URL: http://msdn.microsoft.com/en-us/library/jj663793.aspx.

[20] Microsoft. *Skeletal Tracking*. Last Access: Jan. 2014. URL: http://msdn.microsoft.com/en-us/library/hh973074.aspx.

[21] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake. "Real-time Human Pose Recognition in Parts from Single Depth Images". In: *Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition*. CVPR '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1297–1304. ISBN: 978-1-4577-0394-2. DOI: 10.1109/CVPR.2011.5995316. URL: http://dx.doi.org/10.1109/CVPR.2011.5995316.

[22] Structure. *OpenNI 2 Download and Documentation*. Last Access: Jul. 2014. URL: http://structure.io/openni.

[23] O. Project. *OpenKinect*. Last Access: Jul. 2014. URL: http://openkinect.org/wiki/Main_Page.

[24] Microsoft. *Kinect for Windows features*. Last Access: Oct. 2014. URL: http://www.microsoft.com/en-us/kinectforwindows/meetkinect/features.aspx.

[25] Microsoft. *Collaboration, expertise produce enhanced sensing in Xbox One*. Last Access: Oct. 2014. URL: http://blogs.microsoft.com/blog/2013/10/02/collaboration-expertise-produce-enhanced-sensing-in-xbox-one/.

[26] C. V. Online. *Time-of-Flight Cameras*. Last Access: Oct. 2014. URL: http://www.computervisiononline.com/books/computer-vision/time-flight-cameras.

[27] Microsoft. *Kinect Fusion*. Last Access: Jan. 2014. URL: http://msdn.microsoft.com/en-us/library/dn188670.aspx.

[28] OpenSLAM. *OpenSLAM*. Last Access: Jan. 2014. URL: http://openslam.org/.

[29] V. Metric. *3D Scanning Technology Overview: Kinect Reconstruction Algorithms Explained*. Last Access: Jan. 2014. URL: http://voxelmetric.com/3d-scanning-technology-overview-kinect-reconstruction-algorithms-explained/.

[30] Z. Zhang. "Iterative point matching for registration of free-form curves and surfaces". English. In: *International Journal of Computer Vision* 13.2 (1994), pp. 119–152. ISSN: 0920-5691. DOI: 10.1007/BF01427149. URL: http://dx.doi.org/10.1007/BF01427149.

[31] K. Jahrmann. *Kinect Fusion - Reconstruction*. Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, Feb. 2013. URL: http://www.cg.tuwien.ac.at/research/publications/2013/jahrmann_klemens_KFR/.

[32] B. Curless and M. Levoy. "A Volumetric Method for Building Complex Models from Range Images". In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. New York, NY, USA: ACM, 1996, pp. 303–312. ISBN: 0-89791-746-4. DOI: 10.1145/237170.237269. URL: http://doi.acm.org/10.1145/237170.237269.

[33] ManCTL. *Skanect by Occipital*. Last Access: Dec. 2013. URL: http://skanect.manctl.com/.

[34] ASUS. *Multimedia - Xtion PRO*. Last Access: Jan. 2014. URL: http://www.asus.com/Multimedia/Xtion_PRO/.

[35] SourceForge. *MeshLab*. Last Access: Jan. 2014. URL: http://meshlab.sourceforge.net/.

[36] Autodesk. *3ds Max 3D Modeling and Rendering Software*. Last Access: Jan. 2014. URL: http://www.autodesk.com/products/autodesk-3ds-max/overview.

[37] B. Chazelle. "An optimal convex hull algorithm in any fixed dimension". English. In: *Discrete & Computational Geometry* 10.1 (1993), pp. 377–409. ISSN: 0179-5376. DOI: 10.1007/BF02573985. URL: http://dx.doi.org/10.1007/BF02573985.

[38] J. Brekelmans. *Brekel*. Last Access: Dec. 2013. URL: http://www.brekel.com/.

[39] Autodesk. *3D Character Animation MotionBuilder Software*. Last Access: Jan. 2014. URL: http://www.autodesk.com/products/motionbuilder/overview.

[40] Delicode. *NI Mate*. Last Access: Dec. 2013. URL: http://www.ni-mate.com/.

[41] Autodesk. *Maya 3D Animation Software Computer Animation*. Last Access: Jan. 2014. URL: http://www.autodesk.com/products/autodesk-maya/overview.

[42] U. Technologies. *What is Unity*. Last Access: Oct. 2014. URL: https://unity3d.com/pages/what-is-unity.

[43] B. Institute. *Home of the Blender project - Free and Open 3D Creation Software*. Last Access: Jan. 2014. URL: http://www.blender.org/.

[44] B. Institute. *Elephants Dream*. Last Access: Jan. 2014. URL: http://www.elephantsdream.org/.

[45] B. Institute. *Big Buck Bunny*. Last Access: Jan. 2014. URL: http://www.bigbuckbunny.org/.

[46] B. Institute. *Sintel, the Durian Open Movie Project*. Last Access: Jan. 2014. URL: http://www.sintel.org/.

[47] B. Institute. *Tears of Steel | Mango Open Movie Project*. Last Access: Jan. 2014. URL: http://www.tearsofsteel.org/.

[48] B. Institute. *Yo Frankie! - Apricot Open Game Project*. Last Access: Jan. 2014. URL: http://www.yofrankie.org/.

[49] B. Institute. *Sintel The Game - A game based on the Blender Foundation movie: Sintel*. Last Access: Jan. 2014. URL: http://sintelgame.org/.

[50] BlenderNation. *Hungarian performance uses Blender Game Engine and Kinect for live effects*. Last Access: Jan. 2014. URL: http://www.blendernation.com/2013/10/28/hungarian-performance-uses-blender-game-engine-and-kinect-for-live-effects/.

[51] T. U. of Ostrava. *Blender & Kinect*. Last Access: Jan. 2014. URL: http://blender.vsb.cz/index.php/en/kinect-blender.

[52] Polygon. *Unity for Wii U opens up GamePad hardware and more to developers*. Last Access: Oct. 2014. URL: http://www.polygon.com/2013/8/20/4641786/unity-for-wii-u-opens-up-gamepad-hardware-and-more-to-developers.

[53] U. Technologies. *License Comparisons*. Last Access: Oct. 2014. URL: https://unity3d.com/unity/licenses.

[54] Zigfu. *Zigfu - Kinect Development*. Last Access: Oct. 2014. URL: http://zigfu.com/.

[55] U. Armenia. *KinectV2 + Unity3D Plugin July Update Testing*. Last Access: Nov. 2014. URL: http://unity3d.am/2014/07/03/kinectv2-unity3d-plugin-july-update-testing/.

[56] Microsoft. *Tracking Users with Kinect Skeletal Tracking*. Last Access: Jun. 2014. URL: http://msdn.microsoft.com/en-us/library/jj131025.aspx.

[57] S. Arietta. *CS445: Graphics*. Last Access: Jun. 2014. URL: http://www.cs.virginia.edu/~gfx/Courses/2010/IntroGraphics/Lectures/29-Quaternions.pdf.

[58] C. M. University. *CMU Graphics Lab*. Last Access: Nov. 2014. URL: http://mocap.cs.cmu.edu/.

[59] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice, Second Edition in C*. Addison-Wesley Professional, 1990.

[60] I. Advanced Micro Devices. *Real-Time Physics Simulation*. Last Access: Nov. 2014. URL: http://bulletphysics.org/wordpress/.

[61] F. team. *FFmpeg*. Last Access: Aug. 2014. URL: https://www.ffmpeg.org/.