MdH University - IDT

Università degli Studi dell'Aquila

Virtual Test Environment for Motion Capture Shoots

Master Thesis

GSEEM Master Student:
Claudio Redavid

MDH Examiner:
Rikard Lindell

MDH Supervisor:
Oğuzhan Özcan

Company Advisor:
John Mayhew

UDA Supervisor:
Dajana Cassioli

Declaration of Honor

Declaration

I hereby declare that this master thesis is my own work. No part of this master thesis is taken from other sources without referencing them clearly.

Claudio Redavid
Uppsala, July 6, 2013

Abstract

This master thesis presents the design of an implementation of a working prototype for an augmented motion capture acting environment. Motion capture (MoCap), the recording of movements to be applied to characters or objects in computer graphics simulations, is widely used in video games, advertisement, and cinema. MoCap is realized through different techniques, where one common problem is the efficiency to capture actor's motion performances. To capture motions without obstacles for the motion detectors, actors act blindly, without graphic references while acting. Mistakes or a poor correlation between the actors performances and the computer graphics simulation requires the scene either to be taken many times or to be corrected afterwards in time-consuming post-production. These issues slow down the production process or lead to a low quality product.

We suggest that one way to limit the problem of efficiency in motion capture is to let actors perform in a virtual environment. To this end, this master thesis presents a simple prototype environment with the goal to support actors' performances to improve motion capture efficiency. The idea is to surround an actor with four screens which display the virtual environment. A Microsoft Kinect camera is utilized for motion capture. Gestures are used to trigger interactions between the actor and the virtual environment. Furthermore, the thesis explores the applicability of open source libraries, game engines, and inexpensive general purpose technology. We suggest, as indicated by demonstrated validity, that virtual environments and augmented motion capture improve the conditions for actors, thus providing more efficient motion capture shots. However, further research and quantitative measurements are needed to understand and fully evaluate the effect of the presented prototype tool.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
CAVE	Cave Automatic Virtual Environment
Fubi	Full Body Interaction Framework
HCI	Human Computer Interaction
IDE	Integrated Development Environment
MoCap	Motion Capture
NI	Natural Interaction
NITE	PrimeSense's Natural Interaction Technology for End-user
NIUI	openNI Unreal Integration
UDK	Unreal Development Kit
UI	User Interface
VR	Virtual Reality

1 Introduction

With the large diffusion of multimedia products, such as movies as well as video games, improved quality is a competitive advantage. For example, re-create real world scenario or even more realistic special effects will attract more people and make the product more interesting. Motion Capture (MoCap) comes to help these companies to obtain realistic results without spend money to recreate expensive scenarios or animate complex virtual characters. Motion Capture is a transposition of a real asset into a virtual world, where professional actors are recorded and their gestures are elaborated and reflected on virtual models. During the recording process MoCap actors, most of the time, have to improvise a scene or fake to speak or shoot to an invisible target. Consequently, the result of the recording may not be natural, therefore actors are forced to do it again and again.

The purpose of this thesis is to understand if we can create a simple and inexpensive environment to see the level of immersion of motion capture actors. In this respect, we are not aiming to understand the performance of the test tool but we want to see which technology is utilizable to create such a tool. Therefore, quantitative evaluation of this tool is out of scope of this thesis; further research will accomplish this.

The development process was set for answering the following questions:

- "Is it possible to create an immersive environment for a MoCap scenario?".
- "Is it possible to create such environment flexible and inexpensive?".

We suggest to immerse MoCap actors in a virtual 360 degree interactive environment to let them feel more immerse in the scene and, consequently, helps professionals to increase the overall MoCap process efficiency. This basic idea intrigued professionals working into Motion Capture sector. Therefore, there was proposed a framework with 4 screens, on which a 3D world is projected, placed around the MoCap actor and a camera that recognizes actor gestures and movements. The virtual environment is also interactive. The actors' actions trigger events in the world in accordance with the MoCap scenario. At the end of the development the final system was presented to a MoCap company professionals due to collect feedbacks and critics. The system was not used by real actors since it did not pretend to be a definitive solutions but a starting point to develop a more complete and functional system.

The overall system was divided in hardware and software part. The software part is composed by the UnrealEngine framework to create virtual world. The hardware part is composed by 4 screens placed around the actor, and a Kinect camera that keeps the trace of all the actor gestures and movements. The combinations of the software and hardware components created a virtual environment to help MoCap actors making more natural

gestures and react to events as it should happen in reality. Therefore, the final MoCap scene will need less post-processing, and customers are provided higher quality of the final product. Above all, it has to be clear that the presented framework shows a potentially usable possible solution to increase efficiency in real motion capture sessions, even if it was not pretending to solve all the problems.

2 Related Work

This section present the related research in MoCap. The research area was restricted on two problems in MoCap companies: the loss of data and the not-natural actor interaction. There were analyzed the strong points of the ideas founded in the state of the art to create an immersive virtual environment that can be used in a Motion Capture context.

2.1 Motion Capture

The fast development of advanced technology has permitted computer and 3D modelling to be used into movies, advertisement, education, and of course video games [Lengyel (1998)]. Therefore, a lot of work was requested to the artist to create and animate virtual model of cars, buildings, aliens and so on. Moreover, with the growing market it was necessary more works and money to provide realistic results in a short amount of time [Hilton (1999)].

MoCap helps producers to impress clients with fast and inexpensive realistic results. It is a transposition of a real-world objects, such as a person walking or fighting or even a horse running, into a virtual 3D environment [Maiocchi (1996)] (Figure 2.1).

The motion capture (MoCap) process is composed by basically 4 phases [Moeslund and Granum (2001)]:

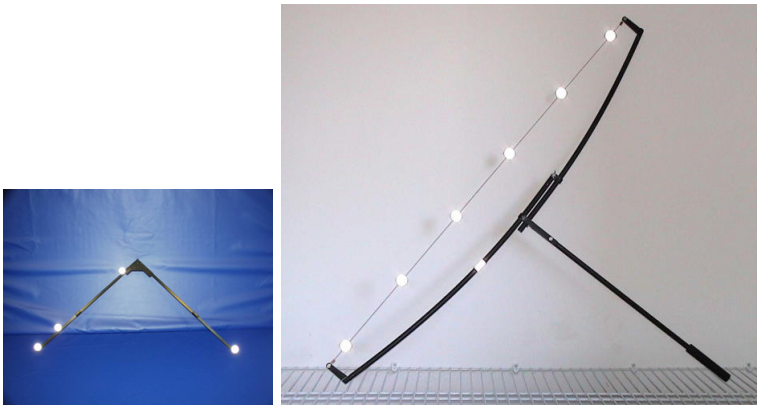
- Initialization
- Tracking
- Pose estimation
- Recognition

In short, during the initialization phase actors, directors and technicians get ready to catch the scene. An important process of the initialization is to calibrate all the motion capture cameras. The tool showed in Figure 2.2(a) is used to indicate where the plane axes should be placed, than the other one in Figure 2.2(b) is brought around the scene field, while shaking and moving it, to calibrate all the cameras. Eventually, the director explain the scenario to the actors.

The tracking step concerns all the actions performed by actors that are traced and stored. In the next phase of pose estimation technicians link movements with a virtual model. Technicians can match, for example, orientation and location of actor bones to a 3D model that will have to move and act like a real human [Silaghi et al. (1998)].

Figure 2.1: MoCap examples [(2013*a*), (2013*b*)].

At the end, the recognition phase groups all post processing steps for improving the quality of the final result, such as fix a missing movement that has been lost during the tracking.



(a) (b)

Figure 2.2: Calibration tools used before any MoCap registration work-set [(2013*c*), (2013*d*)].

The most important phase in MoCap is to record actors movement. The final product quality may vary in base of the technique used or the accuracy of tracking system [Rose et al. (1997)]. Each MoCap company uses different techniques. Some prefer passive solution with markers putted on special clothes dressed by actors and infrared cameras. Others use active technologies, with special hardware components placed on the actor body that transmit his position, orientation, etc. in a wireless or wired way to operators

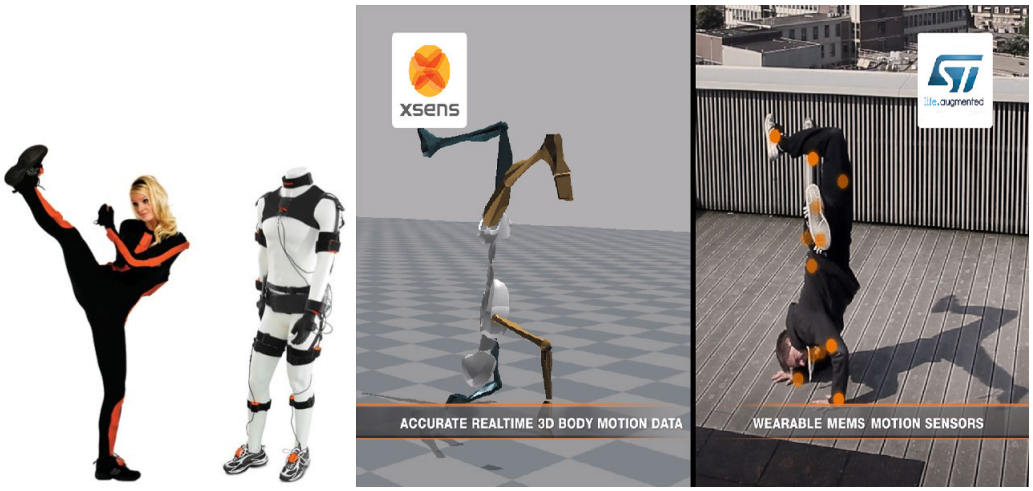


Figure 2.3: MoCap marker less environment [(2013*e*)].

(Figure 2.3). In [Hasler et al. (2009)] is presented a markerless system using multiple cameras calibration and triangulation. As first instance actors have to act in a close studio surrounded by several infrared cameras that catch all the markers movement. Moreover, the scene could be recorded without the use cameras, for example, in the real environment, such as a street or a soccer field. However, recording without markers is much less precise because of the limitation of spots that can be tracked. In the other hand, using markers and cameras is more precise but sacrifice a more real reaction by the actors.

Although sensors accuracy and computer vision technology have increased over the years, and have become more efficient and useful, there are still some problems affecting the MoCap process. Post-processing is still an important component since during recording there can be technical problems or mistakes that require operator's assistance.

For a more accurate and deep description of all the process related to the MoCap world can be consulted [Moeslund and Granum (2001), Rose et al. (1997)].

2.2 Increase Efficiency

Several solutions to reduce post processing and speed up the entire process without loss of data have been proposed [Barbič et al. (2004)]. The approach automatic divide in small segment a long action sequence, indeed in this way the actors don't have to stop and it will result in a more natural interaction. Moreover, all the methods use and automatic segmentation algorithm, avoiding the long and sometimes tedious work of programmer to recognize single action into the sequence.

As result of this first research, an uninterrupted actor performance can reduce post processing time and consequently increase the overall MoCap process efficiency. Therefore,

a virtual and immersive world around MoCap actors can eventually reduce the breaks between different scenes and reduce the need of a director to guide actors during all the MoCap action.

2.3 Increase Immersion

In the state of art, there are a lot of works proposing a CAVE approach. The Cave Automatic Virtual Environment (CAVE) is a high-tech virtual reality system, a cube shaped room which offers a multi-person, multi-screen, high-res 3D video and audio interactive environment. As the user moves and interacts within the display boundaries, the correct 3D perspective is displayed in real-time to achieve a fully immersive experience [Jacobson and Preussner (2010)]. The idea of 4 screens all around the actor, which during all the shoots will remain in the middle, was inspired by this last approach.

One of them is CASALA[1]. It is a project developed by the Netwell Centre and the Software Technology Research Centre, with four screens to project the world and Kinect to recognize gestures. In their approach, four screens are disposed: three around the user and one at the floor. A Microsoft Kinect is detecting the actors' movement in the MoCap scenario.

The actors are statically fixed in the middle of the room and the gestures to walk or rotate the camera are not natural but programmed before. Therefore, this kind of behaviour is not desired in this thesis work. However, this project show how it is possible to use the Kinect to recognize gestures for interacting with virtual objects in the MoCap context. Immersion and interaction are the key points of this approach, implemented in the project through bidirectional interactions between actors and the virtual environment.

Ultimate Battlefield 3 Simulator (2.4) is an experimental 360° environment with a walking platform, Kinect controller, speakers, one big screen all around the user and a paint gun. Although it gives a real deep immersion sensation and uses a nice and effective solution for walking around a huge map in a restricted real room, it's too big and expensive for our purpose and it is not reliable in MoCap context. Even if this approach requires too many resources to be realized, was useful to understand the importance to have a 360° view to give users semi realistic experience and let feel them as they are inside the virtual world, freely to move and interact with it.

The FlatWorld project[2] is developed by ICT (Institute for Creative Technologies) department of South Caroline University. It is an immersive virtual reality environment where the user explores a simulated world projected onto rear projector screens, and reacts with audio and animations due to user movements, that are tracked by infrared cameras. In this environment the interaction between users and the system is bidirectional, while the first walk around the world the latter can activate events or try to speak and communicate with users, similar approach is used in our project.

[1] WebSite of CASALA `http://www.casala.ie/3d-cave.html`
[2] `http://projects.ict.usc.edu/mxr/work/flatworld/`

Figure 2.4: Ultimate Battlefield 3 Simulator [(2013*f*)].

All the projects described above could be applied in a general purpose, while the one developed has been designed for being used in commercial MoCap. Therefore, it was necessary a virtual and interactive scenario that well fits, without interfere, in a MoCap recording session. Also, the main intention was to keep actors' gestures and reactions as close to reality as possible.

[Oshita (2006)] propose an approach to capture movement and reproduce it into a virtual environment with a 3d person camera. They developed a tracking system with a normal camera and a virtual world where the character will move and interact. The camera records the motion, and the system does a gesture database lookup to recognize what the actor is doing. Then, a 3D avatar reflect the user actions using mathematical algorithms to perform reaction of falling or walking. The first difference with our approach is that we use UDK to build the virtual world, the engine, therefore, will manage all the physics of the avatar or the objects into the world. Moreover, we do not have to worry about how to manage impact reaction or animation inside the 3D map, the engine will provide an easy way to build and control all the events that could happen in the recreated reality. In addition, one of advantage in contrast to that approach is the usage of Kinect camera and Kinect framework to overcome all problems that belong to online movements tracking and so animates a virtual character [Shin et al. (2001)]. The developed framework integrates all the algorithms necessary to capture a person gestures. Moreover, it gives to programmes API for manipulating the output data. Bone joints position and rotation are traced and subsequently send to clients in real time with a reasonable delay. Different people with different height or size can be traced, and their movement will be easily transferred to a virtual environment, without using filter, or adapting motion to a new character [Gleicher (1998)].

2.4 Concluding Remarks

In conclusion, in the actual state of the art there was not found a valid solution for increasing immersion in MoCap context. It was also clear that to increase MoCap shots efficiency the scenes must not be repeated so often and the post-processing should not interfere markedly. Accordingly to that, actors have to interact with something, or someone, for acting more naturally; as consequence they will not repeat the scene several times. In this scenario, the actor will be alone in the set and he is forced to imagine the army and the enemies. Recording phase will be repeated more the one time before he takes confidence with that scene.

The time loss will slow the overall process and will require a post-processing works to fix some left imperfections in the actor gestures. As suggested in the CASALA and FlatWorld projects, although they were applied in other fields, there was showing users performance increment using these immersive systems. [Oshita (2006)] inspired to use of a gesture recognition system to make the interactions between the user and the virtual world more complex and natural..

3 Method

The state of the art research introduced new ideas to use modern techniques for creating a virtual, immersive and portable framework. Moreover, some existing project solutions suggested specific hardware for using this virtual scenario in a MoCap context without interfere with motion cameras.

The development process will be divided into three phases:

- Design
- Implementation
- Experimentation phase

Designing part was a key step during the entire system development process. Initial design fixed the idea all together, defined the scope and exposed it to stakeholders through video sketches [Zimmerman (2005)]. Afterwards, there were chosen the right software and hardware components to use. The software was composed with the UnrealEngine framework that was used to created a virtual map for the MoCap scene. There were designed several scenarios to use in the tridimensional world. Every scenario contains specific actions and situations for the MoCap actors. Every action and gesture performed by the actors is recognized and reproduce on the virtual set. Moreover, UnrealEngine [Mooney (2012)] tools and open source libraries were used to reflect actors motions in a virtual map and, if necessary, to create a real time response into the world. The hardware part was composed by four screens placed around the actor, a computer that runs UnrealEngine and a Kinect camera attached to the computer. The main design stated that at motion capture scene start, the virtual world has to be projected on all the four screens to give to the actor. A "simulated 360 degree" view and so increase his immersion feeling. When he or she is acting according to the MoCap action schedule the Kinect is tracing of all the gestures and movements, so, the virtual environment will react as established.

During implementation, open source solutions were used to connect unreal engine and a Kinect camera. An open source driver was used to enable the Kinect camera on a personal computer [Villaroman et al. (2011)]. This step was the most complicated during the development process, because managing three different open source libraries and make them communicate among each other comported certain effort. These libraries were poorly documented, showed incompatibility between versions, and there were several internal bugs. All these issues involved in loss of time and several revisions cycles.

At the end, only professionals inside the company were called to use the system. The framework was an alpha version, it was not stable and therefore, no real actors were called to test it. Despite that, it was an early release and it was possible to experience different

configuration of it. At the beginning there were used the original screen positions: four all around while the actor is set in the middle of them. Afterwards, the framework was run with different screens settings and there were reported all the issues encountered. This first phase was necessary to understand if the system was flexible enough to support changes in the hardware. After some fixing on the framework, in the last phase there were collected feedbacks from professionals.

As a brief conclusion of the development process, it was clear that all the critics gained during experimentation could be used to improve the actual framework and make it possible to use it in a real motion capture session and maybe test it again with real actors while comparing their efficiency either using the system that without it.

A detailed description of all the steps dealt will follow in the next sections.

4 Design

The first challenge during design was finding the right hardware components to support the original purpose. Using rear projection screens or a huge panel all around the MoCap room may be a full immersive solution, however, is too expensive and bulky for a testing purpose. Instead, was opted for a simpler solution with four normal screens, which could be updated into projector screens afterwards, if necessary. All of them are connected to one computer with double graphic cards, controlled by an operator. The disposition of the four screens may vary in base of the MoCap session scenario, but they should surround the actor such that he will feel more inside the action and so act in a more realistic way. The Kinect camera is connected to the same computer. The Kinect camera will always be placed in front of the actors' starting position; in this way the calibration will be fast and the tracking more accurate.

The next challenge was find out how to track the actors. As initial solution we suggested to use the motion capture cameras. However, this option may cause latency in data recording, furthermore, it will be necessary to create a specific library to integrate the cameras with a game engine. Therefore, it was used the Kinect camera because of it is easy to install and to integrate with game engine. Kinect camera comes furnished with USB cable, and on the web it is possible to find two drivers to control it through a PC: Microsoft SDK Driver and OpenNI framework. Despite the Microsoft SDK is more accurate and is installed as default driver of the camera, it was used the OpenNI framework. OpenNI is well compatible with Kinect and has an easy API interface and it is more documented. OpenNI comes with a build library to connect it with different game engines.

To build a 3D world from the base is not complicated; however, it requires knowledge in different topics such as computer graphics, physics, animations, and so on. For this reason, UnrealEngine 3[1] has been chosen to create and animate the virtual environment. UnrealEngine 3 is easy to integrate with the Kinect OpenNI framework. Moreover, it comes with powerful and easy to use tools, such as, the editor. The editor makes programmers life easier, they can use it to both create entire maps without write a single line of code, or make deep customization trough script code.

The next step was to manage all the data captured by Kinect. On the web there are few libraries that allow a connection between OpenNI and UnrealEngine, one in particular was suitable for this project because was free and open source: NIUI. NIUI simply traces the joints position and rotation and give them to UDK in order to animate a 3D model or rotate the camera. The data transmission is almost real-time, with low data loss, and acceptable precision.

[1] `http://www.unrealengine.com/`

During design process it was decided to not implement walking gesture inside the virtual environments. As discussed in Oshita (2006), moving around 3D world start to be hard when it becomes bigger than the motion capture real area. Moreover, for the users the virtual experience effects in loss of natural senses that must be replaced with visual feedback from the system. Yan et al. (2004) proposed an innovative alternative to joypad or joystick for moving a character in a tridimensional environment. Even though the solution seems work there are no evidences to quantify how good it is, especially in a MoCap context. In conclusion, it was designed to not allow a walking gesture but to use keyboard buttons to control camera movement in the virtual space.

As possible solution to give a visual feedback to actors we suggested to show a 3D model representation of the actor in the virtual world. In Oshita (2006) users feedback are limited by visual elements from the virtual space, one above all, a 3D puppet model animated by the user. Therefore, Kallinen et al. (2007) demonstrated that let users have feeling of presence improve their immersion. Even though this approach is confined by the authors only in entertainment areas, the shown results are interesting and can be applied either outside the game context. The experiments conducted measures the "degree of presence" by attention and emotions calculated values. Eventually, they demonstrate that 1st person view generate higher presence that 3rd view, or rather show a 3D model that move according to user commands. However, to keep this project as general as possible it was decided to support both of these approaches allowing the operator to choose between them in base of actor needs or requests.

In conclusion, was suggested to use another open source library, named Fubi, to recognize complex actors gestures. Fubi allows developers to create personalized actions via xml file that are later recognized by the library's algorithms and may trigger events in the virtual world. This library and its components will be discussed in detail later.

4.1 Scenarios

There were created two different maps with simple interactions to present a functional prototype at the end of the development process. Therefore, to improve the immersion sensation, each map scenarios was designed to combines Kinect and four screens to create a semi-realistic scene where actor can interact, simulate gestures and see the environment reacting consequently. Actions, that can be performed by the actor, are limited due to Kinect deficiencies and the testing purpose of this project.

4.1.1 Farm Map

The first map created is set in an abandoned farm in a world full of zombies, with a barn and a jeep (Figure 4.1). The character first mission is to escape alive form the farm.

This map includes two main scenes:

Figure 4.1: Overview of farm map[Screenshot taken with Unreal Editor].

- The actor is in front of the barn door and he opens it with a kick to enter into the barn.
- The actor finds an old shot gun into the barn and wait in front of the door for the zombie invasion. A zombie starts to walk and eventually turn facing the actor. He shots the zombie and escape out of the barn.

4.1.2 Jungle Map

The second map is set in a jungle. There are a river, a bridge, some wood crates, two soldiers and one tank. The objective this time is to kill the enemy soldiers and blow up the tank (Figure 4.2).

This map provides two main scenes:

- The actor is detected by guards so he starts shooting them. He is finding cover behind crates.
- The actor places the bomb close to the tank and run away. At certain point there is an explosion that destroys the tank while the actor is thrown to the ground.

4.2 Video Sketches

At the beginning of design phase, was necessary to create video sketches to present the idea to all the stockholders involved in the project. A video sketch gives a short view of

Figure 4.2: Overview of jungle map [Screenshot taken with Unreal Editor].

the project purpose. It doesn't need to have perfect animations or high quality pictures in it. It will not show how the final version of the project since it is created at the beginning of the process, instead, it expresses all the important concepts at the base of the project idea [Zimmerman (2005)].

This fast prototyping tool is directed to whom want to know the general idea without going deep into details. Moreover, it is fast, low cost and allows to discover issues already at the beginning of the process.

For this project was used Microsoft Paint to create the slides and Movie Maker to put them in a video sequence. To keep it simple and understandable every slide comes with text caption that describes the meaning of what is represented in it.

In short, the following steps were followed to create the final presentation flow:

- Introduction: where it was explained the system components (Figure 4.3(a)) and where it will be placed the MoCap actor (Figure 4.3(b)).
- Simply explanation of how virtual world is projected on screens (Figure 4.4(a)) and how it is possible to switch between different maps (Figure 4.4(b)).
- Description of simple interactions that actors can perform with the system, such as kick barn door (Figure 4.5(a)) and bomb explosion (Figure 4.5(b)).
- Conclusion of the presentation showing the actor leaving the scene (Figure 4.6).

In the next sections it will be presented in detail all the hardware and software parts discuss above.

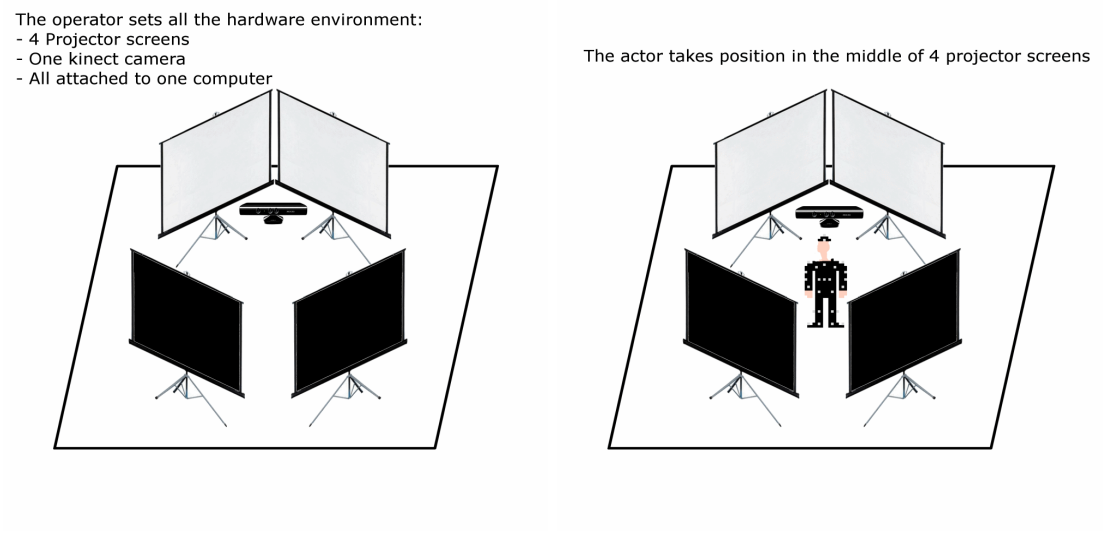


Figure 4.3: Video sketch frames to describe system configuration [Designed with Microsoft PowerPoint].

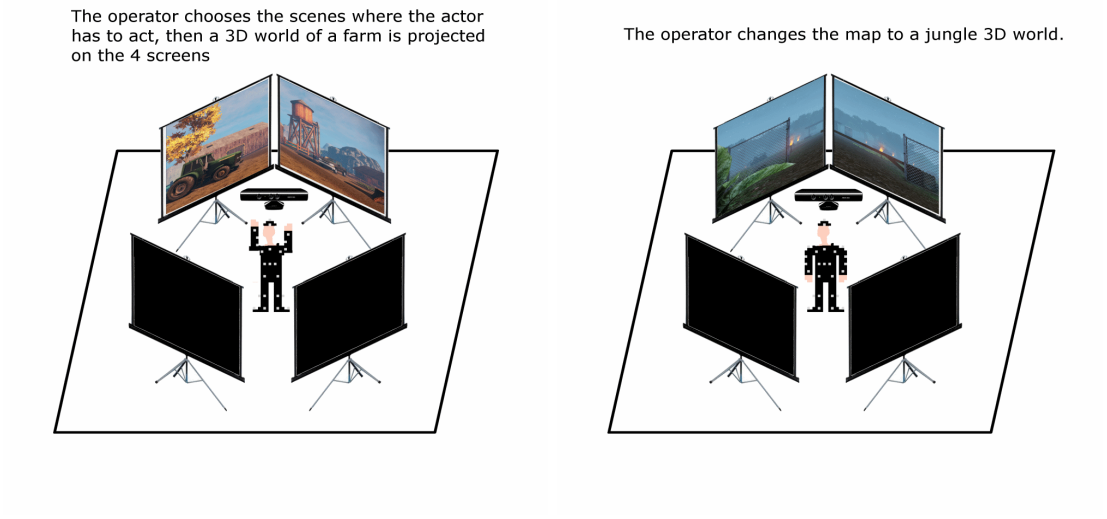


Figure 4.4: Video sketch sequence frames showing how map is projected on screens and showing change map passages [Designed with Microsoft PowerPoint].

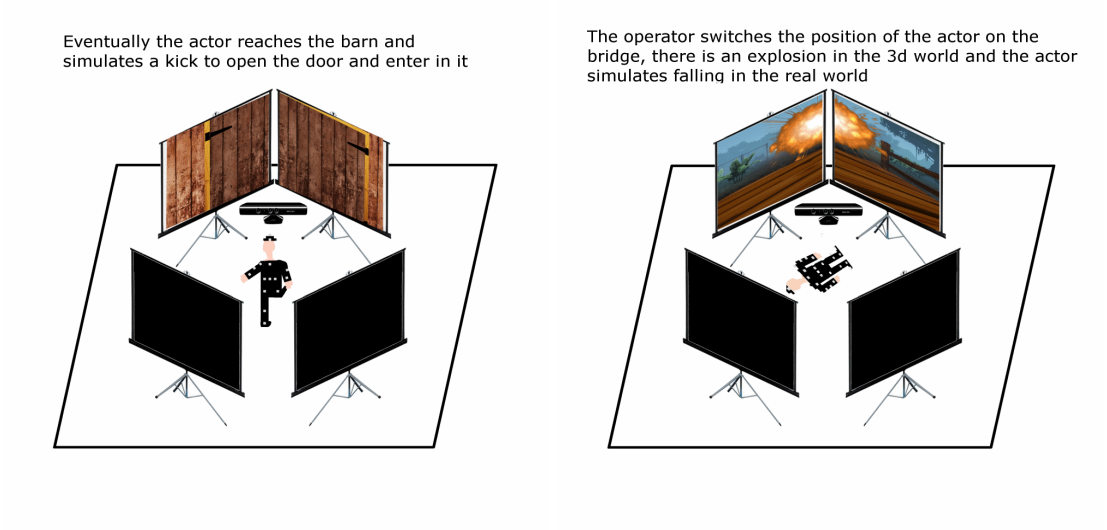


Figure 4.5: Video sketch frames showing how actor can interact with the system [Designed with Microsoft PowerPoint].

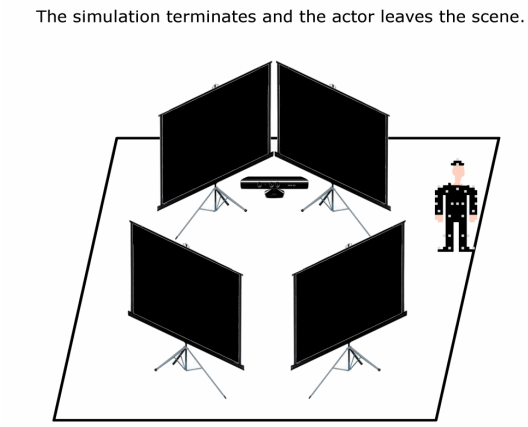


Figure 4.6: Video sketch frame showing how to conclude a system session [Designed with Microsoft PowerPoint].

5 System

Nowadays Virtual Reality environments have been used in different areas for different purposes. Nevertheless, create wide and realistic 3D scenarios with traditional approach is hard to achieve, instead, using game engine frameworks is more simple and fast. Game engines provide a complete and user friendly platform to develop your own VR project.

According to [Shiratuddin and Thabet (2001)] using a game engine is a valid alternative to CAD or other graphical software for the following reasons:

- Low-cost for development.
- Built-in collision detection, audio management, light effects.
- Possibility to use real-world textures to make the scenario more veridical.
- Artificial Intelligence interactions ready to use or create.

5.1 Unreal Development Kit

In the last years Unreal Engine has been one of the most used and appreciated game engine[1]. It was developed by Epic Games in 1998 for the first-person shooter video game Unreal. It has a C++ core and now it is at its third version, with the fourth in developing.

This engine is portable and easy to use. Moreover, the engine comes with several tools and it is provided with the last DirextX and OpenGL graphic libraries. Programmers save time using Unreal Engine integrated features, such as light and physic management, 3D acceleration, user friendly modelling editor and so on. Unreal Engine is object oriented and it makes easy to extend its functionalities without the need of detailed programming knowledge or deep modification into the code. Therefore, with this game engine it is possible to build a complete videogame without writing a single line of code. The complete framework that include the engine and all the tools is called Unreal Development Kit (UDK). UDK is a free for non-commercial purpose framework, while you must pay 25% of royalty to Epic Games for using it in a commercial product. It is composed by two main components:

- Unreal Editor
- UnrealScript

[1] `http://www.unrealengine.com/`

The editor provides graphical User Interface for easy and fast developing and prototyping [Guldbrandsen and Storstein (2010)], while the Java-like scripting language, Unreal Script, is easy to use, well documented, and extremely powerful though. Programmers can create new classes by inheriting from the base ones of the engine adding new features or modifying the aspect of a game.

> *"In the balance[2], the Unreal engine is slightly slower, has more sophisticated graphics, and is probably an easier environment for the inexperienced game programmer."*
>
> [Lewis and Jacobson (2002)]

UDK was chosen instead of other engines for the free access to plenty of expandable features, starting from the possibility to build a complete 3D world in a really short time [Mooney (2012)], to a fast integration of 3D assets already available at the company.

In conclusion, UDK is used not only for creating video games thanks to its flexibility and hundreds of features accessible to not expert developers. Indeed, UDK is often used in other fields:

Robotic. Fore example in [Zaratti et al. (2007)] they use the engine to create a robot simulator environment;

Architectural Design. More easy and fast for prototyping then CAD software [Shiratuddin and Fletcher (2007), Johns and Lowe (2006)];

Medicine. UDK could be used as educational background in medicine sector like in [Marks et al. (2007)] where the engine is extended to simulate surgical training applications;

School. Involve students into project more close to modern technologies such as 3D or Virtual Reality may have effect of their learning curve, indeed, in [Price (2008)] they propose an alternative approach to learn physic using the Unreal Engine;

and plenty more sectors where UDK can be exploit. For a more deep and explanatory knowledge of UDK the [Mooney (2012)] book can be consulted.

5.1.1 Unreal Editor

The editor tool (Figure 5.3) is the most important component of UDK due to its role as game development "interface". Even though Epic Games gives the possibility to create your personal video game just using the proprietary script language, the editor helps the artists to focus more on the visual aspects, such as the creation of game's levels placing 3D objects around the virtual world, while lets the programmers operate on game's logic.

Create from base a 3D environment with realistic aspects is really complex, expensive and requires significant amount of time [Trenholme and Smith (2008)]. Nevertheless, using

[2]Compared to Quake Engine

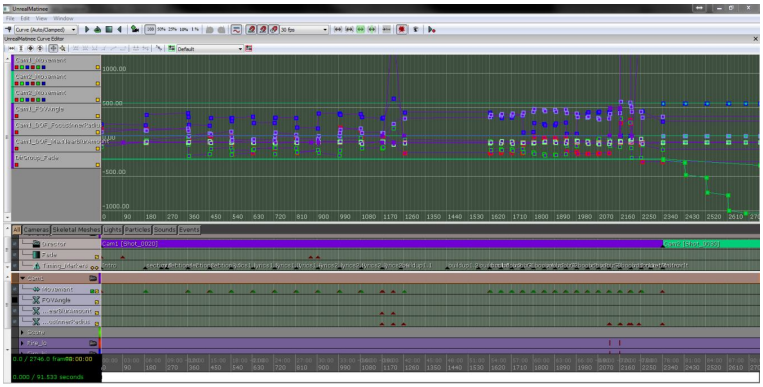


Figure 5.1: Matinee tool, Unreal Editor utility to manage cameras movements, animations and effects [(2013*g*)].

the editor instead of a commercial computer graphic software, such as 3D Studio Max or Maya, makes more faster the process of level design and logic management. Unreal Engine, for example, automatic renders 3D models and lights, auto-handling physics and has scalable performances. In addition, it provides small utilities built-in. For example, within the editor is possible to create/modify materials on rigid or skeletal 3D models, to manage animations or sound sequences, to create cinematic scenes by placing cameras around the level and control their movement or graphic effects (Figure 5.1), or to develop complex AI and navigation systems.

However, one of the most useful utility, inside the editor, is 'Kismet' (Figure 5.2). It is simple and powerful tool, because it integrates a graphical state diagram interface which gives an easy control of all the logic in the game. Even not expert developers can create complex interaction in the level without modify the source code. In addition, it is possible to create new objects to use in kismet with the script code.

Kismet interface has a main window worksheet where the user places five basic element sets:

Action. Set of possible modification at level gameplay, like change position of a model or activate/deactivate a light.

Condition. Execute actions only if a certain parameters are fulfilled, like if a boolean variable becomes true.

Variable. Float, integer, boolean or complex object representation.

Event. Activated by some occurrences in the game, like using a switch.

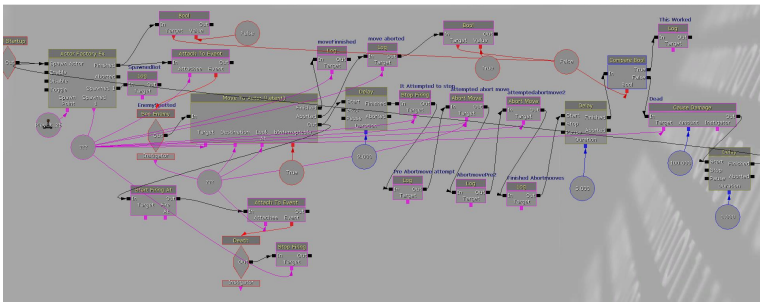


Figure 5.2: Kismet tool for managing game logic like trigger events or play sounds [(2013*h*)].

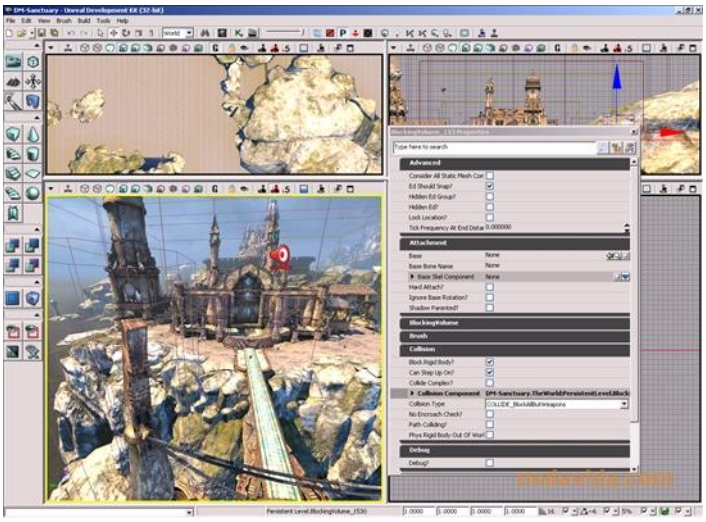


Figure 5.3: Unreal Editor window [(2013*i*)].

Comment. To keep the diagram more readable it is possible to insert comment or group different object together. Eventually, developers combine and link all this elements together without write a single line of code.

5.1.2 UnrealScript

UnrealScript is a high level scripting language for manipulating (almost) all the aspects of the Unreal Engine. Programmer can reach straight their objective modifying or inherit only the base class they need, without affecting the rest of the code. Indeed, the script language has object oriented hierarchy system, delegates and interfaces. This is the real script power because programmers can use it to develop a complete video game, with original gameplay or jus keep it simple and conventional.

All the classes inherit from the *'Object'* class, overwriting, if necessary, the base functions or variables. Some base classes are declared 'native' where the code is not visible or

editable since it has been written in C++ by the engine programmers. Moreover, it is possible to include external DLL library written in C++ and use their functions into the script code, even if under certain conditions and limitations.

5.2 Kinect

Kinect[3] camera is an advanced device provided by Microsoft for the new generation of Xbox consoles. Basically it revolutionized the approach to gaming, introducing a new way to interact with video games without the use of a physical controller. "*Games are more amazing when you are the controller*" is the slogan used by Microsoft at its launch on the market. Indeed, to use this new generation of video games you just need to move, jump or shake your arms to fully control the game (Figure 5.5(b)).

In addition, Kinect is provided with a USB cable and a power adapter to extend the portability, thus, it soon became available to use it with personal computer. Basically there are two drivers that anyone can install to control Kinect with a windows machines:

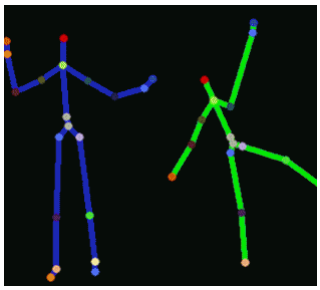
- Prime Sense with OpenNI
- Microsoft Kinect SDK
- Open Kinect Project

With those tools it is possible to get both the RGB image and the Depth image, for a complete 3D vision of a scene. The RGB camera acts like a normal webcam, taking the 2D view of the environment, while the Depth camera is composed by an IR (Infra-Red) light emitter and an IR receiver. The IR emitter projects all over the scene invisible lights dots (Figure 5.4(a)). The IR receiver calculates the distance between the dots and the Kinect camera. Kinect uses this distance to build a 3D image of all the scene in front of it (Figure 5.4(b)). Once acquired the entire environment Kinect uses proprietary algorithms to recognize and keep tracking persons when they appear in front of the camera. Plentiful applications use this elaborated information for different purposes [Giles (2010)]. The sensor can track up to six players and 20 joints for each player, such as the head joint, the left and the right arm joints, etc. The device is also furnished with a microphone and a motor to move up and down the camera.

Kinect was chosen for tracking the MoCap actor and recognizing his gestures. Despite this game device is not accurate like the normal MoCap camera, the project doesn't request so high level of precision due to its test purpose. As further improvement it is possible to substitute the Kinect with the motion cameras and consequently change part of the code.

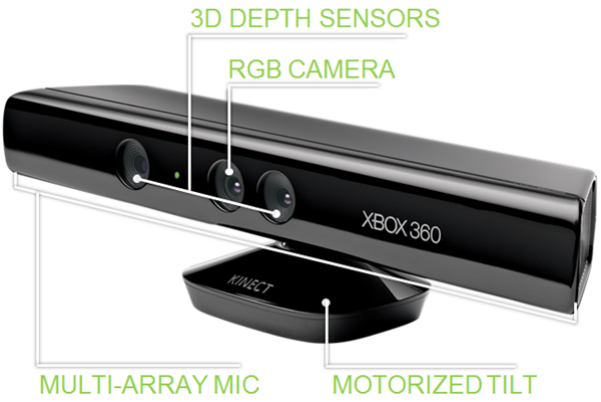
[3] `http://www.xbox.com/en-US/kinect`

(a) (b)



(c)

Figure 5.4: IR, Depth images and skeleton tracking of Kinect camera [(2013*j*), (2013*k*), (2013*l*)].



(a)

(b)

Figure 5.5: The Kinect camera device and an example of Kinect interactions [(2013*m*), (2013*n*)].

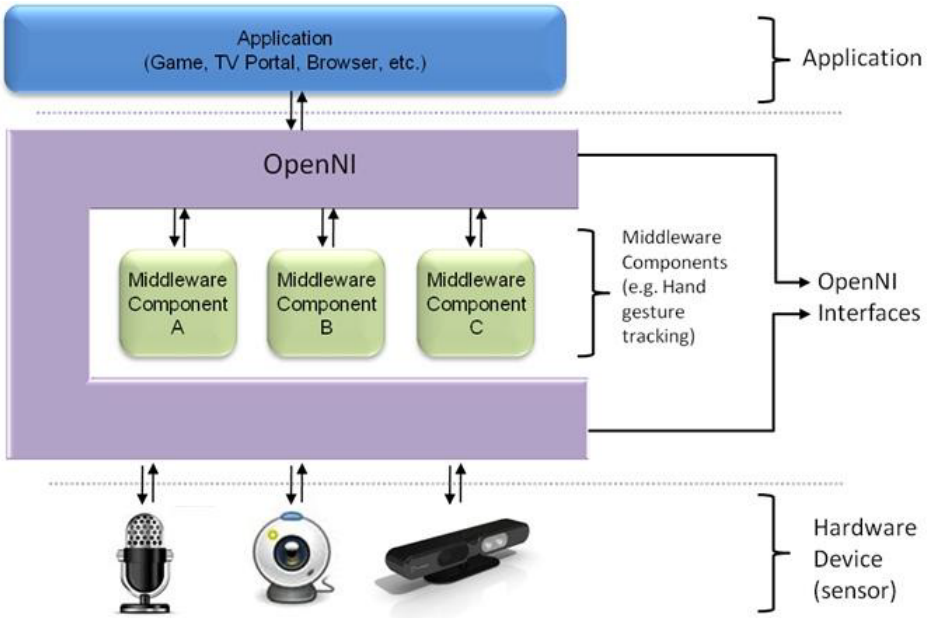


Figure 5.6: Three-layered view of OpenNI Framework [(n.d.)].

5.2.1 OpenNI/NITE

Each of the Kinect drivers proposed above offers different API interfaces, programming languages and community support. It would be impossible to establish which one is the best. Consequently, OpenNI[4] was preferred because it offers a C++ simple API and a furnished documentation. Even through it doesn't reach the level of features implemented by Microsoft SDK or Open Kinect Project, it allows a fast and easy management of basic Kinect 3D view skills. Moreover, Microsoft SDK could not be used with full features on the Kinect device available at the company due to a versions incompatibility. The SDK works only with the "Kinect For Windows" version devices that is different from the "Xbox Kinect" version owned by the company.

OpenNI is a not-for-profit organization involved into compatibility and interoperability improvement of Natural Interaction devices, applications and middleware [Villaroman et al. (2011)]. Basically it provides a cross-platform framework to develop applications that use Natural Interactions, such as the voice or body gestures. OpenNI interfaces serve as connection bridge between the hardware to recognize gestures, like the Kinect camera, and the final application. It allows the user to develop regardless of the sensor or middleware providers (Figure 5.6).

One of the middleware module for recognizing gestures and human skeleton is NITE, created by PrimeSense[5]. It is a set of drivers and algorithms to use in conjunction with OpenNI. NITE use proprietary formulas to achieve basically two objectives:

[4] `http://www.openni.org/`
[5] `http://www.primesense.com/solutions/nite-middleware/`

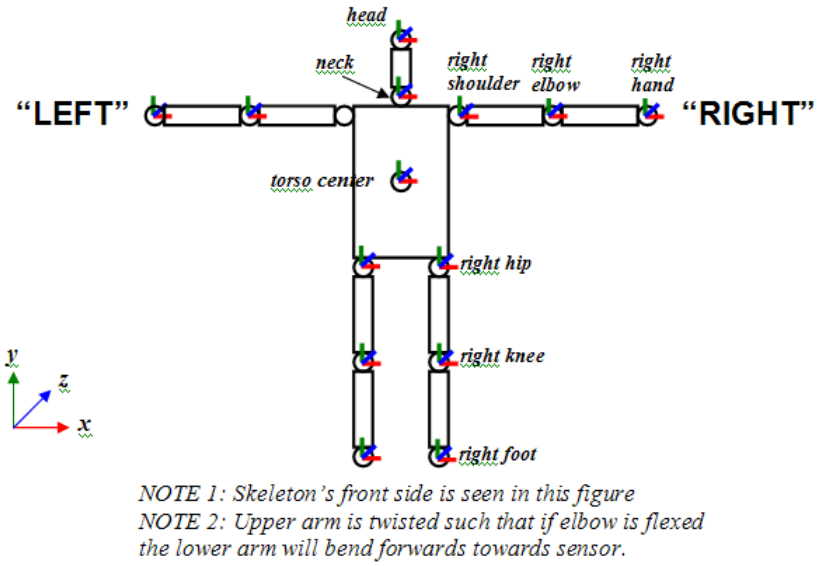


Figure 5.7: Human skeleton represented by NITE Framework [(2013*o*)].

- Recognize a person in the scene, and memorize it and keep it even if he/she left the scene.
- Tracking the skeleton joint of the person previously recognized (Figure 5.7).

NITE is composed by two layers: the algorithms to process raw data from the sensor and extrapolate actor's position in the 3D environment; the controls are a set of high level procedures to analyse the hand points generated by NITE Algorithms, to identify specific gestures, and to provide a set of UI controls that are based on these gestures.

5.2.2 NIUI

Once defined the sensor and the framework it was necessary to find a solution to integrate Kinect gesture tracking into Unreal Engine. There was only one library that uses OpenNI APIs in conjunction with UDK: NIUI.

In spite of NIUI was not updated and in beta version, it is open source and quite well documented. Therefore, was necessary to modify the library source code and adapt it to the new version of the OpenNI framework and NITE drivers. Basically this library acts like a "bridge" between Kinect and UDK to:

- Initialize the Kinect device and open a preview window showing the different images from RGB and depth cameras.
- Provide internal delegates called when a user is detected or lost.
- Give easy interface to get all tracking data, such as joint position and orientation.

5.2.3 Fubi

To give actors a more immersive experience, it was planned to recognize gestures that trigs actions in the virtual world. NIUI alone cannot satisfy that request, it is necessary to hard code a function that recognize gestures just analysing raw data from the device. This solution is both inefficient and really hard to apply especially when the gesture is complex.

According to that was necessary to find another solution. Therefore, Fubi[6] library was a more reasonable way to solve the problem. It is an open source DLL that with a set of C++ classes recognize gestures performed by users tracked with Kinect.

In short, Fubi allows programmers to use C++ code or XML file to recognize four gesture categories:

- Static postures.
- Linear movement.
- Combination of postures and linear movement.
- Complex gestures.

Static posture and linear movement track the position and orientation of the joints; the first recognize just static position of the person such as 'arms crossed' or 'left leg up', while the second captures simple joint movements speed and direction, such as 'raise left arm'. Furthermore, Fubi combines static postures and dynamic movement into combinations and complex gestures. With the latter the user can dictate time constraints on the gestures, so it is possible to build complex patterns where defined sequence of action must be executed in a certain amount of time to be recognized. For a deep explanation about this gesture patterns and how them are recognized, [Kistler et al. (2010)] can be consulted.

There are two ways to define gestures: through C++ code, using Fubi interfaces and classes, and through an XML file structured as follow:

[6] `https://www.informatik.uni-augsburg.de/en/chairs/hcm/projects/fubi/`

<!ELEMENT JointRelationRecognizer(Joints+)>

Define a simple 'distance' relation constrain between two or more joints, with minimum and maximum distance values.

<!ELEMENT JointOrientationRecognizer(Joints+)>

Define a simple 'angle' relation constrain between two or more joints, with minimum and maximum angle values.

<!ELEMENT LinearMovementRecognizer(Joints+,Direction?,Speed?)>

Define a simple 'movement' relation constrain between two or more joints, where is possible to set the direction (above x,y or z axe) and the speed values.

<!ELEMENT PostureCombinationRecognizer(State+)>

This is more complex element composed by 'States' where each of them could have one or more of the previous relation constrains. Moreover, to each state the user can set for how long the user should be in that position since the gesture is recognized

If the programmer wants more accurate and flexible control over the gestures, Fubi provides gesture recognizer interface

```
class IGestureRecognizer
{
public:
        virtual ~IGestureRecognizer() {}

        virtual bool isRecognizedFrom(FubiUser* user) = 0;
        virtual IGestureRecognizer* clone() = 0;
};
```

for linear or angular movement and the class *'FubiRecognizerFactory'* with the function *'getCombinationName'* to define the states sequence.

In addition, Fubi can track hands and count the number of fingers showed by the user (Figure 5.8), however, it was not taken in consideration for the project.

In conclusion, Fubi framework is an easy and intuitive library to create and recognize new and complex gestures. It is also compatible with OpenNI drivers and ensures a good quality results. Indeed, [Kistler et al. (2011)] was conducing experimental studies for an interactive storytelling scenario, they get 97% of their gestures (65 out of 67 gestures) recognized by Fubi over 18 participants.

5.3 SofTH

Natively UDK supports not multiple screens. SoftTH[7] library solves this limitation. It can span video output on multiple screens (Figure 5.9), even with different resolution or disposition (landscape, portrait, etc.). Moreover, SoftTH supports multiple graphics cards and different kind of screens, from normal LCD to video projector.

This library is really easy to use. To use it is necessary to copy two files in the folder of UDK engine binaries, one is the DLL and the other is the configuration file. To set on how many screens the video should be split and with which resolution it has necessary to modify the configuration file. In addition, the user can configure other parameters such as the quality of the video, any margins up or beside the screens, and so on.

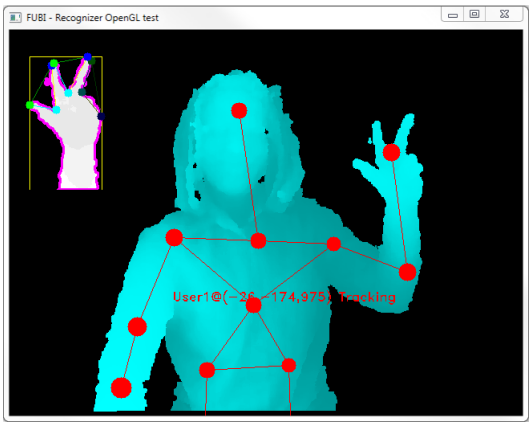


Figure 5.8: Fubi finger counting feature [(2013*p*)].

Figure 5.9: Example of SoftTH library application [(2013*q*)].

[7] `http://softth.com/`

6 Implementation

Microsoft Visual Studio 10[1] IDE was mainly used to manage two projects: a DLL to connect Kinect and UDK and a set of classes written in Unreal Script language. These classes managed the logic behind the virtual environment while there was used the Unreal editor to create the 3D world and place the models inside it. Unreal Editor was also useful for managing animations and event without adding more script code.

There is no user interface, except for the command line panel provided by the engine and there is no walking implementation because the test map didn't require it. Moreover, no improvements to efficiency or memory management were made to NIUI library, due to concentrate more on functional implementation. Only one actor is allowed in the environment.

In the test case, the MoCap actor will be surrounded by the screens and he/she will stand in front of the Kinect camera. Meanwhile, the operator will run the Unreal Engine. Once the level is loaded, jungle or farm map depends which one was set into the configuration file, the system will provide the following functionalities:

Camera Mode. The operator can decide to immerse the actor in a 360° panoramic view where the virtual world is projected on all the screens around. At the same time the operator can choose different view angles using the console or the configuration file. This mode ensures a quite realistic sensation to the actor, however, the screens position or the action to record by MoCap cameras require a different approach. Therefore, the operator can easily press a button on the keyboard to change the camera mode to the normal view. Consequently, the 3D model of the actor, if it is showed, can rotate according to the front camera. While two screes will remain black the other two will project only the frontal view of the actor. When the acto is rotating to the right, depends of the angle of rotation, the screen more distant on the left will became black and vice versa if the actor is rotating to the left. The screen that was previously dark will be activated. In addition, the camera view will rotate not only to the left or to the right, but also up or down, it can lean to the left or to the right side due to actor's head inclination angle. The latter feature is available only in the normal mode.

Positions. Operators or designers will use the editor to place special object called "Position point" in a map. When the engine is running, actor virtual model can be moved along these position objects. This is useful when the map is really large and is necessary to change area due to MoCap's recording needs.

[1] `http://www.microsoft.com/visualstudio/eng/products/visual-studio-overview`

Virtual Model. Kinect device will keep tracing of the MoCap actor and, if requested, a 3D model may be shown on the map. The model's head, arms and legs will follow the real actor movements. The torso is static due to avoid annoying visualization issue, for example, sometimes it happened that the camera view was showing the inside of the model's head or body. It is possible also to show only the model without cast any shadows.

Gesture Recognition. Kinect will trace actor's movements and gestures. All the gestures that Kinect should capture must be listed inside an xml file that the engine will load at the start up. To connect these gestures to specific actions or events in the level it is necessary to modify the script and the code for the designed reaction. As future possibility we suggested to manage the creation of reactions with the editor and not modifying the code.

Command Line. Even if it is not present a user interface, there is a built in function provided by the Unreal Engine: the command line. When the user will press the tab key it will show a green panel where it is possible to write the command's name to execute. Using this feature our project has provides functions to modify the camera view mode, move the 3D model, and so on. All the commands are listed later.

6.1 NIUI-Fubi Library Project

The version of OpenNI library used is "1.5.4.0", but on the web NIUI API is at the "1.0" release that is not compatible with it. Therefore, it was necessary to update the library compiling it with the new version, in addition, the library was improved deleting some part, by now, obsolete. First of all, it was added a support to 'auto-calibration' feature making useless assume the 'T' pose for being recognized by Kinect. Moreover, NIUI was originally saving actor joints position for a fast calibration, it was set on automatic without the need to save the position. Furthermore, there were some parts of source code not compatible with the new version, so it was necessary to fix them.

The NIUI library is divided into following parts:

Kinect connection. It is the collection of classes and functions that connect the DLL to OpenNI and NITE framework. It takes raw data from Kinect device and packs them into high level objects to easy manipulation.

Fubi integration. Since Fubi itself has components that communicate with Kinect, the two DLL, OpenNI and Fubi, cannot work together and they generate execution conflicts errors, thus, Fubi was integrated inside the NIUI library. So far, only classes that include the logic and algorithms to recognize gestures were kept while the others were deleted. In addition, some gestures were created adding new code, but some for others it was just created a new xml file for a fast test and fixing process. In Figure [6.1] is showed part of this file.

Unreal Engine interface. It is a set of high level functions to communicate with the Unreal Engine. It exposes all the API to manage the initialization of Kinect, the user calibration and the gestures recognition.

```
<?xml version="1.0"?>
<!DOCTYPE FubiRecognizers SYSTEM "FubiRecognizers.dtd">
<FubiRecognizers>
  <JointOrientationRecognizer name="LeanFront">
    <Joint name="torso"/>
    <MinDegrees x="15"/>
  </JointOrientationRecognizer>

  <JointOrientationRecognizer name="LeanBack">
    <Joint name="torso"/>
    <MaxDegrees x="-15"/>
  </JointOrientationRecognizer>

<!--GESTURE: AIMING RIFLE-->
  <JointRelationRecognizer name="LeftKneeInFrontOfRightKnee" visibility="hidden">
    <Joints main="leftKnee" relative="rightKnee"/>
    <!--cm in front of right-->
    <MaxValues z="-150"/>
  </JointRelationRecognizer>

  <JointRelationRecognizer name="LeftHandOut">
    <Joints main="leftHand" relative="leftShoulder"/>
    <MinValues y="-200"/>
    <MaxValues z="-300"/>
  </JointRelationRecognizer>

  <JointRelationRecognizer name="RightHandOverRightHip">
    <Joints main="rightHand" relative="rightHip"/>
    <MinValues y="0"/>
  </JointRelationRecognizer>

  <PostureCombinationRecognizer name="AimingRifle">
    <State minDuration="0.1">
      <Recognizer name="RightHandOverRightHip"/>
      <Recognizer name="LeftKneeInFrontOfRightKnee"/>
      <Recognizer name="LeftHandOut"/>
    </State>
  </PostureCombinationRecognizer>
```

Figure 6.1: Part of xml file to create new complex gestures in Fubi [Screenshot of gesture.xml file].

6.2 Unreal Script Project

Developing with Unreal Script is not the same of working with `C++` or `C#`. The script has its proprietary syntax and it can be complied only with Unreal Engine binaries. nFringe[2] plug-in for Visual Studio was used to edit and compile Unreal Script projects. The plug-in comes with many helpful features, such as auto-complete function and debugging tool.

The classes set (Figure 6.2) created with Unreal Script is composed by:

[2] `http://pixelminegames.com/nfringe/`

IMSMain. The main class where the engine is lunched. Basically this class loads the map and all the models inside, creates shadows and lights and the user interface. Moreover, it is responsible for the creation of 'controllers' and 'pawn' classes.

In this class the most of command functions are implemented, the NIUI core object is initialized and the preview window is loaded. The latter will show what Kinect camera is recording. The window is showing a RGB view plus a coloured layer of the actor currently tracked.

IMSPlayerController. All the objects inside a level are managed by controllers. IMSPlayerController is controlling the main 'pawn' or rather the virtual representation of the actor. It is the mainly responsible for the movement, rotation and gesture tracking. This class manages the system logic that makes possible the event triggering, such as the bomb explosion, and the gesture recognition.

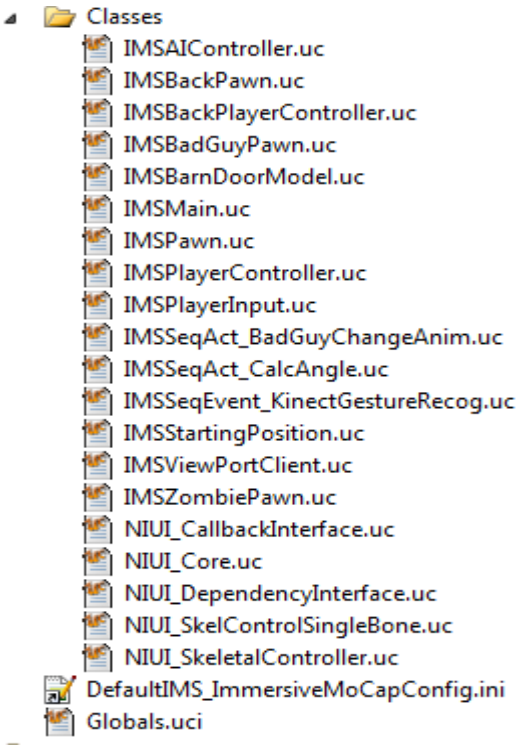


Figure 6.2: Unreal Script Project Hierarchy [Screenshot taken with Microsoft Visual Studio].

IMSPawn. This class represent the actor inside a level. In short, it initializes the 3D model, controls the camera movement with 'CalcCamera' function, sets all the parameters of the camera, such as the field of view, and updates the model movements based on joint positions get from the Kinect.

IMSPlayerInput. This class overwrites the basic one to manage the input channel. All the keyboard events are redirected here to avoid unexpected behaves.

IMSAIController, IMSZombiePawn, IMSBarnDoorModel, IMSBadGuyPawn. These classes represents the pawn and the controller for the zombie model and the barn door present in the farm map and the two enemy soldiers in the jungle map. Movements and reactions are managed by the controller while the pawn loads the 3D mesh and the attached materials.

IMSBackPlayerController, IMSBackPawn. Made-up pawn and controller classes to create a "simulated" 360° view. These classes don't have a lot of script code inside, however, they are necessary to manage an 'invisible' actor that always look behind the first character. It is necessary to have a camera attached to the back of the actor 3D model that show the back view.

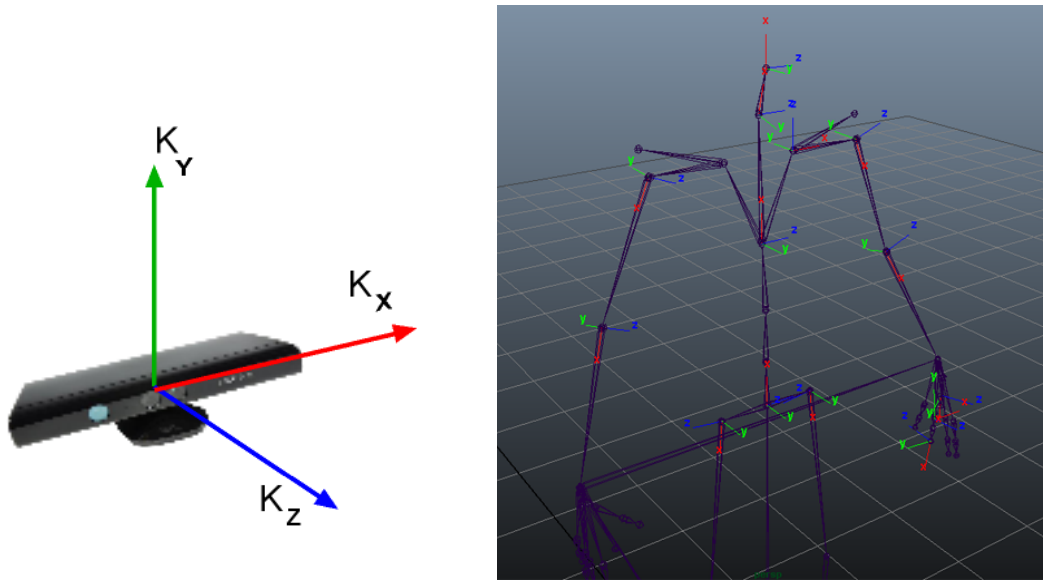
IMSViewPortClient. Important class that implements the back view for the 360° mode. It implements functions for switching between the 360 and normal mode thus it makes easy change the hardware setup without restating the engine.

IMSSeqAct_BadGuyChangeAnim, IMSSeqAct_CalcAngle, IMSSeqEvent_KinectGestureRecog. These classes create the elements to use in the Kismet editor. SeqAct classes represent Kismet actions while SeqEvent classes describe the events. In shorts, the two implemented actions play or stop the enemy soldiers' animations. The implemented event is created to easy develop the action/reaction logic, based on gesture recognition, directly from the Unreal Editor instead of using the script-driven approach inside the Player Controller.

IMSStartingPosition. This class creates objects that designers can place into the map directly from the editor. The purpose of this object is to insert positions around the level in which the actor can be moved. It makes the movements inside the map entirely script independent. Operators have to use the editor to place the objects and the console commands to change the position of the model.

NIUI_Core, NIUI_CallbackInterface, NIUI_DependencyInterface, NIUI_SkelControlSingleBone,NIUI_SkeletalController. All these classes were provided by NIUI framework and basically they use Unreal DLL linking feature to connect with the NIUI `C++` library. The 'Core' and 'DependencyInterface' expose the API to send and receive data from Kinect and Fubi algorithms. 'Skeletal' classes manage model bones link to OpenNI joints. Moreover, inside these classes there are high level functions that convert Kinect coordinates into UDK ones because axes and values have different configuration among these two systems (Figure 6.3).

Globals.uci This class is an utility file with pre-processor directives. There is just one for now that checks if there is or not the Kinect device installed on the machine were the system is compiled.



(a) Kinect (b) UDK

Figure 6.3: Axes representation [(2013*r*), (2013*s*)].

6.2.1 Configuration files

DefaultIMS_ImmersiveMoCapConfig.ini Unreal Engine flexibility is possible thanks to the configuration files system. Every class may have variables that can be changed before or even during the execution of a game, for example saving all the graphic settings. Therefore, Unreal Engine gives the possibility to isolate that values into configuration files. All the values inside these files are grouped by the name of the class in which they are declared.

For this project is available one configuration file. It contains the following sections:

[IMS_ImmersiveMoCap.IMSPlayerInput] Under this section there are all the key bindings and the relative command line function. There are all the keyboard's key command that if pressed will execute specific action into the level. It is useful if the operator wants to trigger events regardless what the actor is doing.

[IMS_ImmersiveMoCap.IMSMain] In this section the operator can add the 3D map names list that will be loaded before the system starts. Therefore, operators can switch to the next map without restarting the engine.

[IMS_ImmersiveMoCap.IMSViewPortClient] It contains the ViewType parameter that sets which camera view should be loaded at the start-up. It will be possible to change it at run time as well.

There is another parameter named OriginX_2 that sets the starting coordinate for the back camera view. This value is useful when the operators want to adapt the camera view in base of screens setup disposition.

[IMS_ImmersiveMoCap.IMSPlayerController] In this section the operator can set values to vary how fast the camera will rotate in the normal view, before the engine starts.

[IMS_ImmersiveMoCap.IMSPawn] In this section are grouped the parameters for setting camera rotation speed, field of view angle for front and back camera, and the degree of rotation for the back camera. The angle of rotation for the back camera is useful to easy manipulate, also at runtime, the 360° view in base of screens setup.

6.3 Ureal

Unreal editor was used together with the script project. Therefore, using the editor and script code as hybrid approach has speeded up the development process [Guldbrandsen and Storstein (2010)]. The last version of the Unreal Engine compatible with NIUI library is the UDK-2012-01 (January 2012).

The training time spent to learn most of the Editor's functionalities was worth as training to fast manipulate the map models and the Kismet tool. Therefore, in less than two months of work there were build two different map with several 3D models inside, such as trees, rocks and houses.

After that, to create the logic of each level were used both code and console commands. On one side actions and events were triggered by code, on the other by command line that activates kismet action flow.

The potential of mix these two approaches is showed, for example, by the position object. Positions are created to allow operators to move the actor's model from one point to another one without the need of real movement on the scene. Before the engine starts, the operator will use the editor to place position objects, declared in a script class, directly in the map with easy drag and drop action. When the engine will start all these objects will be loaded in a list. When the operator presses a button on the keyboard the 3D model of the actor will move in the next position found in the list. Moreover, the position objects have also orientation parameter to give max personalization. Therefore, operators can decide where the actor's model has to look at when it will be moved to that position.

6.3.1 Command Line

There are now listed all the commands available in the console. Some commands are associated to a button on the keyboard. The operators have to write the command name

in the game console, or just press the relative button, if it is associated, and the related action will be triggered.

MoveForwardMod(), MoveBackwardMod(), TurnLeftMod(), TurnRightMod() These commands allow to move the 3D model, if it is showed, forward, backward, turn left, turn right. The operator can also use ↑ ↓ ← → buttons on the keyboard.

Exit() It closes the engine and the Kinect preview window. `"Escape"` button can be used as well.

SwitchPosition() It will move the 3D model to the next position in the list. `"P"` button is associated to this command.

ChangeMap() It will change to the next map on the list. It can be used `"M"` button.

Recalibrate() It resets the user tracking, accordingly, Kinect will have to recalibrate the user. `"R"` button is associated to this command.

RotateCamera() It is a fast command to turn the view of 180°. It will switch the front and back cameras. This command has `"T"` button associated.

OpenBarnDoor() It will start the opening animation for the barn door, only for the farm map. The operator can press `"O"` button as well.

SwitchDoorMesh() It will change the mesh of the door barn and consequently the opening animation related to it, only for the farm map. `"D"` button is associated to this command.

SwitchCameraView() It will switch between the two view modes: 360° or normal. The button `"C"` can be used as well.

TogglePlayerMesh() It will hide or show the actor virtual 3D model. It can be trigged using also the `"H"` button.

TogglePlayerShadow() It will hide or show the 3D model shadows, if they are activated.

RotateCameraAngle(angle) With this command is possible to manually rotates the front camera of specified angle. If is activated the 360° mode the back camera is also rotated. For a left rotation the value passed should be negative while positive for right rotation.

setFOV(fov), setFrontFOV(fov), setBackFOV(fov) These command set the field of view angle (from 0 to 170 maximum) for respectively the front and the back cameras.

setBackAngle(angle) It sets the rotation angle only for the back camera, relative to the front camera. As default, the angle is 180 for a complete 360° view but sometimes it may be useful to set with a different value due to screens position or a MoCap scene requirements.

6.3.2 Kismet

As explained in previous sections, the UDK framework, specifically the Unreal Editor, has plenty of tools to speed up the development process. One of this is Kismet editor. Moreover, Kismet tool was useful for creating event/reaction logic flow and to control some models' animations inside the two maps created previously.

The animations of the zombie model in the farm map were created with Kismet. In Figure 6.4 the event is represented as a trigger object placed inside the barn just close to the door. When the actor will touch it, two actions will be activated. If the actor will leave the barn the animation will stop and the zombie model will disappear. Two different events will

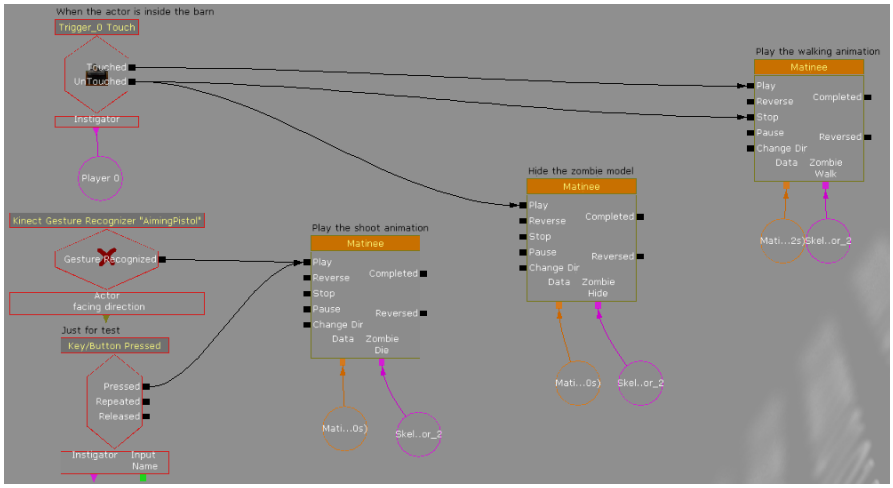


Figure 6.4: Part of kismet interface for the farm map; is represented how is managed the animation of the zombie and the shoot event that will let it fall [Screenshot of Kismet tool window].

activate the zombie model falling animation. The operator has to press a button, in this case is the "space" key, to start the animation. Alternatively, the MoCap actor can do the "shooting" gesture to trigger it.

For the jungle map three action sequences were developed whit Kismet:

- When the actor is behind some wood boxes, Kismet will activate the animation flow. Two enemy soldiers will be created in front of the actor and they will stand up and crunch continuously. It is a simple simulation of a fire fight against the actor. As showed in the Figure 6.5, it is relative easy to create animation object ("Actor Factory" elements) and animate it ("Change animation of Bad Guy" elements). However, it was more long and complicated calculating how to turn the enemy models to face the actor.

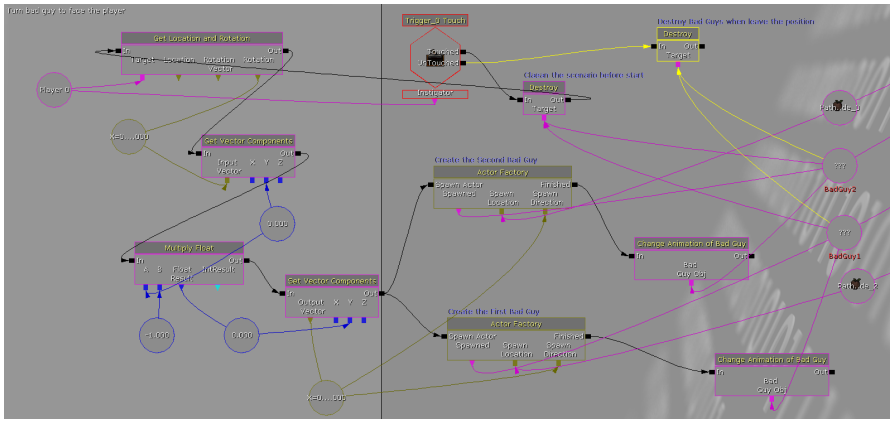


Figure 6.5: Kismet frame view for the creation of the two enemy soldiers and start of their animations [Screenshot of Kismet tool window].

- Another animation may be activated by two events, similar to the farm map ones. When the actor will do the shooting gestures or the operator will press the "space" key it will start the animation. The enemy models will die like they have been shoot by the actor. As showed in the Figure 6.6, the animations will start only if the actor is facing the enemy model. To accomplish this feature, it was used the "Calculate angle between vectors" element. It takes the direction of where the actor is looking ("Get Location and Rotation" element), as a 3D vector, and compares the angle between this direction and the enemy position ("Compare Float" element). If the difference is less than 5° the animation will continue.
- In conclusion, there is another animation sequence for this map (Figure 6.7). It is created to test the actor's reaction to unexpected events. Moreover, this animation starts after the operator pressed the "B" button. Alternately, this animation starts also when the actor avatar is in a specific position for a defined time. In both cases, a virtual explosion will be triggered, with fire, smoke and sound effects, and objects that fly away. The more times the operator presses the button the bigger the

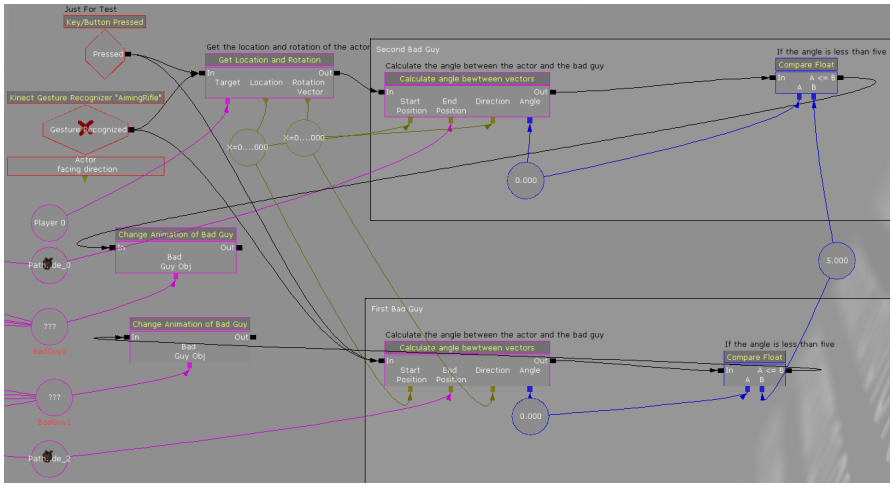


Figure 6.6: Kismet flow for shooting and die animation [Screenshot of Kismet tool window].

explosion will be. The impact of the explosion depends on how realistic or strong the MoCap scene has to be.

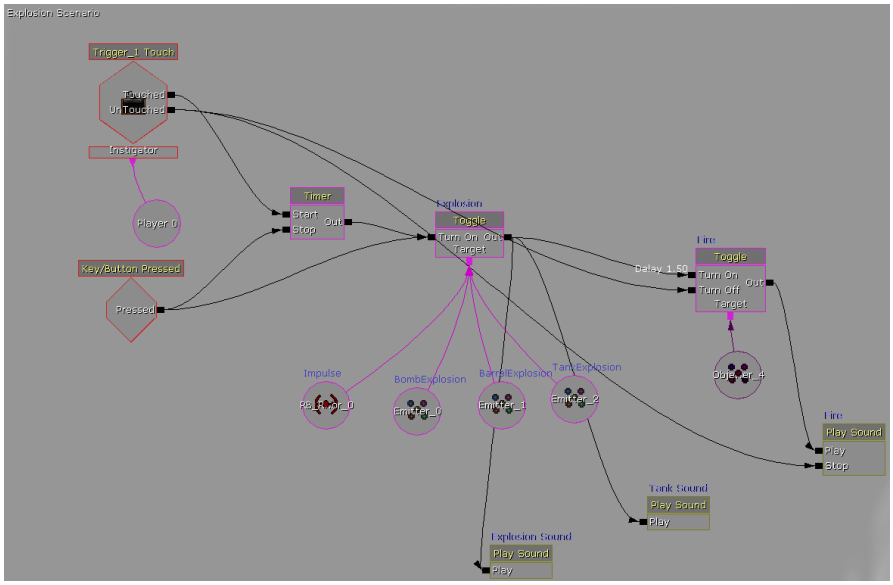


Figure 6.7: Kismet view for the explosion sequence [Screenshot of Kismet tool window].

7 Experimentation

7.1 Set-Up

Hardware

The setup environment included:

Computer Intel i7 processor at 3.7GHz; two Nvidia Geforce GTX 670 in SLI; 16GB of RAM; 4GB of graphic memory.

Operating System Windows 7 Professional.

Screens four Acer screens with HDMI connection at resolution of 1920x1080 pixel for each, for a total of 7680x1080 pixel (Figure 7.1).

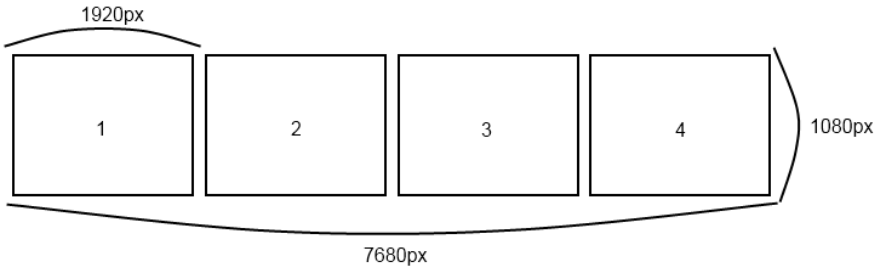


Figure 7.1: Screens resolution configuration [Made with Microsoft PowerPoint].

Configuration

Each of the following configuration include disposition of Kinect camera and screens:

1. Two screens are attached and placed in front of the actor while the other two are attached behind. The Kinect camera is between the two screens in the front. This setup may be useful when the scene requires actor to react of something important in the front, or in the back, of the level, while the left and the right side are in the blind view of the actor (Figure 7.2).

Figure 7.2: First screens setup[Picture taken with Reflex Camera].

2. This setup is a variant of the previous, the screens in front and on the back are not attached anymore but there is a gap among them that could be changed based on MoCap requests. In this case the Kinect device will be still fixed between the two front screens so the actor still have to face them, but this configuration may be used when events or actor reactions are needed not exactly on the front but shifted on one side (Figure 7.3).

Figure 7.3: Second screens setup[Picture taken with Reflex Camera].

3. Two screens are attached and placed in front of the actor while the other two are attached but placed on the actor's right, or left side. The Kinect camera is between the two screens in the front. This setup was thought to give flexibility on the scene, for example, the back view is not necessary while the actor is facing the two screens

in the front and something happens on his right and he has to react after seeing it at the corner of his eye (Figure 7.4).

Figure 7.4: Third screens setup[Picture taken with Reflex Camera].

4. In the last solution the four screens are placed on each side of the area, one front, one on the left side, one on the right side and one in the back. This setup is thought principally for projector screens and not normal LCD, but was used anyway to show how it could work with the system (Figure 7.5).

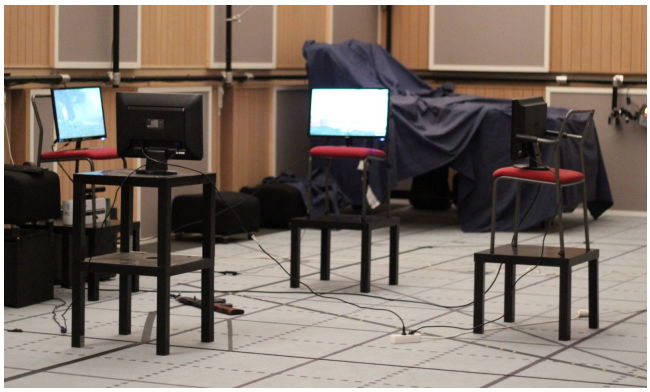


Figure 7.5: Third screens setup[Picture taken with Reflex Camera].

7.2 Execution

The entire system was placed in the same area where the MoCap scenes are recorded. It was asked to give a direct feedback through a thinking aloud technique to all the

persons that were using the system. As enounced in [Krippendorff (2005)], the validation of a project could be expressed by explaining the design and implementing it. The same approach was used for this system. The solution was designed, implemented and used by a pair of persons to demonstrate that it was possible to realize such framework; on the other hand, this non-scientific approach cannot provide a definitive and functional solution to the stakeholder. In other words, this simple executions of the system was meant to show the idea and fix some bugs encountered during the development process.

It was not necessary to involve neither a high number of persons nor real MoCap actors, rather there were called to use it only who was directly affected by the system. The final feedback was useful to demonstrate how good the initial idea was and to show that was possible to implement it. The main focus of this entire project was, indeed, to show the technologies necessary to create a virtual, immersive and portable framework in a MoCap context rather than conduct scientific tests.

The execution background was set with all the lights turned on, while extension cables were used to connect the screens due to keep the area uncluttered.

The experimentation process was divided into two main steps:

As first step were called 1 researcher and 1 director. The researcher was directly involved in the development idea, while the director leads the company were this project was developed. It was explained to both to say what they were thinking loudly while they were using the system, as part of the think aloud technique. Both were put into different simulated situations, for each configuration setup presented before, there were used two different maps (Jungle and Farm) and two view modes (normal and 360). This combination set is resumed in the Table 7.1. For each row in the Table 7.1 were asked to perform two actions for each map (Table 7.2). All verbal and visual feedbacks from the persons called to use the system were written on a notebook. Every comments and reactions for each action performed by them was noted and written down. Once all the actions requested were performed the execution ended. Afterwards, the list of what the users liked or disliked was used to ask them questions about the prototype.

Based on the answers and the annotations reported during the first phase, a resume was created of what in the prototype was working well and what not. These positive and negative aspects are all explained into the results section.

As second step, the initial prototype was improved to solve parts of the negative aspects observed during the first step. The final version of the system was eventually proposed to the entire group working in the company. The prototypes functionality was demonstrated and people were given the possibility to try it. This demonstration collected, more feedback and impressions of the system.

MoCap actors were not asked to use the system. The project is still “immature”, and is not ready to be introduced in real situations.

Setup	Map	View mode
1	Jungle	Normal
1	Jungle	360
1	Farm	Normal
1	Farm	360
2	Jungle	Normal
2	Jungle	360
2	Farm	Normal
2	Farm	360
3	Jungle	Normal
3	Jungle	360
3	Farm	Normal
3	Farm	360
4	Jungle	Normal
4	Jungle	360
4	Farm	Normal
4	Farm	360

Table 7.1: Configuration setting solutions.

Map	Actions
Jungle	Door kick Zombie shoot
Farm	Shooting Explosion

Table 7.2: Preseeded actions according to the map.

7.3 Results

Setup 1 With normal LCD screens there was an inconvenient gap between two attached screens making less realistic the scene displayed.

The stakeholder called to use this first setup were giving back feedback to improve the entire project. First of all they have advanced some doubts about 3D model of the actor. Showing the model and animate it as a puppet maybe could be unrealistic and not really helpful for actors, moreover, as initial implementation the model was moving all the body parts like torso and head, creating sometimes annoying view of "inside" the model. Consequently, there was suggested to show only arms and legs on the screens. Moreover, the system was able to cast or not the shadows and show or hide the 3D model.

Some gestures were not well recognized so it will need some fix and improvement on Fubi xml gestures file. For the 360° mode the field of view was too low so it was found 110° angle value enough good to show the most of the scene without too much distortion. Other

minor corrections were noticed about both the map and how they look, for example, some objects were not in the right position, or shadows had graphical defects.

At the end, this screens position has proved to be quite good for almost all the scenes and maps. Frontal view was essential to give actors realistic vision of what is going to happen in the scenario and react naturally. The back view as well was useful; it gives the user sensation of being immersed in a virtual world, even if the Kinect cannot trace all the gestures. The latter works only with 360 view mode. The normal mode can be used but it will just make two screens black while the others are showing only what is in front of the actor.

Setup 2 As the previous one this solution had more or less the same results. In addition, the gap between the front, or back, screens became too wide when used with a scene just in front of the actor. Therefore, even though the Kinect is still in the middle of this space, the scenario can be shifted more on the left, or on the right, according to how the MoCap actor should react.

Setup 3 Using this option as first setup fit not because it was not possible to change the back camera angle for 360° view mode. Therefore, the screens on the left, or on the right, in base of chosen configuration, were showing the back view. This approach was creating confusion to the user. This issue was solved making the angle variable by command line.

Although, the actors will not see the scenario behind them. This setup was more immersive than the others, due to the position of the screens more close to each other. The small gap between the screens gave a more wide view of the world. Moreover, Kinect camera was more precise and was more accurate for recognizing gestures.

Setup 4 As explained before this setup doesn't really worth with LCD screens, but it was used as well to collect the impression from the users. Basically, it is the one that better fits with the idea of complete 360° immersion. However, Kinect camera cannot be placed anymore under the one screen in the front. This is due to screen resolution, for example, if the resolution is 7680 pixels in width the front camera will be the half of that, so, 3840 pixels. Moreover, the Kinect is placed in the middle of the front camera to face the actor, it will be 1920 pixels shifted on the right, which is like 45° moved on the right, between the first and the second screen. To fix this issue it is now possible to set the initial angle of the front camera. In this way, the actor will start the scene facing just the front screen.

With the normal view mode this setup was still useful because of its flexibility. Therefore, when the actor is turning, one or two screens could be activated while the other could be left black. In addition, if there will be used large projector screens it will be possible to keep active more than two screens increasing the field of view angle and consequently the overall immersion feeling.

After this first session, a meeting with others professional inside the MoCap company was set to present them the project. There were presented the advantages of the system, how it works, and the limitations. Moreover, the participants of the meeting were also invited to try the environment, in order to discuss more about the privileges and defects. Overall, all the solution proposed were well accepted by the stakeholder even though with some issues discovered during the fist execution phase or problems caused by the Kinect's low precision. Moreover, with all the previous configurations, the Kinect was able to keep trace and not lose the user between 1.2 and 3.5 meters distance range from it, this limitation was also increased by the length of screen cables. These conditions both limit the moving area for the actor avoiding a high freedom of movement.

Despite all the limitations, as shown in the Table 7.3, users were feeling directly involved in the scene. They felt like acting inside the world and reacting consequently. This bidirectional interactions approach showed how make the users reactions inside a virtual immersive environment more realistic and natural.

Action/Event	Expected	Reaction
Door kick	The actor has to simulate a kick	Users simulate a kick and see the door in the virtual world that opened
Zombie shoot	The actor has to simulate a gun shoot	The operator moves the virtual character inside the barn and when the users saw the zombie coming in front they consequently react shooting at it
Shooting	The actor has to kill two enemy that are shooting him back	Users were naturally crouching and hiding in order to avoid the enemy fire, at the same time they were shooting the enemy back
Explosion	The actor has to place the bomb on the tank while it will explode suddenly	After the users placed the imaginary bomb in front of them, the operator trigged the explosion event, thus causing an immediate reaction of the candidates, that were jumping back or covering the face with the arms

Table 7.3: Table of reaction examples collected during the execution session.

8 Conclusion

In the previous sections was described the development process to create an immersive environment for a MoCap scenario based on open source software and a cheap and flexible hardware configuration. The initial idea provided four screens all around MoCap actors, on which an interactive 3D world was projected, and one camera to track all the actor movements. The final framework was composed by 4 lCD screens, a Kinect camera and a personal computer to connect and manage all these components. The choice of these hardware set was made to keep the cost down, beside, it was eventually easy to install, even in the existing scenario of a Motion Capture company. UnrealEngine and NIUI library were used to create and animate the virtual world. The first is free to use for non-commercial purpose while the latter is completely open source. the open drivers of OpenNI was used to connect the Kinect camera to the game engine, while Fubi algorithms were integrated in the system for recognizing complex gestures. All these different software components were eventually connected in a unique framework, easy to configure and setup thanks to a basic user interface for helping non-professionals to use it.

The framework was used by non-professional in real MoCap company. Thanks to the feedbacks received there were consequently fixed some bugs. As result of the executions conducted, MoCap shots scenario changed. Before using the system the area was free of screens and computer, except scene objects, useful to help actors to act. After installing the system, in the recording area were added the screens and the computer, moreover, actors could not freely move around due to Kinect vision range limitations. Despite this issue, users were feeling more involved in the scene. They were interacting with the system when some events were triggered in the virtual world.

Several technical innovative proposes come out as consequences of this research work. If the framework could be improved with adequate equipment, from the screens to the Kinect device, it could be possible to use in a real Motion Capture scene. Indeed, use wide screens instead of small LCD can reduce the flexibility of the system but it may help actors to feel more immersed into the virtual world. Moreover, substitute Kinect with MoCap camera for sure will improve gesture recognition accuracy, but consequently it will be necessary to update the framework for making it communicates with cameras. Despite its positive response, the system, as it is, needs several revision passages to fix some issues that limit its effectiveness. For example, it was able to trace only one actor at the same time; it was not implemented a solution to transfer actors walking into the virtual map without the operator intervention; the Kinect camera was not accurate and sometimes was missing the position or completely losing the actor while he was walking out of its view spot. On the other side, the system was used by non-professional and it collected important impressions and results.

In conclusion, after the experimentation, it was certainly faced a cheap test tool using simple and open source components. Even though the system was not completely ready for working in real contest with professional actors, it has advanced a concrete idea how to improve MoCap actor confidence with scenes. The idea is to increase efficiency, in terms of how natural were the gestures, and consequently reduce post processing time. A quantitative evaluations of this tool was out of its scope. It is planned to accomplish this step in a further research.

9 Future Work

The system was set and used inside the company, and several issues and relative improvements were gathered. As first consideration the main purpose of this project is not to give a final solution to improve quality of MoCap scenes, but to propose different approaches using interactive environment, evaluate them and report all the results. Therefore, it is expected a further quantitative and complete test sessions, where the system can be tested by several real MoCap actors guided by a scientific team.

Besides that, during development process was created a semi-final product that later could be used by operators in the company without knowledge of what is inside it. To keep the development time shorter and to have fast results from the test this side was not completely satisfied, therefore, it will be created a user manual that will come with the project. The latter will guide the operators on how to setup the system, screens, Kinect, computer, and how to create new content within the Unreal editor tool.

Still on the subject of usability, it will be provided a new user interface, which could be created again with Unreal Engine as well, to easy start the program and modify its properties, instead of changing them by configuration files. Moreover, the associated user interface may allow switching among different maps, starting positions, enable animations, and enable effects without using the command line console. Basically the framework should provide:

- Creation of 3D maps and integration with 3D models easy with UDK.
- Creation of simple action-reaction pipeline with Kismet(UDK).
- Easy modification of values, such as angle and FOV without recompile the code, therefore allowing multiple kind of screens setup.

A part from the usability, the system has encountered several limitations. One above all is the Kinect camera. It is good to recognize movements of one or two persons in bright space and without markers. However, it becomes less precise with annoying errors on joint position when the user start to move faster, or go out from its visual. The 3D puppet model will move in a unnatural way don't respecting the real actor movements if Kinect camera lost the user. Moreover, the camera rotation will be not precise and sometimes lost user rotation angle in the normal view mode. Rotation lost will give to the actor the wrong view of the world. A solution to these issues will be using the MoCap cameras that are already on the scene. These cameras are definitively more accurate than Kinect with rarely loss of data, therefore, the system will response in more realistic way and recognize faster and without false positive results any gestures performed by the actors.

We noticed that 4 simple LCD screens were not enough to create a real immersive environment. The screens were too small and the user was not able to see all the detail inside the virtual world. We suggested to use projector screens instead. However, for a full immersion experience it is possible to create virtual dome like in [Hirose et al. (1993)] or give to actors virtual reality glasses or head mounted display. Any of these approaches must not interfere with MoCap cameras and body markers. These solutions will make actors feel as if they are inside the world, consequently help them to act more naturally. Using MoCap cameras will keep the cost of the prototype down since they are already present on the real set.

Most of the MoCap scenes are not performed by only one actor, but by a group of several professionals that sometimes have to interact with each other. The system will be more useful if it can handle more than one user at the same time. MoCap cameras will also overcome this problem. Interactive environment that react in real time according with several inputs from different actors is a challenge that certain worth the effort.

10 Acknowledgments

First I would like to thank my advisor Daniel Kade for his support and helpful guidance, it was long and hard work but eventually really fulfilling. Moreover, my gratitude goes out as well to John Mayhew and all the staff of Imagination Studios for being so nice with me and treat me like one of the group, I really appreciate that and it let me apprise Swedish work and life style.

I want to remember the great job done by professor Muccini, he gave me the opportunity to start this experience that made me grow both professionally and as person. He has always believed in the GSEEM program and in all the students who are part of it.

I wish to thank my family, my mom, my dad, my sister, my brother in low and my two marvellous nepotes, my two cousins Nanni and Marco, whit whom I spent my childhood, to be my anchor when I am far from my home country.

I can't miss to remember all my close friends in Italy, Daniele, Domenico, Antonella, Andrea and Luigi, always ready to party and hang out with me.

I will always bring with me all the best moments spent during my erasmus experience in Västerås, where I met extraordinary people like Andrea, Michele, Angie, Edoardo, George, Adil, Eljar, Arvin, Maurizio, Vicky, Reyes, and all the friends from different part of the world. Thank you guys to be part of my life and it is also through you that I could improve my English, without your help I could never write my thesis.

A particular thanks goes to Alessio and Manuel with whom I spent the most of my academic route in Italy and Sweden, they were always ready to help me whenever I needed it more.

Lastly a special thanks is for my sweet love Nawel, thank you for correcting my English errors and give me the motivation to realize this important achievement. You were always there for me when I needed company and encouragement.

Bibliography

(2013*a*).
URL: *http://www.digitalperformance.it/?p=473*

(2013*b*).
URL: *http://stoffwechsel-rees.de/motion-capture*

(2013*c*).
URL: *http://accad.osu.edu/ bwindsor/mocap2/calibration/calibration.html*

(2013*d*).
URL: *http://www.c-motion.com/v3dwiki/index.php?title=AMASS_Installation*

(2013*e*).
URL: *http://www.xsens.com/en/general/mvn*

(2013*f*).
URL: *http://www.slashgear.com/most-awesome-battlefield-3-simulator-ever-in-history-18188609/*

(2013*g*).
URL: *http://www.unrealengine.com/showcase/film_television/mudbrick_creative_marketing_1/*

(2013*h*).
URL: *http://forums.epicgames.com/threads/603843-Kismet-Abort-Move-action-does-not-work*

(2013*i*).
URL: *http://blog.teachbook.com.au/index.php/2011/05/unreal-development-kit/*

(2013*j*).
URL: *http://graphics.stanford.edu/ mdfisher/Kinect.html*

(2013*k*).
URL: *http://www.creativeapplications.net/news/kinect-opensource-news/*

(2013*l*).
URL: *http://it.emcelettronica.com/photogallery-con-kinect-della-microsoft-xbox360*

(2013*m*).
URL: *http://dotnetcampania.org/blogs/paolopat/archive/2011/08/17/kinect-il-dispositivo-l-installazione-dell-sdk-ed-i-primi-passi.aspx*

(2013*n*).
URL: *http://www.informaticaeasy.net/hardware/altro/1006-e-record-10-milioni-di-unita-kinect-e-altrettanti-di-giochi-venduti-nel-mondo.html*

(2013*o*).
URL: *http://www.triballabs.net/2011/06/kinectapis/*

(2013*p*).
URL: *https://www.informatik.uni-augsburg.de/en/chairs/hcm/projects/fubi/*

(2013*q*).
URL: *http://softth.com/?page_id=16&cpage=1*

(2013*r*).
URL: *http://www.depthbiomechanics.co.uk/?p=2496*

(2013*s*).
URL: *http://forums.epicgames.com/threads/908427-important-questions-about-rigs-and-animation*

(n.d.).

Barbič, J., Safonova, A., Pan, J., Faloutsos, C., Hodgins, J. and Pollard, N. (2004), Segmenting motion capture data into distinct behaviors, *in* 'Proceedings of Graphics Interface 2004', Canadian Human-Computer Communications Society, pp. 185–194.

Giles, J. (2010), 'Inside the race to hack the kinect', *The New Scientist* **208**(2789), 22–23.

Gleicher, M. (1998), Retargetting motion to new characters, *in* 'Proceedings of the 25th annual conference on Computer graphics and interactive techniques', SIGGRAPH '98, ACM, New York, NY, USA, pp. 33–42.
URL: *http://doi.acm.org/10.1145/280814.280820*

Guldbrandsen, K. and Storstein, K. (2010), Evolutionary Game Prototyping using the Unreal Development Kit, PhD thesis, Norwegian University of Science and Technology.

Hasler, N., Rosenhahn, B., Thormahlen, T., Wand, M., Gall, J. and Seidel, H. (2009), Markerless motion capture with unsynchronized moving cameras, *in* 'Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on', IEEE, pp. 224–231.

Hilton, A. (1999), Towards model-based capture of a persons shape, appearance and motion, *in* 'Modelling People, 1999. Proceedings. IEEE International Workshop on', IEEE, pp. 37–44.

Hirose, M., Yokoyama, K. and Sato, S.-i. (1993), Transmission of realistic sensation: Development of a virtual dome, *in* 'Virtual Reality Annual International Symposium, 1993., 1993 IEEE', pp. 125 –131.

Jacobson, J. and Preussner, G. (2010), Visually immersive theater with caveut 2.5, *in* 'World Conference on Educational Multimedia, Hypermedia and Telecommunications', Vol. 2010, pp. 3817–3825.

Johns, R. and Lowe, R. (2006), 'Unreal editor as a virtual design instrument in landscape architecture studio', *Journal of Design Research* **5**(2), 172–187.

Kallinen, K., Salminen, M., Ravaja, N., Kedzior, R. and Sääksjärvi, M. (2007), 'Presence and emotion in computer game players during 1st person vs. 3rd person playing view: Evidence from self-report, eye-tracking, and facial muscle activity data', *Proceedings of Presence* pp. 187–190.

Kistler, F., Endrass, B., Damian, I., Dang, C. and André, E. (2010), 'Natural interaction with culturally adaptive virtual characters', *Journal on Multimodal User Interfaces* pp. 1–9.
URL: *http://dx.doi.org/10.1007/s12193-011-0087-z*

Kistler, F., Sollfrank, D., Bee, N. and André, E. (2011), Full body gestures enhancing a game book for interactive story telling, *in* M. Si, D. Thue, E. André, J. Lester, J. Tanenbaum and V. Zammitto, eds, 'Interactive Storytelling', Vol. 7069 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 207–218.
URL: *http://dx.doi.org/10.1007/978-3-642-25289-1_23*

Krippendorff, K. (2005), *The Semantic Turn: A New Foundation for Design*, Taylor & Francis.
URL: *http://books.google.se/books?id=xwINxyVBeuIC*

Lengyel, J. (1998), 'The convergence of graphics and vision', *Computer* **31**(7), 46–53.

Lewis, M. and Jacobson, J. (2002), 'Game engines', *Communications of the ACM* **45**(1), 27.

Maiocchi, R. (1996), 3-d character animation using motion capture, *in* 'Interactive computer animation', Prentice-Hall, Inc., pp. 10–39.

Marks, S., Windsor, J. and Wünsche, B. (2007), 'Collaborative soft object manipulation for game engine-based virtual reality surgery simulators'.

Moeslund, T. and Granum, E. (2001), 'A survey of computer vision-based human motion capture', *Computer Vision and Image Understanding* **81**(3), 231–268.

Mooney, T. (2012), *Unreal Development Kit Game Design Cookbook*, Packt Publishing Ltd.

Oshita, M. (2006), Motion-capture-based avatar control framework in third-person view virtual environments, *in* 'Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology', ACE '06, ACM, New York, NY, USA.
URL: *http://doi.acm.org/10.1145/1178823.1178826*

Price, C. (2008), 'Learning physics with the unreal tournament engine', *Physics Education* **43**(3), 291.

Rose, B., Rosenthal, S. and Pella, J. (1997), The process of motion capture: Dealing with the data, *in* 'Computer Animation and Simulation', Vol. 97, Citeseer, p. 2.

Shin, H. J., Lee, J., Shin, S. Y. and Gleicher, M. (2001), 'Computer puppetry: An importance-based approach', *ACM Trans. Graph.* **20**(2), 67–94.
URL: *http://doi.acm.org/10.1145/502122.502123*

Shiratuddin, M. and Fletcher, D. (2007), Utilizing 3d games development tool for architectural design in a virtual environment, *in* '7th International Conference on Construction Applications of Virtual Reality CONVR 2007'.
URL: *http://researchrepository.murdoch.edu.au/7266/*

Shiratuddin, M. and Thabet, W. (2001), Making the transition toward an alternative vr, *in* 'Conference on Applied Virtual Reality in Engineering and Construction. Applications of Virtual Reality: Current initiatives and future challenge CONVR 2001'.
URL: *http://researchrepository.murdoch.edu.au/7370/*

Silaghi, M., Plänkers, R., Boulic, R., Fua, P. and Thalmann, D. (1998), 'Local and global skeleton fitting techniques for optical motion capture', *Modelling and Motion Capture Techniques for Virtual Environments* pp. 26–40.

Trenholme, D. and Smith, S. (2008), 'Computer game engines for developing first-person virtual environments', *Virtual reality* **12**(3), 181–187.

Villaroman, N., Rowe, D. and Swan, B. (2011), Teaching natural user interaction using openni and the microsoft kinect sensor, *in* 'Proceedings of the 2011 conference on Information technology education', ACM, pp. 227–232.

Yan, L., Allison, R. and Rushton, S. (2004), New simple virtual walking method-walking on the spot, *in* 'Proceedings of the IPT Symposium', Citeseer.

Zaratti, M., Fratarcangeli, M. and Iocchi, L. (2007), 'A 3d simulator of multiple legged robots based on usarsim', *Robocup 2006: Robot Soccer World Cup X* pp. 13–24.

Zimmerman, J. (2005), 'Video sketches: Exploring pervasive computing interaction designs', *Pervasive Computing, IEEE* **4**(4), 91–94.