
**University Institute of Lisbon**

Department of Information Science and Technology


# MotionDesigner: A Tool for Creating Interactive Performances Using RGB-D Cameras


Filipe Miguel Simões Baptista


A Dissertation presented in partial fulfillment of the Requirements for the Degree of

**Master in Computer Science**


**Supervisor**

Prof. Dr. Pedro Faria Lopes, Associate Professor

ISCTE-IUL

**Co-Supervisor**

Prof. Dr. Pedro Santana, Assistant Professor

ISCTE-IUL


September 2015

*"The art challenges the technology, and the technology inspires the art."* [20]

John Lasseter

# *Resumo*

Desde há duas décadas que o uso da tecnologia em projetos artísticos tem proliferado cada vez mais, como é o caso das projeções interativas baseadas em movimento utilizadas em performances e instalações artísticas. No entanto, os artistas responsáveis pela criação destes trabalhos têm, tipicamente, de recorrer a especialistas em computadores para implementar este tipo de sistemas interativos. A ferramenta apresentada nesta dissertação, o *MotionDesigner*, pretende auxiliar o papel do criador artístico na conceção destes sistemas, permitindo que haja autonomia e eficiência no processo criativo das suas próprias obras. A ferramenta proposta possui um design orientado para estes utilizadores de modo a estimular e agilizar a criação autónoma deste tipo de obras e tem uma natureza extensível, na medida em que mais conteúdo pode ser adicionado no futuro. O software desenvolvido foi testado com bailarinos, coreógrafos e arquitetos, revelando-se como uma ajuda e um catalisador do processo criativo da suas obras interativas.

**Palavras-chave:** Design, Interatividade, Tempo-real, Audiovisual, Projeção interativa, Câmaras de profundidade, Kinect, Arte, Multimédia, Performance, Instalação.

# *Abstract*

In the last two decades the use of technology in art projects has proliferated, as is the case of the interactive projections based on movement used in art performances and installations. However, the artists responsible for creating this work typically have to rely on computer experts to implement this type of interactive systems. The tool herein presented, *MotionDesigner*, intends to assist the role of the artistic creator in the design of these systems, allowing them to have autonomy and efficiency during the creative process of their own works. The proposed tool has a design oriented to these users so that it stimulates and proliferates their work, having an extensible nature, in the way that more content may be added further in the future. The developed software was tested with dancers, choreographers and architects, revealing itself as an aid and catalyst of the creative process.

**Keywords:** Design, Interactivity, Real-time, Audiovisual, Interactive projection, Depth cameras, Kinect, Art, Multimedia, Performance, Installation.

# *Acknowledgements*

I would like to acknowledge Prof. Dr. Pedro Faria Lopes for supervising this work and Prof. Dr. Pedro Santana for his careful supervision and constant aid in the development of this project. I also want to acknowledge all the dancers, choreographers, architects and multimedia artists interviewed during the development process, specially Beatriz Couto, Júlio Núncio, Maria Antunes and Catarina Júlio for their valuable feedback.

I direct a special acknowledgment to Maria Antunes, the dancer which collaborated in the creation of the small interactive dance performance we prepared using the developed tool, and to the Information Sciences, Technologies and Architecture Research Center (ISTAR) at ISCTE-IUL for the continuous support and for providing the necessary conditions to implement and test this project.

# Contents

**Resumo** **v**

**Abstract** **vii**

**Acknowledgements** **ix**

**List of Figures** **xiii**

**Abbreviations** **xv**

**1 Introduction** **1**
1.1 Context . . . . . 1
1.2 Motivation . . . . . 2
1.3 Research Questions . . . . . 5
1.4 Objectives . . . . . 6
1.5 Research Method . . . . . 7
1.6 Document Structure . . . . . 8

**2 Literature Survey** **9**
2.1 Depth-Cameras Based Systems . . . . . 10
2.2 Interactive Projections . . . . . 10
2.2.1 Graphical Interactivity . . . . . 11
2.2.2 Sound Interactivity . . . . . 12
2.3 Tools for Creating Interactive Audiovisual Art . . . . . 14

**3 System Overview** **19**
3.1 User Interface . . . . . 19
3.2 Hardware Setup . . . . . 22
3.2.1 Kinect sensor . . . . . 23
3.3 Software Setup . . . . . 25
3.3.1 openFrameworks . . . . . 26
3.3.2 OpenNI & NiTE middleware . . . . . 27
3.3.3 openFrameworks Add-ons . . . . . 27

**4 Development and Implementation** **29**
4.1 Editing Studio . . . . . 30

4.1.1 Scenes Parameterizations . . . 33
4.1.2 Previewing the Projection Sequence . . . 36
4.1.3 Displaying the Scenes . . . 37
4.1.4 Projecting the Sequence . . . 39
4.2 Interactive Graphical Scenes . . . 41
4.2.1 Scenes Editing GUI . . . 42
4.2.2 Motion Capture Algorithms . . . 49
4.2.3 Particle System . . . 50
4.2.4 Drawing With Joints . . . 58
4.3 Interactive Audio . . . 62

**5 Evaluation and Discussion** **67**
5.1 Evaluation Method . . . 67
5.2 Results . . . 70

**6 Conclusions and Future Work** **83**
6.1 Conclusions . . . 83
6.2 Future Work . . . 85

**Appendices** **89**

**A GUI Design Sketches** **89**
A.1 Editing Studio . . . 89
A.2 Interactive Scenes . . . 90

**B Source Code Snippets** **91**
B.1 Interactive Scenes GUI . . . 91
B.2 NiTE . . . 94
B.3 Particle System . . . 95
B.4 Joints Draw . . . 99

**Bibliography** **101**

# List of Figures


1.1 *Pixel* Performance . . . 3
1.2 *Mind the Dots* Performance . . . 3

2.1 Interactive Projection Setup Scheme . . . 10
2.2 *Divided By Zero* Performance . . . 11
2.3 MotionDraw's User Interface . . . 15

3.1 System Setup . . . 20
3.2 MotionDesigner's Editing Studio GUI Layout . . . 21
3.3 MotionDesigner's Interactive Scene GUI Layout . . . 22
3.4 Kinect Sensor Hardware . . . 24
3.5 Kinect Sensor Inner Structure . . . 24

4.1 MotionDesigner's Software Architecture . . . 30
4.2 Editing Studio's Timeline . . . 31
4.3 Scenes Palette GUI . . . 32
4.4 Editing Studio's Timeline Object . . . 33
4.5 Timeline Objects Parameterization Diagram . . . 34
4.6 Editing Studio's Parameterizations Panel . . . 35
4.7 Editing Studio's Preview Player . . . 37
4.8 Editing Studio's Explore Button . . . 38
4.9 Copy/Paste Parameterizations Data Flow . . . 39
4.10 Editing Studio's Panels Distribution . . . 40
4.11 Second Window for Projection . . . 41
4.12 Interactive Graphical Scenes GUI . . . 42
4.13 Parameters Panel Iterations . . . 43
4.14 Color Picker . . . 45
4.15 Color Picker Experimentations . . . 46
4.16 Interactive Scenes Editing GUI (using *ofxUI*) . . . 47
4.17 Joints Selector Panel . . . 47
4.18 Particle System GUI with Joints Selector . . . 48
4.19 Particle System Example . . . 51
4.20 OF Functions Diagram . . . 53
4.21 Particle System Parameters . . . 54
4.22 Particles Color Interpolation . . . 55
4.23 Particles Motion Blur . . . 57

4.24 Particle System Kinect Screenshot . . . . . 59
4.25 Joints Draw GUI . . . . . 60
4.26 Joints Draw Kinect Screenshot . . . . . 61
4.27 Explore Audio Button . . . . . 63
4.28 Interactive Audio Scene GUI . . . . . 64
4.29 MotionDesigner Logo . . . . . 66
4.30 Sketch and Final Implementation Comparison . . . . . 66

5.1 Users Implementation Suggestions Graph . . . . . 72
5.2 Evaluation Question 1 Graph . . . . . 73
5.3 Evaluation Question 3 Graph . . . . . 74
5.4 Evaluation Question 4 Graph . . . . . 75
5.5 Evaluation Question 5 Graph . . . . . 76
5.6 Evaluation Question 8 Graph . . . . . 79
5.7 Evaluation Question 9 Graph . . . . . 79
5.8 Interactive Performance Rehearsal (Performer) . . . . . 81
5.9 Interactive Performance Rehearsal (User) . . . . . 81

# Abbreviations

| | |
|---|---|
| **OF** | **O**pen**F**rameworks |
| **IDE** | **I**ntegrated **D**evelopment **E**nviroment |
| **GUI** | **G**raphical **U**ser **I**nterface |
| **Mocap** | **Mo**tion **Cap**ture |
| **OSC** | **O**pen **S**ound **C**ontrol |
| **SDK** | **S**oftware **D**evelopment **K**it |
| **FBO** | **F**rame **B**uffer **O**bject |
| **XML** | e**X**tensible **M**arkup **L**anguage |
| **FPS** | **F**rames **P**er **S**econd |

# Chapter 1

# Introduction

## 1.1 Context

Since the 1950s a lot of artists and computer scientists started to use programming to create different art pieces and this is a practice that continues today thanks to an enthusiast group of creative programmers all around the world [3]. Nowadays the community of artists and programmers that create computer art is bigger than ever. It was in the 90s that the use of technology in contemporary art increased exponentially and a lot of interactive systems started to be developed as result of a collaboration between computer programmers/software engineers and artists from different fields like painting, dancing and cinema [36]. The term coined to describe what these programmers and engineers were doing is creative coding.

Creative coding is a term that was coined to distinguish a particular type of programming in which the developer pretends to create something expressive. It is used to create live audiovisuals, interactive projections for art performances or installations, interactive soundscapes, etc [4]. Programming such systems helps artists to lately produce real interesting works, but it represents a real challenge to programmers and software engineers.

Today there are many different tools and libraries that help programmers and engineers to prototype and develop interactive systems to embed in art performances or installations. Many of these libraries can be used within different toolkits. The most popular toolkits used nowadays by creative coders are openFrameworks (OF), Processing, Cinder, Max/MSP, etc [9] [12] [1] [5]. By integrating one of these toolkits and their libraries within a chosen Integrated Development Environment (IDE), like Microsoft Visual Studio for instance, a lot of possibilities can be explored and implemented in a much more prolific way.

Despite having many tools that support the implementation of interactive audiovisuals generated by the computer and affected by another person (through the use of a video camera, for instance [24]) there are not many tools, already implemented, that help the creative people to produce these interactive works without having to program them or without depending on the technical knowledge of another person.

## 1.2 Motivation

Despite the challenge of creating interactive systems for an art performance or installation, many programmers and engineers collaborated with different artists to produce them over the past years. Today there are many art works based on the interaction between a person, typically an art performer, and a computer system [26] [11] [24]. Two of the most explored types of human-machine interaction in an artistic work are installation art and interactive performance, where special cameras, called RGB-Depth cameras (or simply depth-cameras), are used to capture the position and movement of the user/performer, allowing him/her to directly interact with the art piece.

Figures 1.1 and 1.2 show two different art performances which used interactive computer systems, both capable of producing computer graphics that would be manipulated by a dancer or performer. In both of these systems, the body movement data was captured by RGB-Depth cameras.

FIGURE 1.1: "Pixel", a dance piece where the graphics projected on the wall and floor are transformed by the dancers position and movement [11].

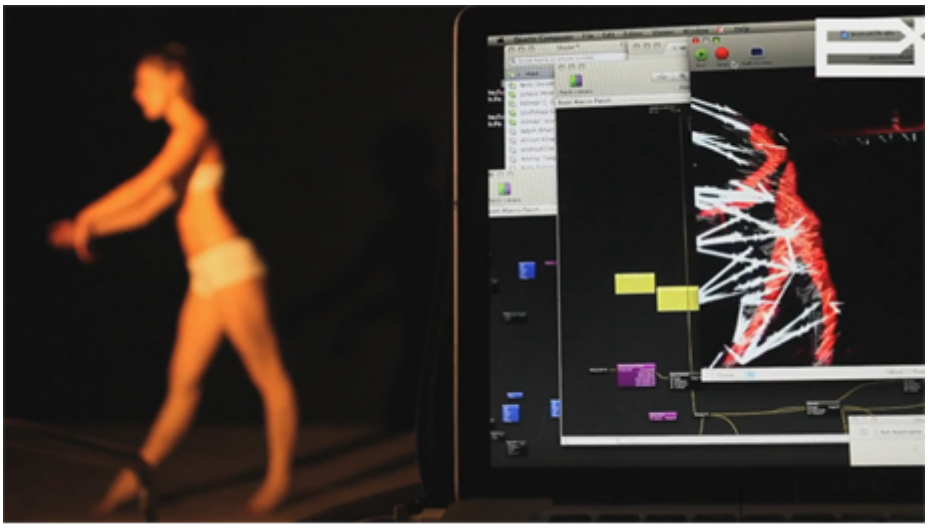


FIGURE 1.2: "Mind the Dots" a piece where a solo dancer interacts with virtual dots creating abstract graphic shapes generated by a computer [7].

Interactive systems like those used on the shows pictured on Figures 1.1 and 1.2 are complicated to implement, not because a special hardware is needed (in these cases only a depth-camera connected to the computer and calibrated correctly), but mainly because of the programming complexity behind such systems. To surpass this challenge the artists often collaborate with skilled programmers and software engineers to implement the system they want for a specific art piece. However, despite the interesting aspects of having different people from different work fields

collaborating with each other, the fact that they need to interact so intricately can be seen less positively, since it often works as an undesired inter-dependency, typically for the creative person. Due to this dependency issue, the creative people also need a tool to help them create these type of works autonomously. Such dependency questions arise because the creative process behind any work needs the ideas and their concretion to be a quick and fluid action-reaction process.

Having the artists dependent of computer programmers to successfully implement their own ideas for an interactive system will make the creative process longer and not as fluid as should be. This happens because the artists need the implementation of their ideas to be completed and presented to them in order to know if it feels right and give feedback to the programmer. Therefore, the action-reaction process behind any creative work is impaired by this inter-dependency between artist and programmer.

Through early interviews with the target audience, we found that the artists also like to have real-time control over the projection content and over how the use of the technology is explored during the performance, which sets the basis for any improvisational work. Although the artists can show their work in the presence of a programmer or other computer operator, they can not immediately and autonomously perform those changes without depending on the programmer's knowledge [35].

In order to fight the dependency problem that artists often have to face, and to allow them to autonomously explore the creative possibilities in a more comfortable and effortless way, there should be a tool oriented for these people. It would be through this tool that they would be able to manipulate and conduct the content of the projection they are creating and specify the rules of interaction between the performer/viewer and the content created, in an intuitive way.

Having a tool that aids the creation of interactive projections the artists can create interactive installations or performances without (necessarily) depending on the technical skills of a computer programmer. This will, hopefully, allow the creative people to quickly see results during the implementation of their ideas

since, through the interaction with a simple and intuitive Graphical User Interface (GUI), they can quickly and intuitively manipulate the audiovisual content of the piece and explore the combination of rules by which the content plays in relation to the viewer or the performer without needing to write code.

Developing a tool for the creation of interactive projections that serve a creative purpose, will be beneficial for artists like choreographers, performers, theater directors, etc., since they can have a tool that will possibly serve their needs for producing interactive digital art in a comfortable and autonomous way. But many different people can also benefit from such tool even if they are not artists. People like researchers and professionals working with human-machine interaction can benefit from the creation of such system, as well as architects. The latter can, for instance, use a tool like this to create a media wall for a building or use it to create an interactive art piece for a museum or other space found pertinent.

The tool herein proposed, *MotionDesigner*, is a computer program with which the creative people can autonomously create an interactive projection and manipulate its audiovisual content and relate it with data gathered from the body movement of a third-person (performer, person from the audience, etc.). The interaction between the performer and the projection content is done through the use of an RGB-D camera. This tool will hopefully boost the creative process of such interactive art works, since now the art creator can autonomously and immediately test out the different possibilities for the projection content.

## 1.3 Research Questions

Facing the fact that artists often need to depend on a computer programmer to implement the interactive systems to use in their art works, the tool herein presented was developed with the goal of answering the following research questions:

1. Can a computer program oriented for the creative people give them the sufficient autonomy to create their own interactive projections?

2. Is it possible for the artists to feel they have control over the projection content and the interaction rules between the digital work and the performance?

These questions were a conductive force in the software development, in order to produce a more intuitive system and to better understand what are the elements the software must provide the users so they can create a set of graphical scenes to be projected, create an interactive soundscape and set the rules of interaction between the body of another person and the projection according to their own needs.

## 1.4 Objectives

In order to know if a single computer program can help artists to autonomously create their interactive projections, the main objective of this work is to elaborate and develop a standalone software to be used by choreographers, theater directors, performers, architects or anyone who wants to create an interactive performance or art piece by manipulating audiovisual content in real-time through motion capture, and facilitate the creative process of such works.

Using some affordable hardware (a depth-camera like Microsoft Kinect) the software should provide all the means to create, edit and sequence audiovisual content for a projection, allowing the user to create a sequence of different 2D/3D graphical scenes (with computer graphics created in real-time) and set, if wanted, an interactive soundscape using the desired sound samples. The software should also allow the user to specify how the person being detected by the depth sensor will affect the projected graphical scenes and/or the sounds being played, by choosing witch body joints participate in the interaction, as well as adjust the parameters that dictate how the graphical information is represented.

By giving control over the projection content, the tool herein proposed pretends that its users autonomously integrate the elements they need, without having to depend on programming skills or other technical dependencies, by making the

generation, transformation and sequencing of audiovisual content manageable by the user through an intuitive and simple interface. This is how the proposed tool aims to help the creative people, like artists and architects, to develop their work independently and productively.

## 1.5 Research Method

During the development process the Design Science Research method was used [37], which is a process with different models of approach for solving problems and is divided in the following steps:

1. Problem identification and motivation
2. Objectives of a solution
3. Design and development
4. Demonstration
5. Evaluation
6. Communication

The first step of this process resulted in what is explained in Sections 1.1 and 1.2 of this chapter, since they explain the artists' need for having a tool like the one herein proposed. The second step is covered in Section 1.4, which describes which kind of solution we are going to implement. The third and fourth steps will be described in Chapters 3 and 4, which explain each feature of the proposed solution and describe its development process, respectively. Lately, in Chapter 5 the evaluation process and its results will be presented.

## 1.6 Document Structure

This document describes all the aspects of our project, from the proposed system's premises to its implementation and evaluation processes, having the following structure:

- Chapter 2 overviews some of the existing projects in interactive performance using RGB-D cameras and the tools that aid the artists in the creation of such works, comparing them with our approach.
- Chapter 3 gives an overview of the proposed system, describing its principles, features and design and introduces the hardware and software tools we chose to implement this system.
- Chapter 4 describes the implementation of the proposed system, explaining how each feature was implemented and how they are presented to the final user.
- Chapter 5 describes the evaluation method chosen to validate our work and the results of these evaluation tests with the final users.
- Chapter 6 presents the final conclusions of our work and proposes a number of features that can be implemented in the future

# Chapter 2

# Literature Survey

Creating interactive systems using motion capture data is a practice that resulted in many different applications until today, specially in artistic mediums. From all these different applications, a great majority are in the context of interactive projections for a dance performance or installation. Typically what these systems have is a sequence of computer graphics projected on a surface, like a wall or the floor [11] [24] [26], and/or an interactive soundscape to set the mood of the performative work [18]. This area of interactive art performance and installation art is one of the most creative areas explored until today regarding the marriage between arts and technology.

Researchers like O'Neal, Meador and Kurt worked with dancers using motion capture (*mocap*) suits in order to generate computer data by their movement and input this data into an interactive system [30] [31]. Other researchers like Latulipe and Huskey [29] used instead a portable mice, while Hewison, Bailey and Turner [16] used vision sensors that they integrated with the performers' suits. Some years after these projects have appeared, the *mocap* process started to be made using external markers attached to the performers' bodies [21]. The main limitations associated with all these different motion capture techniques are the occlusion of markers or sensors and the movement limitations that arise by the use of special suits and the gear attached to it.

## 2.1 Depth-Cameras Based Systems

The appearance of RGB-D cameras, or depth cameras, opened a new space of opportunity for multimedia computing, allowing the implementation of motion-capture-based systems without the need of special apparel or hardware apart from the cameras/sensors themselves [38]. Many researchers started to work with these depth sensors in order to fight the problems of marker occlusions and movement limitations [26] [11] [24], making the *mocap* process more comfortable and productive.

## 2.2 Interactive Projections

The possibility of having a simple camera capturing all the movement data from a person without the need of special apparel caused the arise of interactive art pieces where a person, or more, can stand in front of the sensor and instantly and comfortably interact with the audiovisuals being computed. The visual result of this interaction is often projected on a plane surface like a wall or the floor. In Figure 2.1 it is shown the setup we typically find in an interactive work like this (e.g., [24] [35]).

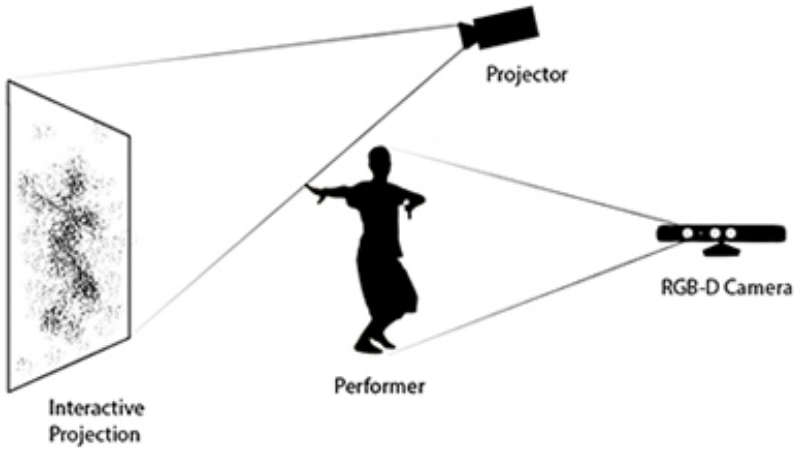



FIGURE 2.1: Typical setup for an interactive projection using RGB-D cameras.

### 2.2.1 Graphical Interactivity

An example of a work that uses depth-sensors to capture the movement data of a person, in this case a dancer, is an interactive performance called *.cyclic.*, which combines computer graphics with dancing and Kinect technology [26]. In *.cyclic.* a pre-sequenced set of images to be iterated are synced with the music and drawn to the screen according to the performer's position.

There are other projects that had a similar approach to *.cyclic.*, providing computer generated graphics manipulated by the performer in real-time, using a depth camera. An example is the project *Divided By Zero* by Hellicar and Lewis [24], which resulted in an interactive dance performance that used depth sensor technology to track the dancer's body silhouette which would affect the visuals that were generated in response, in real-time. In this project, no pre-rendered video was used and the soundscape used was not generated or affected in real-time, but rather was something pre-recorded. See Figure 2.2 to have an idea of the visual aesthetic of this performance.

FIGURE 2.2: An abstract representation of the dancer's silhouette is projected on a wall and transformed by her movement in real-time, so as in Hellicar&Lewis's *Divided By Zero*, 2010.

Another project that focuses on the interaction between projected graphics and a performer using depth sensors technology is one of Benjamin Glover[22]. In

this project, Glover developed an interactive system using openFrameworks (OF) libraries and a Kinect sensor to interact in real-time with computer graphics projected on a wall. The system developed by Glover allowed the configuration of the visual aspect of the projection to be changed in real-time through the interaction with a simple GUI [22], but the order of the elements that would be projected was pre-programed and not easily changeable. The system also did not cover any aspect of interactive sound in real-time. Despite the graphical interactivity of these systems, none of them focuses on sound manipulation in real-time. However, some researchers already explored the possibility of having the performer's movement interact with both the graphical elements and the soundscape of the art piece.

### 2.2.2 Sound Interactivity

A research project that marries interactive sound with graphical elements is the one by Joey Bargsten, in which he developed two interactive applications using PureData and Quartz Composer software to control audio and graphical content, respectively, using a Kinect sensor [17]. To extract the data to input in both PureData and Quartz Composer he used a software called Synapse along with some Open Sound Control (OSC) routers [17]. He concludes that OSC controllers with Kinect technology can bring new opportunities in translating the motion of the human body to transformations in both sound and images. However, Bargsten defended the division of labors and focused on the advantages of exploring sound and video in their separate domains. Despite this, Bargsten's article addresses the possibility of having a system that displays interactive sound and graphics simultaneously and in real-time. Ultimately, no software capable of doing both of these things was actually implemented.

There is another project, by Berg et al. [18], in which sound interactivity is the main focus. In this project the researchers produced a music generator system where each joint of the mover was responsible to transform a specific sound sample. In this system, no graphical interaction was developed, but the generation and

manipulation of different sound samples through the movement of a person's body was a considerable advance in this research area.

By analyzing all these different projects we conclude that, to the best of our knowledge, there is a scarcity of systems that provide the possibility to interact with both the computer-generated graphics and the sound used in the performance or installation. This scenario of having a person not only conducting the behavior of the projection elements but also being the conductor of the soundscape is something yet to be explored. A commom issue among all these projects is that the use of more than one RGB-D camera for the *mocap* process is never referred. The reason for not having these interactive systems using more than one depth camera may be the fact that this represents a big technological challenge, since problems like the unreadability of the point clouds generated by the Kinects may arise (due to the infrared mapping overlap), which makes the data partially unreadable and much more difficult to deal in terms of programing and calibration of the sensors. The system herein proposed was implemented using only one RGB-D camera due to these reasons.

Taking into account the state of the art regarding the use of technology in art performances and installations, from the most suitable hardware to the solutions implemented, one can conclude that the implementation of such systems could be richer if they provided both audio and visual control and if the creative person could have the control of these elements, since lot of interesting possibilities could be fulfilled by their artistic vision.

As depicted in the literature review, the artists or architects interested in creating an interactive projection to integrate in their art work, often do it by asking a programmer or a software developer to implement the solution they need for that specific work [26] [23]. What happens in these situations is that the artists depend on the programmer or engineer productivity to see the results implemented, i.e., they need to wait for the implementation to be completed in order to test it. Only then the artist can continue the creative process by refining that idea and thinking about new ones. This is a problem the artists often find during

the creative process of these type of works. Additionally, the solution typically implemented are a sequence of code lines that run during the exhibition of the art piece and have a preset order of events, i.e., there is no possibility for the creator of the art piece to change the order of what is being projected in real-time.

## 2.3 Tools for Creating Interactive Audiovisual Art

Facing the autonomy problem that the creative people typically have during the creative process of interactive art works that embed technology, there should be a tool that aids them by allowing them to:

1. Choose their own projection content without needing to program them;
2. Change the aspect of the projection content and the interaction rules between the performer and the projection and/or the soundscape, in real-time;
3. Freely sequence the order of the interactive content he/she is using in the projection, i.e., what is seen/heard during the performance/exhibition and when.

One of the few examples of artists-oriented tools for creating interactive performances/installations is a tool called *MotionDraw* [35], implemented using OF libraries, which focuses on enhancing the experience of the artist when conducting an interactive projection in real-time. In this work only the graphical interactivity is considered, i.e., no sound interactivity is explored, but they explore the graphical interaction in a different way than the works discussed previously by creating a tool oriented for the artist. In *MotionDraw*, the creative person can directly manipulate the visual aspect of the projection - which conceptually is something similar to brushes painting on a canvas - by interacting with a simple GUI that allows him/her to configure the aspect of the brush for each body joint being captured by the Kinect sensor and choose which joints are active (2). This system's interface, depicted in Figure 2.3, allows the artist to freely explore the

possible visual representations of the scene in real-time. However, this tool does not give the user freedom to choose another content for the projection apart from the drawing brushes. So, this is a tool for a one-case scenario and lacks on other types of interactive scenes for the users to work with and an environment where they can sequence the projection content and use them as they want.

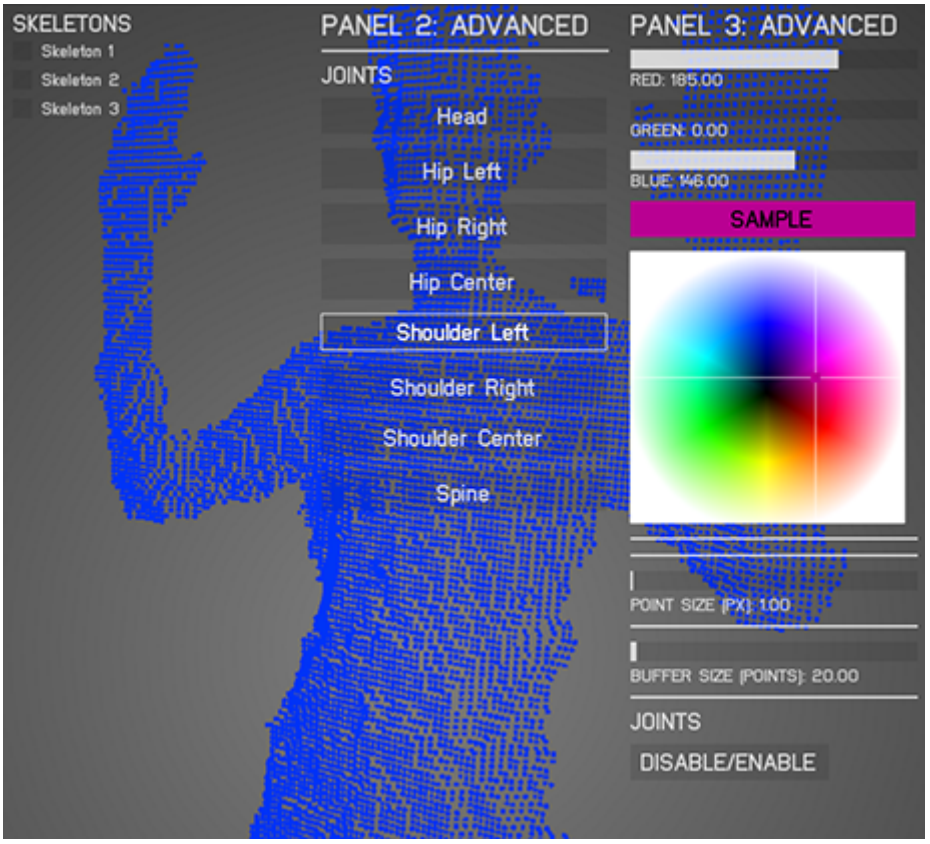



FIGURE 2.3: MotionDraw's simple GUI allows the user to control the aspect of each joint's brush in real-time.

Another example of a tool that was developed for the artists to directly manipulate the projection content by interacting with a simple and intuitive GUI is Adrien Monodot and Claire Bardainne's *eMotion* [15], which is a software developed for displaying virtual objects chosen by the user (1) and make the body motion of a performer, which is being captured by a depth sensor, to interact with those virtual objects. The user of the software can easily change the behavior and representation of the virtual objects by manipulating the physics laws governing those objects, through the interaction with a simple GUI (2). However, this tool is also a one-case scenario, meaning that the users do not have much freedom to

choose which elements are represented on the scene nor sequence them as they want.

In the case of *MotionDraw* the user can set the aspect of the projection in real-time by manipulating the color of the scene elements and the width of the trail left by the joints, as well as choose which joints are visible, which verifies the second feature of the three previously proposed. In *eMotion* the users have more freedom to choose which elements are shown by choosing which type of 3D graphics are shown in the projection, however they can not choose and sequence a set of different types of graphics to be projected, so it is not a considerable freedom for the user in terms of control. This, however, can validate both the first and the second conditions in the case of *eMotion*.

In both the *MotionDraw* [35] and *eMotion* [15] cases the user of the software, typically the artist responsible for envisioning the interactive projection, does not have more than one type of graphical scene to use in the projection and, consequently, can not sequence multiple types of scenes to project (3). In both these cases the artist also can not use interactive sound within the provided tools, which means that if they want an interactive soundscape to go along with the projection they need to implement it (or ask for that implementation) outside the environment of the provided tool. This makes none of the tools we know at the moment eligible for the third feature, i.e., allowing the users to freely sequence the order of the interactive content they want to project.

At this point, one can see that the number of tools available for the creative people to make and conduct their own interactive performances or installations is small and that, even those that already exist, do not give the user freedom to choose different types of interactive content to be projected and sequence them along the exhibition time. If the users wants to mix different types of interactive graphics in their interactive projection they can not do it autonomously, since they need a programmer to implement other scenarios. Having he ability to use more than one type of interactive scenes in the projection and, for each of them, having the freedom to parameterize it and configure its interaction rules is something we

will cover in the system herein proposed, as well as the ability for the user to create an interactive soundscape to go with the projection.

In order to fill the gap on the existing tools that intend to aid the artists, and to fulfill all the three features proposed, the tool herein presented, *MotionDesigner*, provides the users an environment where they can set which type of graphical and audio elements are going to be projected during the interactive performance or installation and also when, i.e., the users can sequence the audiovisual elements along the projection time, as well as change their aspect and behavior in real-time. This makes the proposed tool fulfill all the three aspects that shall aid the users in creating an interactive projection.

# Chapter 3

# System Overview

In order to cover the gap that exists in the supporting tools for the artists to create their own interactive projections based on the capture of human body movements, we propose a software tool that focuses mainly on the creative person responsible for the artistic ideas behind the interactive work. In the system herein proposed, the creative person is the final user of the software and will act as the conductor of the projection itself, from dictating which content is being presented in the projection to setting the rules of interaction between the performer's body and the audiovisual elements of the performance.

## 3.1 User Interface

The system presented in this dissertation is composed of a software we developed, which we called *MotionDesigner*, and an RGB-D camera, like Microsoft Kinect, as seen on Figure 3.1. The RGB-D camera captures all the movement from the performer's body and feeds it into the software, which computed the graphical and audio data according to the body pose. A projector connected to the computer projects the resulting visuals in a plain surface. The developed software and is to be used by anyone who wants to create an interactive projection using real-time computer-graphics and sound manipulation through the movement of a human

body. It is by directly interacting with the software that the users can to change the aspect and behavior of the projection.

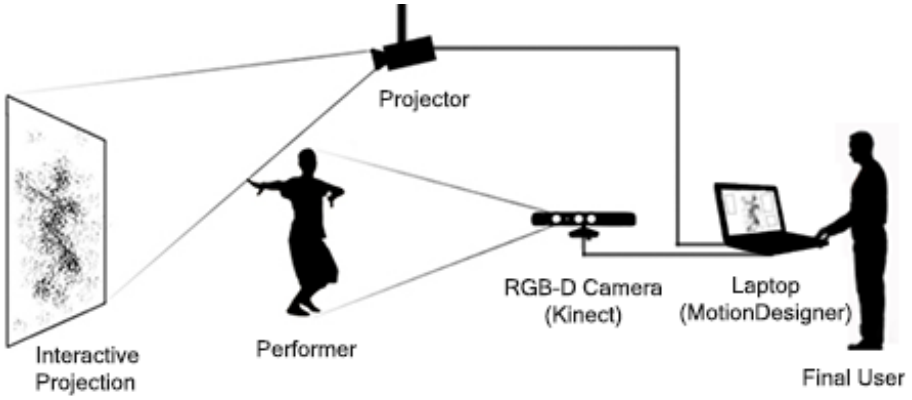



FIGURE 3.1: The system herein proposed is composed by a software, *MotionDesigner*, an RGB-D camera, which captures the body movement of at least one person, and a projector that maps the resulting visuals in a plain surface.

The software to be used by the creative person who pretends to create an interactive projection, *MotionDesigner*, has a simple and intuitive Graphical User Interface (GUI). It is by interacting with this GUI that the creator of the projection can choose what to display on the projection (which graphical elements), what is their visual aspect (which can be set before projecting the graphical content or during the live projection) and when these graphical elements, as well as the sounds the user wants to play, will appear during the projection. Figure 3.2 shows the main interface of *MotionDesigner*, which is called Editing Studio.

The Editing Studio is the first environment to be presented to the users, and it has a timeline-based interface. In this environment the users can drag to a timeline a set of graphical scenes to be projected and sounds samples to play along with the projection. The panel with the graphical scenes and the panel with the audio files are displayed according to the tab currently selected by the user (in Figure 3.2 the scenes palette is the panel currently selected). It is on this timeline where the users can manipulate the projection time of each audiovisual element and set their order of appearance. These are all concepts borrowed from video editing systems.

Besides sequencing the audiovisual elements and set their duration, the user is also be able to set timestamps/markers during the projection time of a scene

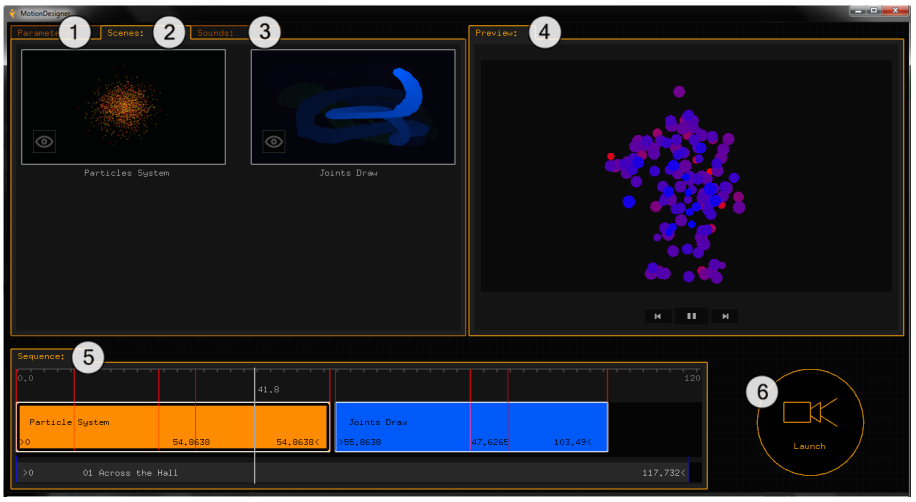



FIGURE 3.2: **1 - Parameterizations Panel**, where the user can add timestamps to a scene form the timeline and set the desired parameters for that time instant; **2 - Scenes Palette**, which displays the graphical scenes the user can drag to the timeline; **3 - Sounds Palette**, which displays the sound samples the user can drag to the timeline; **4 - Preview Player**, where the user can preview the scenes representation over time according to the current parameterizations; **5 - Timeline**, where the user can sequence the graphical scenes and set their duration; **6 - Launch Button**, where the user can play/launch the sequence onto the projector.

(represented by the red lines on the timeline) and associate to them a set of values for that scene's parameters. This allows the user to pre-parameterize the scene without having to change the scene visual aspect only in real-time, i.e., only when the scene is being projected. This feature is provided because, although the real-time live manipulation is one of the core principles of the proposed system, it is also interesting to allow other scenarios where the users already know what they want to project and how.

The users also need an interface to freely parameterize an interactive scene, i.e., a GUI that allows them to experiment different possible representations for a specific scene. Figure 3.3 shows the interactive scenes editing GUI. This interface is presented to the users whenever they press the *Explore* button, which is represented in Figure 3.2 as an eye icon on the bottom left corner of each scene from the scenes palette. When the users press this button the respective scene is launched in fullscreen along with this GUI, so that they experiment with that scene before the live projection. When the timeline sequence is projected during

the live performance the users can see the current scene content replicated on the computer screen along with this editing GUI, which allows live parameterizations in real-time.

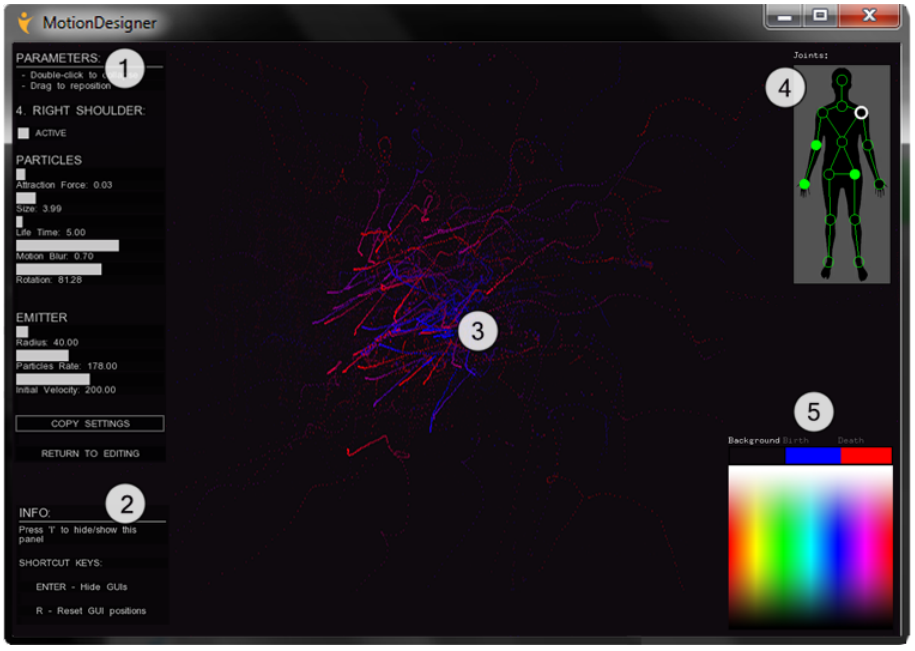



FIGURE 3.3: **1 - Parameters Control Panel**, sliders-based panel where the user can set a different value for each scene parameter; **2 - Info Panel**, panel with useful info, namely short key commands; **3 - Scene Content**, where the scene is displayed as is being rendered on the projection; **4 - Joints Selector Panel**, where the user can choose which of the performer's body joint is going to be parameterized (how much that joint affects the projection); **5 - Color Picker**, where the user sets the color for each scene element

Whether the users are live projecting the audiovisuals sequence or simply experimenting different parameterizations for a specific scene, they are presented with the interface depicted in Figure 3.3. The content of the scene depends on the type of scene that is being previewed or launched.

## 3.2 Hardware Setup

Since the audiovisuals used in the projection focus on the interactivity with body movement, a technology to successfully recognize the human body and provide

that information as usable data to be computed by the software is needed. *MotionDesigner* is prepared to be used with an RGB-D camera, which is responsible for capturing point clouds of the scene. These point clouds data are processed by a third-party software, namely, NiTE, to infer the pose of the performer's body.

In order for the proposed system to be easily usable and accessible to the majority of the target audience, we needed to integrate a reasonably accessible depth camera. Microsoft Kinect was seen as the most appropriate camera since it provides all the necessary technology by a considerably low monetary cost.

### 3.2.1 Kinect sensor

Kinect is a consumer-grade range camera, or depth camera, created by Microsoft in 2010 [34]. It was initially designed with the purpose to commercialize it along with Xbox 360 so that the users of this console could physically interact with the game and control them through their body movements [6]. However, since its release, many people started to explore its application in different domains [27].

Like any other RGB-D camera, the Kinect has an RGB camera and a 3D depth sensor. As depicted in Figure 3.4, it also has a multi-array microphone (composed of four microphones) for sound capture and a 3-axis accelerometer, from which is possible to infer the current orientation of the camera [38]. The RGB camera has a resolution of 1280x960 and it can be used to capture a colored image or to stream video [22]. The depth sensors are composed of an infrared projector and an infrared camera (see Figure 3.5). The first emits a dot pattern of infrared light onto the scene and the other receives the reflected beams of the pattern and infers the depth information from the deformations imposed by the geometry of the scene to the projected dots [38].

After the release of Microsoft Kinect for the Xbox console, Microsoft announced, on February 21, 2011, the release of Kinect for Windows. This is the same device already released for the console but usable and programmable on the

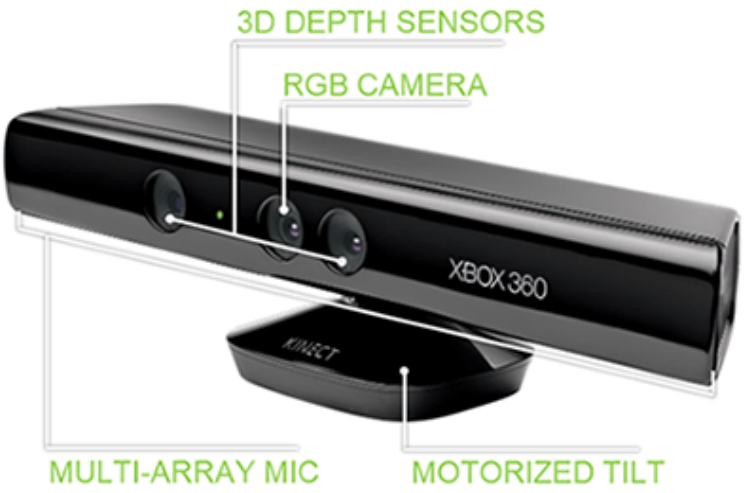



FIGURE 3.4: Microsoft Kinect hardware composition

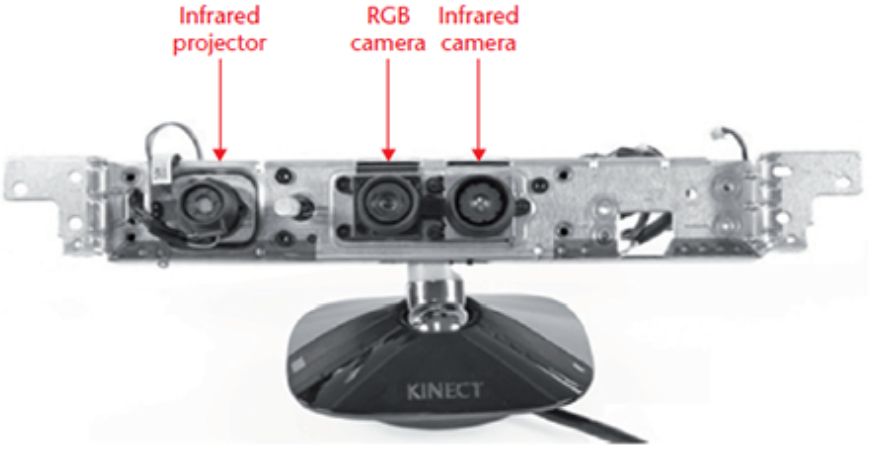



FIGURE 3.5: Inner structure of the Microsoft Kinect sensors

computer. They released the device in the same year, with a non-commercial Software Development Kit (SDK) associated to it [32].

Soon after the release of Kinect for Windows, a lot of people started to program different type of systems, exploring the creative possibilities behind the application of this technology. Since its release a lot of artists already integrated the Kinect in their artistic works, specifically in interactive dance performances and art installations [24] [26] [11].

## 3.3 Software Setup

Since the Kinect sensor is being used to capture the body movement data to input into our system, the choice of the programming environment and the libraries used to manipulate this data was done carefully. Currently there are two strong candidates for the programming libraries to be used to develop the proposed system, being those:

**Kinect SDK** Released from Microsoft along with Kinect for Windows [2]
Programming Languages: C++, C#, Visual Basic or other .NET language

**openFrameworks** An open source toolkit designed to assist creative coding [9]
Programming Language: C++

Table 3.1 presents the main characteristics of both options side-by-side. This comparison allowed us to understand what was the most appropriate choice for the project implementation.

| **openFrameworks vs Kinect SDK** | |
|---|---|
| **openFrameworks** | **Kinect SDK** |
| Open-source and cross-platform | Not open-source nor cross-platform |
| Commercial usage | Commercial applications not allowed |
| Skeletal and Hand Tracking | Skeletal and Hand Tracking |
| Oriented for sound sampling control | Sound sampling control is possible |
| Oriented for computer graphics generation | Computer graphics generation is possible |

Table 3.1: Comparison between openFrameworks and Kinect SDK features

Choosing between Kinect SDK and openFrameworks simply regarding the motion tracking capabilities is something that should suit the developer preferences, since both can do it. However, there are several reasons that highlighted openFrameworks over Kinect SDK. As Table 3.1 shows, openFrameworks, unlike Kinect SDK, is cross-platform, allowing to port the software to Windows, OSX and Linux

systems in a more comfortable way, which was an important aspect since our system was projected with no operating system constraints.

Another issue that highlights openFrameworks over Kinect SDK is how it facilitates the process of drawing graphical content on the screen. Since openFrameworks is a wrapper of libraries such OpenGL, GLEW, GLUT and Cairo [9], it already provides basic methods for drawing geometric shapes, in the case of 2D, and easily generate 3D models and shaders, which makes the programming of the graphical elements much more comfortable.

Regarding the sound elements control, openFrameworks also provides simple methods for playing sound samples in formats like MP3, WAV and AIFF, and transform their speed, volume and panning options in a more comfortable and productive way thanks to some libraries it wraps, like rtAudio, PortAudio, OpenAL, etc. All the playback and grabbing of video files are done by using libraries like Quicktime, GStreamer and videoInput [9].

Since openFrameworks was created with a focus on multimedia computing, from 2D/3D graphics to images, sound and video, it is a very good tool for developing multimedia projects that will portray real-time environments. Also, openFrameworks uses C++ language, which is a low-level programming language, allowing us to control and process data very fast and optimize the use of memory in projects that demand a high computing speed and deal with performance issues, which is the case. Considering all these aspects, openFrameworks is the most suitable choice for the project herein presented.

### 3.3.1 openFrameworks

OpenFrameworks (OF) is an open-source C++ toolkit developed and maintained by a large community of programmers and creative coding enthusiasts [9]. OF allows programmers to easily generate and render 2D/3D graphical content, as well as manipulate different sound samples. The libraries it wraps around allows the developer to manipulate all these different types of media in a more comfortable

way. Since the tool herein proposed, *MotionDesigner*, also deals with point data gathered from the Kinect sensor some additional libraries and middleware were needed in order to successfully implement the system.

### 3.3.2 OpenNI & NiTE middleware

OpenNI, which stands for *Open Natural Interaction*, is an open-source library, used by openFrameworks, which allows the programmer to access and control PrimeSense compatible depth sensors [10]. Since PrimeSense was the company behind the depth sensing technology of the Kinect sensor, this library was a suitable choice. The version used in the project implementation is OpenNI 2.0, and its API allows us to initialize the Kinect sensor and receive RGB, depth and IR video streams from it.

Along with OpenNI2, a middleware, also released from PrimeSense, was used, which is called NiTE (*Natural Interaction Middleware*). NiTE is a very useful and efficient middleware when dealing with human body recognition and tracking, since it can detect when a body is in front of the sensor and detect its joints very quickly, with minimal CPU usage [8]. This middleware already provides algorithms for hands and full-body tracking, which use data from the RGB, depth and IR streams provided by OpenNI2 functions.

### 3.3.3 openFrameworks Add-ons

Additional libraries were used along with openFrameworks default tools, namely *ofxUI* and *ofxSecondWindow*. The first add-on, *ofxUI*, was developed by Reza Ali and it allows us to create GL based GUIs like buttons/toggles, drop-down lists, sliders, labels and text fields without having to program them from scratch. Its functions are implemented using OpenGL and use OF drawing calls to render the different widgets onto the screen [14].

The second, *ofxSecondWindow*, is an add-on created by Gene Kogan, which allows the programmer to easily create and call multiple windows to use within the application [28]. This was a great help since the management of multiple windows is typically a time consuming task to program in C++.

# Chapter 4

# Development and Implementation

For the development of *MotionDesigner*, openFrameworks 0.8.4, OpenNI 2.2 and NiTE 2.2 middleware were used, along with *ofxUI* and *ofxSecondWindow* add-ons. The chosen IDE was Microsoft Visual Studio 2012 Express and for version-control it was used Git. The software architecture for *MotionDesigner* is depicted in Figure 4.1, which also shows how each software and hardware components were integrated. The user interacts with the software through a GUI, which sets the graphical aspect of the current graphical scene and the rules for the motion capture process. OpenGL renders the scene according to the user's parameterization, OpenNI turns the depth sensor on when the user chooses to project the sequence and NiTE tracks the skeleton joints the user wants to be tracked.

There are two main environments that make *MotionDesigner* the tool we envisioned: (1) Editing Studio, where the user can choose which graphical scenes and audio samples will be used in the interactive projection, sequence them and pre-parameterize them; and (2) Interactive Scenes/Audio Editing environment, which allows the user to explore the different possible parameterizations for each scene and for affecting the audio samples being played.

Whenever the users open *MotionDesigner* to create an interactive projection, the environment presented to them is (1), form which they can see the audiovisual material available to use in the projection and sequence them as they like. For

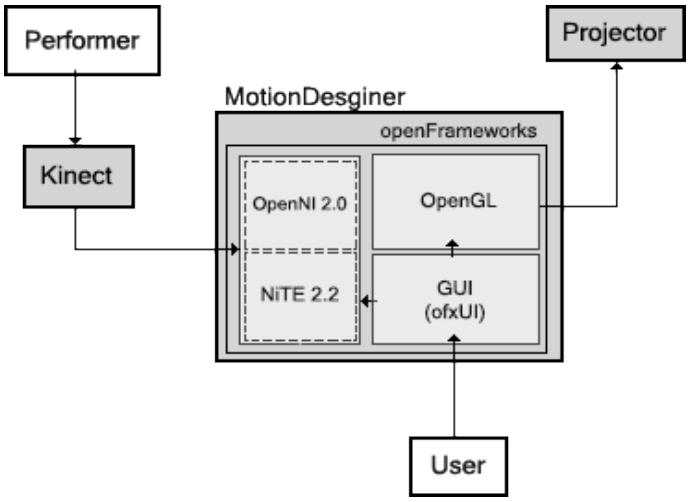



FIGURE 4.1: System architecture. The performer interacts with the Kinect, which feeds the software with body movement data. The graphical content is rendered according to this data and and the current parameterization set by the user through a GUI. A projector renders the scene in a plain surface.

controlling the representation and behavior of the projection element, the environment (2) is presented, where the user can try different possible representations for each graphical scene and configure the motion capture rules, e.g., deactivate some joints so they do not affect the projection content. A similar environment is used to control the interaction with the sound samples played during the projection.

## 4.1 Editing Studio

The main environment of *MotionDesigner* is the one called Editing Studio. It is through this environment that the users can access all the interactive scenes and audio samples to use in the projection and where they can also call the editing environment for the interactive scenes or audio elements.

Since the users would be able to sequence the scenes and the sound samples to be played during the projection as they like, the Editing Studio has a timeline-based interface. The timeline allows the users to manage the order of the audiovisual elements during the projection and the duration time for each of them. This timeline is presented to the users with a simple and minimalist design,

as shown in Figure 4.2, and was created by using geometric primitives, e.g., lines and rectangles, using *OpenGL*'s 2D primitives drawing operations.

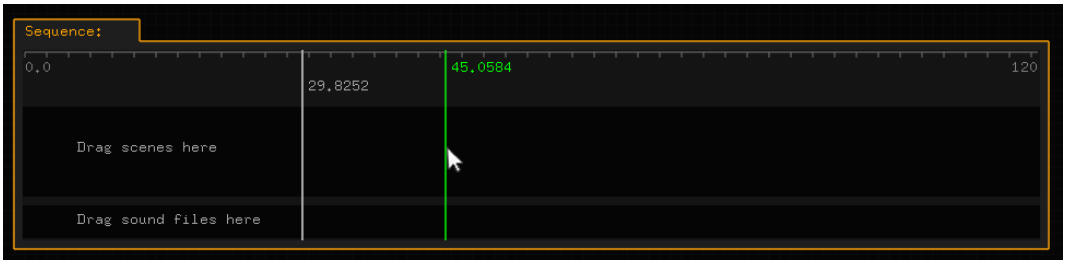



FIGURE 4.2: The Editing Studio's timeline is represented with a simple interface. The user should be able to drag the time marker (in white) and see the current time while hovering the timeline with the mouse (green)

The dimensions of the timeline take in consideration the current size of the window frame. Every time the user resizes the window, the timeline reshapes itself so that it always fits on the window frame with proper dimensions (the timeline width value depends on the window width).

For showing the current time on the time marker (white line) we used a linear relationship between the position of the markers on the screen, given where the timeline begins and ends on the screen, and the time it is supposed to show given the initial time (0 seconds) and the maximum time on the timeline (for instance 120 seconds, as on Figure 4.2). By doing so, we can get the time value to be shown in the marker in relation to the boundaries of the timeline and the maximum time of the timeline. The same principle was used for calculating the time value to display in the cursor marker (green line), but now knowing the position of the marker and wanting to know the time corresponding to that position on the timeline.

The users can control the time marker by dragging it with the mouse or by pressing the arrow keys on the keyboard. Besides this, a few more key commands were implemented and binded to the following keyboard keys:

- **Left/Right Arrows** - Move time marker backwards or forward in time, respectively
- **SPACE** - Play/pause sequence on the timeline

- **ENTER** - Replay sequence
- **+/-** - Increase/decrease the timeline maximum time (sequence duration)
- **F** - Set timeline maximum time (sequence ending) to the end of the last scene
- **DEL** - Delete selected scene from the timeline
- **H** - Show/hide virtual skeleton on the preview player

These commands allowed the user to have a more intuitive control over the timeline, since these are standard key commands in most video editing softwares.

In order to store the graphical scenes that the user could drag to the timeline, a scenes palette panel was created. This panel has the same visual aesthetic as the timeline, as Figure 4.3 shows, with an image resembling each type of interactive scene, and a bitmap string to displaying its name. This reinforces the simple and minimalist aesthetic we are looking for in the development of the software.


FIGURE 4.3: A separate panel is provided with a palette of the available interactive graphical scenes. The user can directly drag each scene from the palette onto the timeline, in order to use them as part of the projection sequence

Whenever the users drag a scene from the scenes palette onto the timeline, they create a new timeline object, i.e., a new timeline component which can be

dragged and resized within the timeline and stores all the data related to that scene (when it begins and ends on the sequence, the parameterizations it has on a given time instant, etc.). This timeline object stores the type of scene it refers to (the *Particle System* or the *Joints Draw* scene) and all the parameterizations data related to that scene. This newly created timeline object will automatically start on the instant where the last scene from the timeline ends. This kind of knowledge is made possible since all the timeline objects, i.e., all the scenes instances on the timeline, are stored in an array.

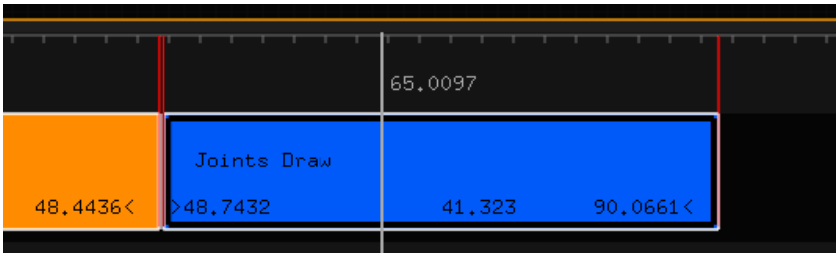



FIGURE 4.4: A scene on the timeline is represented as a rectangular object with text information regarding its time properties (beginning, ending and duration variables) and the correspondent type of scene

### 4.1.1 Scenes Parameterizations

One of the main features of *MotionDesigner*'s Editing Studio is the possibility for the users to pre-parameterize each graphical scene before projecting it. This means the users can preset the aspect and behavior of the graphical scene's elements over the projection time, without having to do it only during the live projection. The users can still parameterize the scenes elements in real-time during the live projection, but when the timeline sequence is projected it will be rendered and computed according to these pre-parameterizations data, so it can always have a preset aesthetic.

Each timeline object (graphical scene or sound sample) has a bi-dimensional vector to store the list of parameterizations for that element. What this vector stores is the list of times instants, i.e., timestamps, that have a parameterization associated to it. A parameterization vector stores real-valued numbers, which

refer to the values of the parameters we allow the user to control associated to a specific instant in time. This vector will take part of the list of vectors (list of parameterizations), as illustrated in Figure 4.5.

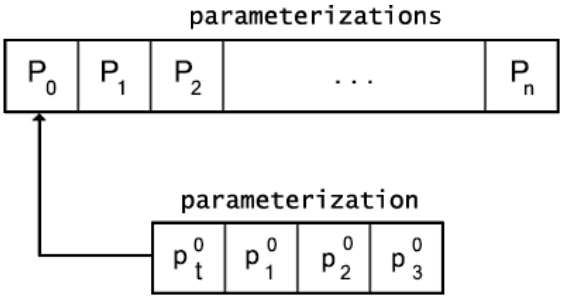



FIGURE 4.5: The parameterizations for a scene on the timeline are stored in a vector (`parameterizations`). Each parameterization ($P$) corresponds to a vector which stores a timestamp ($p_t$), i.e., a time instant between the beginning and the end of the scene, and the parameters values for that scene in that instant $p_t$ (e.g., in *Joints Draw* scene $p_1$, $p_2$ and $p_3$ refer to `size`, `trail` and `speed`, respectively)

Another panel was created just to manage all the operations related with the scenes parameterizations. Through this new panel, shown in Figure 4.6 the user can access the list of existing timestamps, i.e., parameterizations, for a specific scene selected from the timeline, add timestamps to the list or delete an existing timestamp. The parameterizations panel was added to the scenes palette panel location and can be accessed through a tab selecting system.

In order to create a new timestamp with a parameterization associated to it, the user should first click on the respective scene on the timeline and then click on the corresponding button on the parameters panel (see Figure 4.6). After clicking this button, a new timestamp is automatically added to the list (a new parameterization vector is created and added to the `parameterizations` vector of the selected timeline object). The timestamp value for the newly created parameterization is the current value on the time marker (white line on timeline) and the values of the parameters are default values (which can later be changed). If the value on the time marker is not within the beginning and ending of the selected scene from the timeline, the timestamp can not be created. If the users try to

create the same timestamp twice, i.e., two parameterizations for the same time instant, they can not do it, since duplicates are not allowed in the timestamps list.

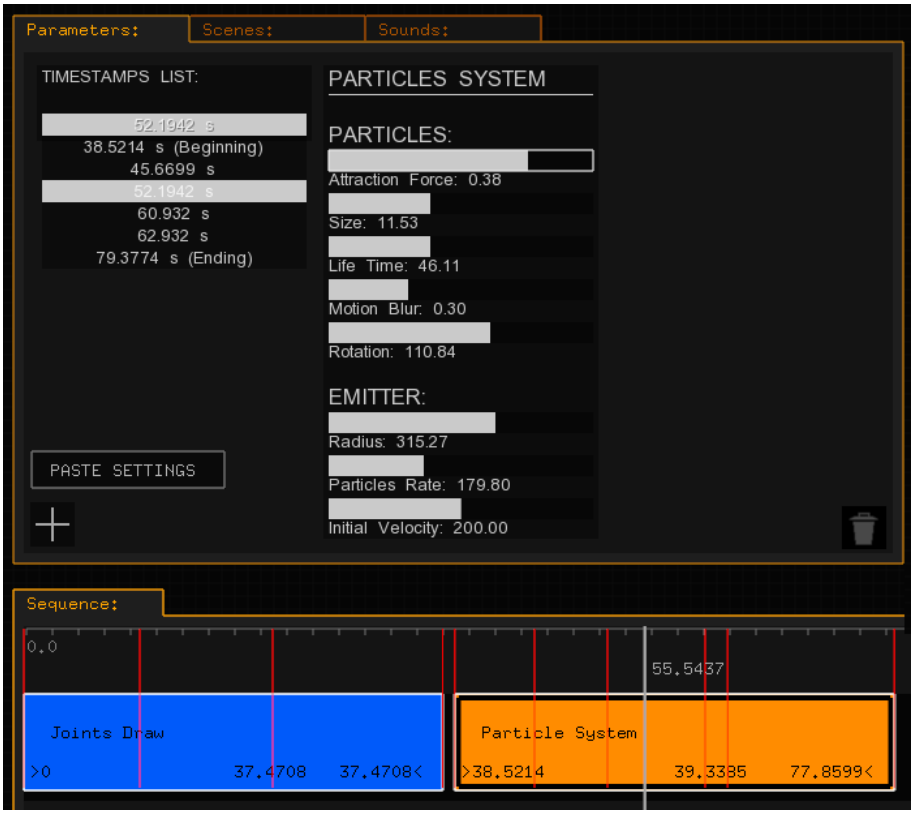



FIGURE 4.6: The Parameterizations Panel shows a drop-down list of the timestamps already created for the selected scene on the timeline. On the right a sliders interface is shown, where the user can change the parameters values for that time instant. A button for adding a new timestamp is provided on the bottom left corner of this panel. The timestamps appear as red lines on the timeline.

To allow a more efficient management of the parameterizations, specially if one wants to know which timestamp will be read next when playing back the timeline sequence, when a new timestamp is added to the list it will be added orderly, i.e., when a new parameterization vector is added to the parameterizations vector, the latter is automatically sorted according to the timestamps values, in an ascendant order. To sort the parameterizations array the Selection Sort algorithm was implemented [25]. This algorithm was chosen due to its simplicity and to the performance advantages associated to it, since this is one of the vector sorting algorithms that best preserves the auxiliary memory usage.

After selecting one of the timestamps from the list, a sliders interface is shown (similar to those found on the interactive scenes GUI) where the parameters values for that time instant can be changed by the users as they desire (see Figure 4.6). If the users want to delete a timestamp parameterization they can do it by clicking on the respective button displayed on the bottom right corner of the parameterizations panel, after that timestamp is selected from the list.

A *Paste Settings* button was also added, which allows the users to set the parameters values of the currently selected timestamp to the values copied from the interactive scenes editing GUI (these copy/paste operations for the parameters values are further discussed on Section 4.1.3 of the current chapter).

### 4.1.2 Previewing the Projection Sequence

In order for the users to be able to preview the aspect and behavior of the scenes elements, according to the timeline sequence, before projecting it, a preview player was added to the Editing Studio environment. This is a necessary feature in order to give the users comfort and efficiency during the management of the graphical scenes to be used in the interactive projection.

A new panel was created with a small preview screen and three control buttons to control the time marker along the timeline (besides the keyboard and mouse controls already implemented). A *Play/Pause* button and two *Skip* buttons were added. The *Skip* buttons skip the time marker to the next (or to the previous) timestamp of the scene or, if it is on the last timestamp, to the beginning of the next scene on the timeline (and vice-versa).

The preview player panel was ultimately placed where the Scenes Palette panel was on the original sketches (see Section A.1 of Appendix A) and it renders each scene from the timeline according to the parameters values on the current time instant (the parameters values interpolate between each timestamp). A virtual skeleton, moving in a loop routine, is also displayed so that the user can see the behavior of the content previewed in relation to the movement of a human body.

Figure 4.7 shows the preview player displaying a particle system scene with the virtual skeleton reference, which can be set as visible by the user by pressing the H key on the keyboard.

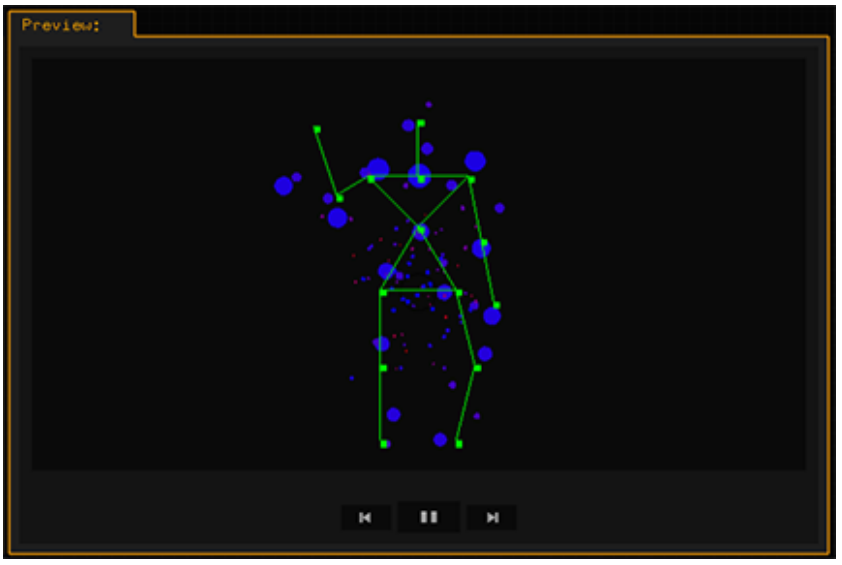



FIGURE 4.7: The Preview Panel provides a player which previews the timeline scenes content over time. It also displays a virtual skeleton that moves with a preset of movements so that the user can see not only the visual aspect of the scene elements but also their behavior. The circles represent the particles tracking the skeleton joints.

### 4.1.3 Displaying the Scenes

As depicted on Figure 4.3, each scene image from the scenes palette has a small button on the bottom left corner. This is the *Explore* button that in our early sketch was reserved to the control buttons area on the bottom right corner of the screen. It was found, though user testing, that clicking on a scene from the timeline or from the scenes palette and then clicking on this button on the other side of the screen was an unnecessary process. With the new solution implemented, the user can launch a specific type of graphical scene on fullscreen, in order to explore the parameterizations possibilities, by just clicking on this button on the respective scene from the scenes palette.

If the user clicks on the *Explore* button, represented on Figure 4.8, it launches the respective interactive scene with the GUI elements reserved for this context,

which are a parameters panel, a joints selector and a color picker (these are further discussed on Section 4.2 of the current chapter).

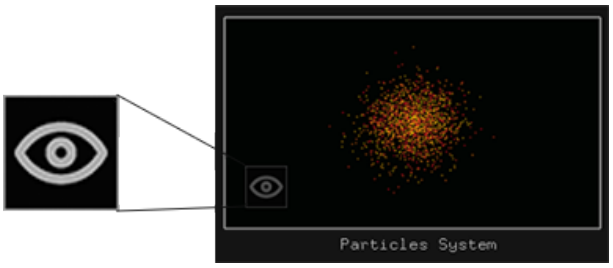



FIGURE 4.8: The *Explore* button allows the user to launch an interactive scene and freely explore the possible parameterizations of the scene elements

When the user launches a scene, the respective scene is presented with a default aspect and behavior but with a GUI that allows the user to change these characteristics in real-time by freely exploring the possible different parameterizations for that scene. On the panel where the users can change the parameters values of the scene's elements, the parameters panel, there is a *Copy Settings* button, which, when clicked, writes the current parameters values of that scene in a XML file. This XML file is read whenever the user clicks on the *Paste Settings* button from the Parameterizations Panel of the Editing Studio. See Figure 4.9 to see the data flow in this copy/paste parameterization scenario.

By alternating between each interactive scene exploration and the editing studio, where the user can sequence these scenes and pre-parameterize them along the projection time, the users can comfortably achieve the interactive audiovisual experience they want within the provided materials.

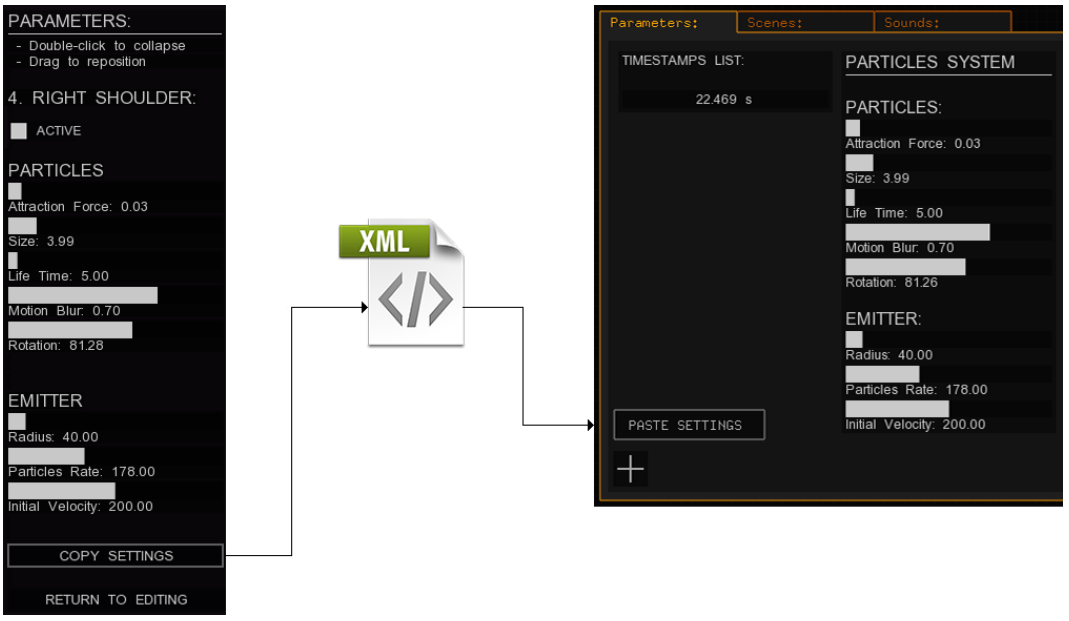



FIGURE 4.9: Parameterizations data flow. When the users are controlling an interactive graphical scene (on the left) they can click on a *Copy Settings* button, which will save all the parameters sliders values into a XML file. Then, on the Editing Studio (on the right), the user can click on a *Paste Settings* button to set the parameters values of the currently selected timestamp to those on the XML file.

### 4.1.4 Projecting the Sequence

As depicted in the Editing Studio's interface early sketch (see Section A.1 of Appendix A) the bottom right area of this environment was reserved for control buttons. A button for launching the projection sequence, i.e., the interactive scenes and audio on the timeline, was added to this area of the GUI, as shown in Figure 4.10.

When the user clicks on the *Launch* button a second window is created, which will render the interactive scenes as they appear on the original window but without the GUI elements. This window is the one which is supposed to be dragged to the projector area, i.e., to the second screen of the computer (assuming it has the projection configuration in Extend mode). This duplicated window is created using the *ofxSecondWindow* add-on, getting the result shown in Figure 4.11.

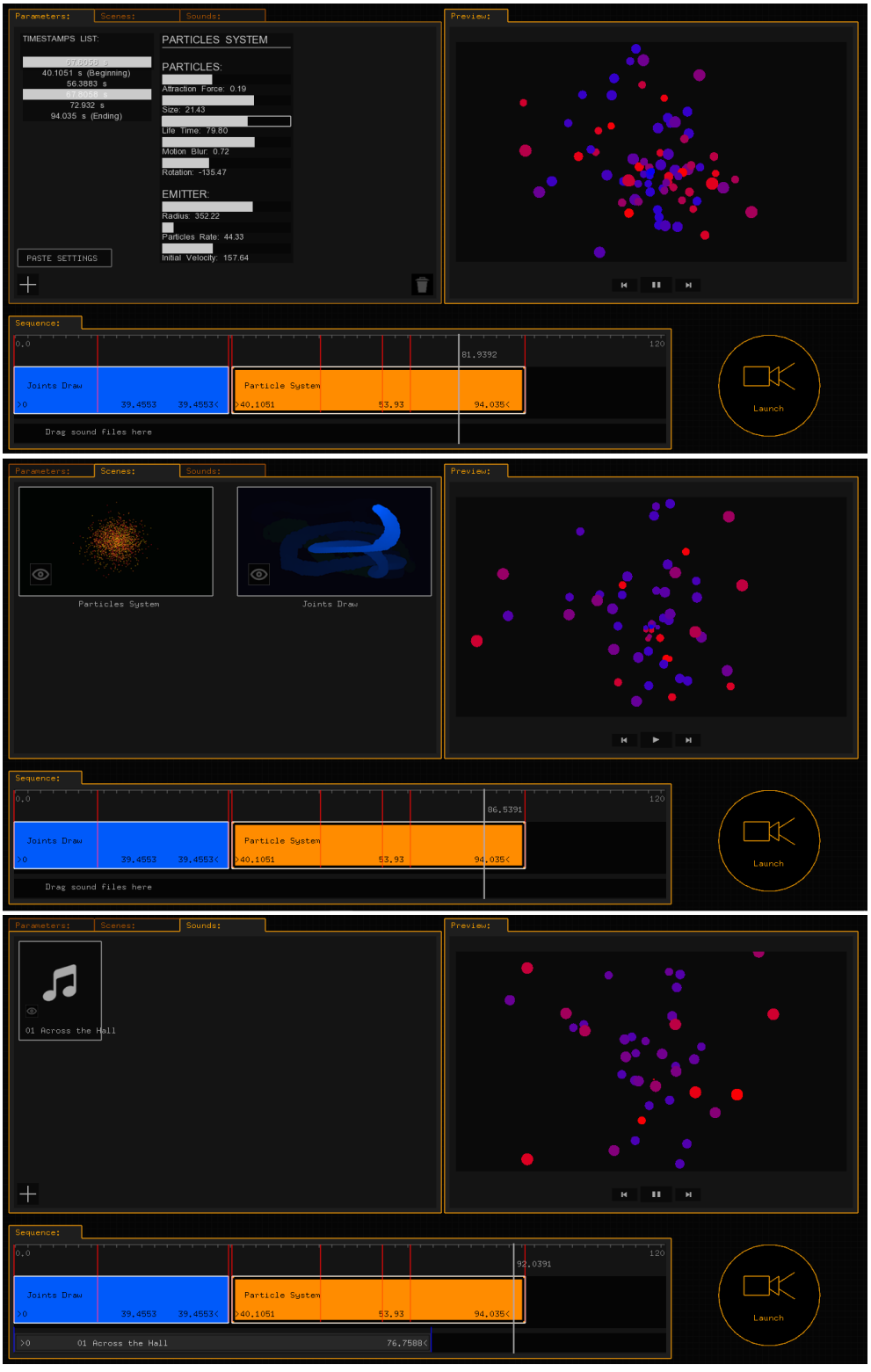



FIGURE 4.10: The Editing Studio interface provides four different panels. A panel with a Preview Player on the right and on the left a panel that can show three different contents: the audio files the users can drag to the timeline, the graphical scenes they can use and the parameterizations control panel for a scene or audio element selected from the timeline

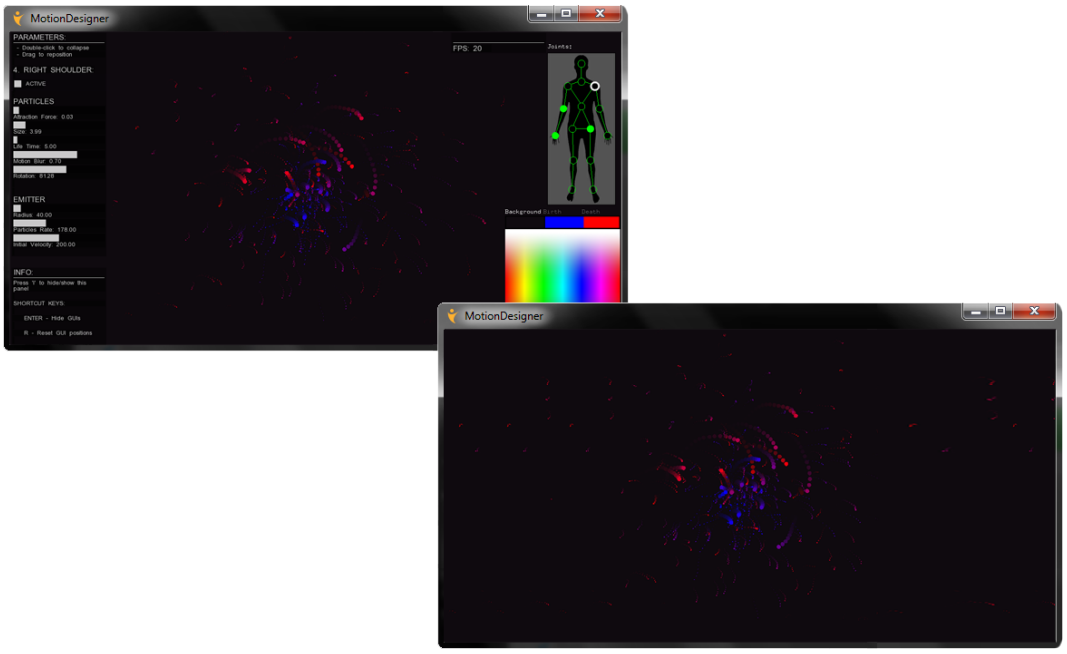



FIGURE 4.11: When the user launches the timeline content into the projection a second window is created without the GUI elements. It is the content of this second window that will be shown by the projector

## 4.2 Interactive Graphical Scenes

Unlike other systems that have a set of images to embed with the projection or some preset graphics to be projected the way they are, like in *cyclic* for instance [26], in the software herein proposed the computer completely generates the graphics algorithmically. The aspect and behavior of the rendered scenes are dependent not only on the movement of the person interacting with the projection but also the configuration set by the conductor of the projection, which can be always changed in real-time.

For each type of graphical scene, the user should be able to control its visual aspect and behavior in real-time, which is possible due to the GUI shown in Figure 4.12.

### 4.2.1 Scenes Editing GUI

In order to allow the users to have real-time control over the graphical scenes content, a scenes editing environment was implemented. Whenever the users are either live projecting interactive graphical content or simply experimenting each scene parameterization on the editing studio, the GUI shown in Figure 4.12 is presented to them.

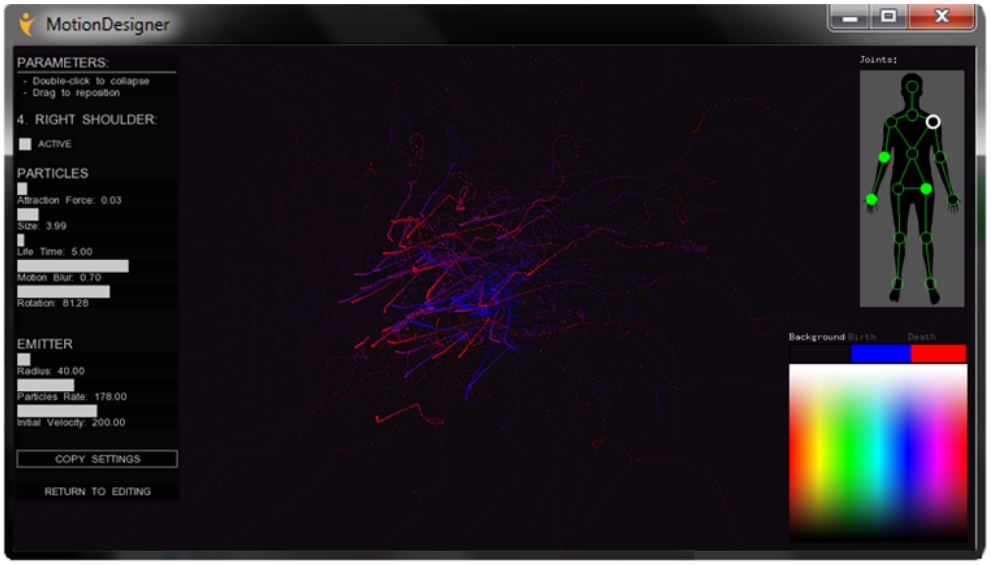



FIGURE 4.12: Interactive Scenes Editing GUI. A sliders-based panel allows real-time parameterizations for the scene's elements (left panel). A joints selector panel allows the user to choose to which joint the current parameterization will be associated with (upper right panel). The color of the scene's elements can be set through a color picker (bottom right corner).

Regarding the parameters panel (left corner of the screen), there were two different implementation options for the user to change the parameters values of the scene's elements: either making a sliders-based UI or one based on text input. Through testing a simple implementation of each option with the users, the sliders-based UI revealed itself as the strongest candidate, since the changes to the parameters values could be performed much quicker by simple dragging a slider, allowing the users to test different values very quickly. On the other hand, if the users need to input the desired values for experimentation on a text field they first need to click on the desired text field and then press the corresponding

value numbers on the keyboard. This is a slower process when compared to sliders control and it needed to be repeated for each value the user wanted to test.

Taking the differences between a sliders-based UI and a text-based UI into account, the parameters control panel was implemented based on sliders control. Figure 4.13 shows the differences between the first prototype of the parameters panel and the final iteration, implemented using the *ofxUI* add-on. The users felt that the final iteration provided a clearer and more pleasant design then the first iteration.

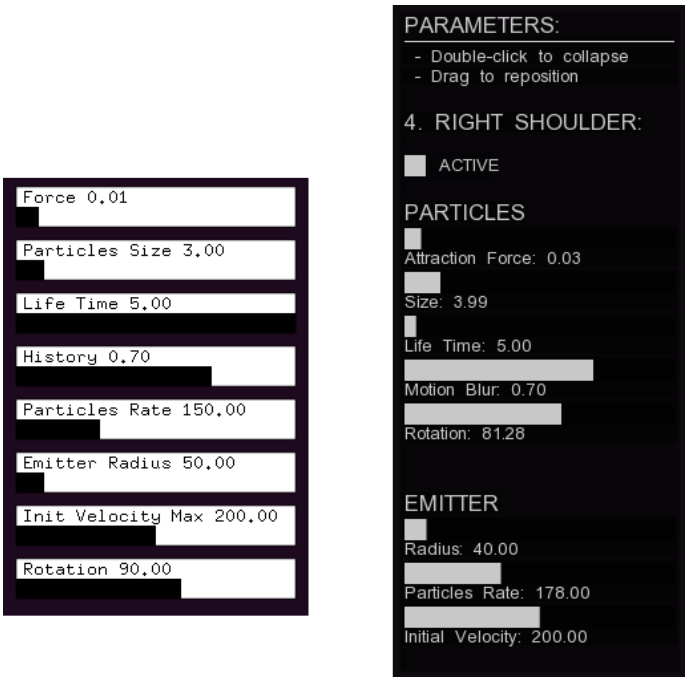



FIGURE 4.13: The first iteration of the parameters panel (left) simply provided a slider to control the value for each parameter. The final iteration (right) still provides a minimalist design but with its content much more organized. It also displays the name of the joint currently selected on the joints selector panel, since the current parameterization will be associated to that joint.

Additionally, an informative panel was added underneath the parameters panel, providing useful info for the users. This info is specifically short key commands for the user to control the interface or the scene's elements, which are the following:

- **ENTER** Hide/Show GUI elements;
- **F1** Capture and save a screenshot of the scene;

- **SPACE** Randomly reset particles position *(Particle System scene)*;
- **CTRL** Set tracking on/off for each living particle *(Particle System scene)*.

Having these key commands for minor operations like hiding the GUI, taking a screenshot, etc., made the interaction between the user and the software much more practical and efficient.

After implementing the parameters panel based on sliders-control, that component was promptly tested with an architect and a dancer/choreographer in order to validate the concept before moving onto the rest of the implementation. This was a necessary step of the development process since, throughout our research on previous related works, it was found that some artists, or other people not familiar with programming concepts, do not easily understand the concept of parameters [26]. The results and procedures of these tests are further discussed on Chapter 5, but it can be advanced that all the testers easily assimilated the concept of parameters and considered the interaction with sliders a very efficient method for controlling the parameters values.

To allow the user to set the desired colors for the scene elements, the GUI also encompasses a color picker. For implementing the color picker, it was used a Frame Buffer Object (FBO), which is an off-screen buffer to which the color picker and all its components are drawn. After having the color picker correctly rendered according to its current properties, the FBO is drawn into the right bottom corner of the screen, as seen on Figure 4.12 (for implementation details see `ColorPicker.cpp` on Section B.1 of Appendix B).

The color picker is composed of a color gradient, two or three color containers (depending on the number of customizable elements of the scene) and the display names of each container (whose text color is always contrasting with the background color), as depicted on Figure 4.14. The user must first click on a color container and only then pick a color to associate it with, by clicking or dragging the mouse inside the color gradient.

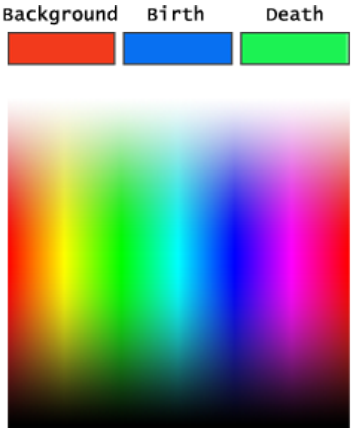



FIGURE 4.14: The color picker is composed by a color gradient and two or three color containers (*Joints Draw* or *Particle System* scene, respectively). This is a color picker as shown in the *Particle System* scene GUI.

Depending on the graphical scene the users are controlling, the color containers will be associated to the following elements:

*Particle System:*

- **Background Color** - The background color of the scene;
- **Birth Color** - The color with which the particles are born on the emitter;
- **Death Color** - The color the particles should have when they are about to die.

*Joints Draw:*

- **Background Color** - The background color of the scene;
- **Brush Color** - The color of the joints' brushes.

By experimenting with the color picker and making different color combinations the aesthetics differences they cause on the scene can be seen right away, since all the changes are processed in real-time. Figure 4.15 shows two different situations arrived by experimenting with the color picker in the particle system scene.

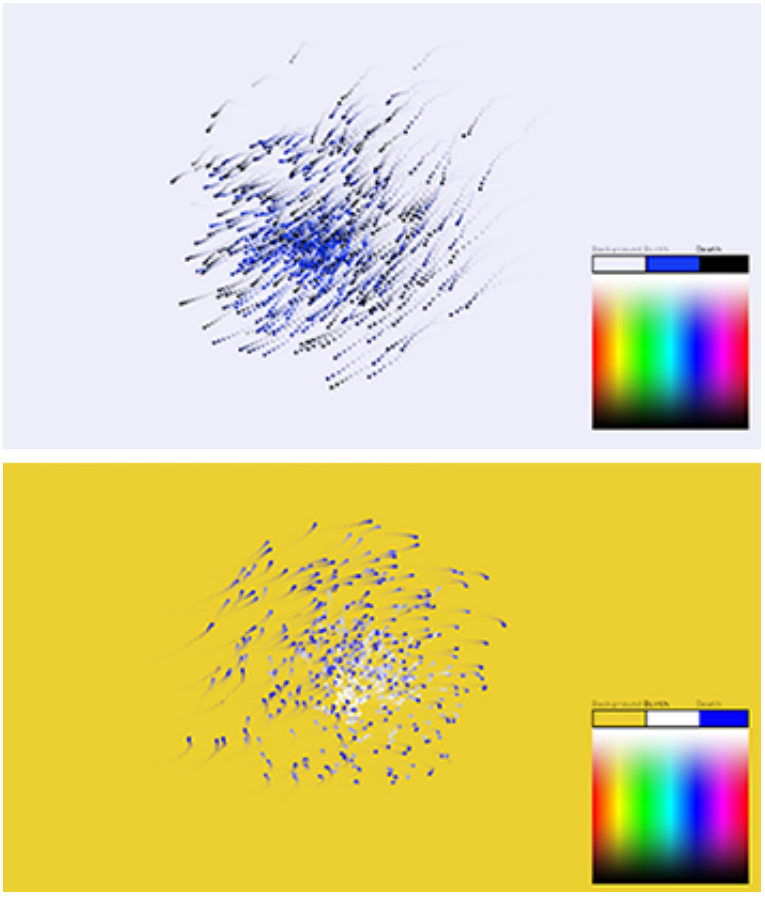


FIGURE 4.15: Two different results from experimenting with the color picker on the particles system scene. The circles represent the floating particles and their color is set according to the values on the color picker.

An FPS (Frames Per Second) counter was also added to the interactive scenes editing GUI, so that the users could keeps track of the computer performance while processing and rendering all the data from the current scene. All these GUI elements can be seen displayed on the screen on Figure 4.16.

A joint selector panel is also displayed on the interactive scenes editing GUI, which is what allows the user to control which joints from the performer's body will affect the projection graphical content. The design for the joints selector is also minimalist, simply providing a human body silhouette with circles representing each joint traceable by the Kinect sensor and lines connecting each joint circle, as seen on Figure 4.17.

When the user selects a joint from the joints selector, the parameters control


FIGURE 4.16: All the three GUI panels added to the scenes editing GUI (parameters panel, info panel and FPS counter) were implemented using the *ofxUI* add-on.

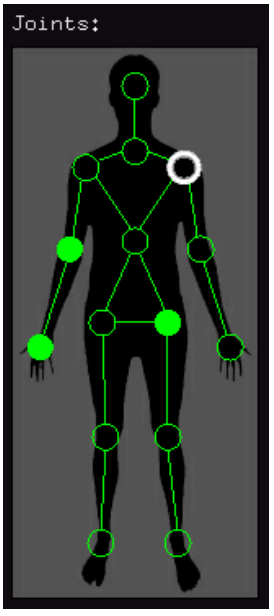



FIGURE 4.17: The joint selector panel shows a human silhouette with circles representing the skeleton joints. The currently selected joint is surrounded by a white circle and the currently active joints are filled with the color green.

panel automatically changes to the parameters panel corresponding to that joint. Therefore, when the scenes editing environment is called by the user, fifteen different parameters panels are created, but only one of them is shown at a time (depending on the joint currently selected on the joints selector). This allows the users to associate a different parameterization of the scene's element to each joint, making each body joint affect the scene's content in a different way. For completely deactivating a joint, i.e., making a performer's body joint to not affect the projection at all, the user must select it on the joints selector panel and then, on the parameters panel, click the toggle that activates/deactivates that joint.

Having the possibility to choose which joints affect the projection graphical content and how each of them affects it (e.g., different intensity values for attracting the scene's elements) the results are much more interesting. As seen on Figure 4.18, now it is possible to avoid that every particle has the same size, since now the users can parameterize the scene's content in relation to the joints properties.

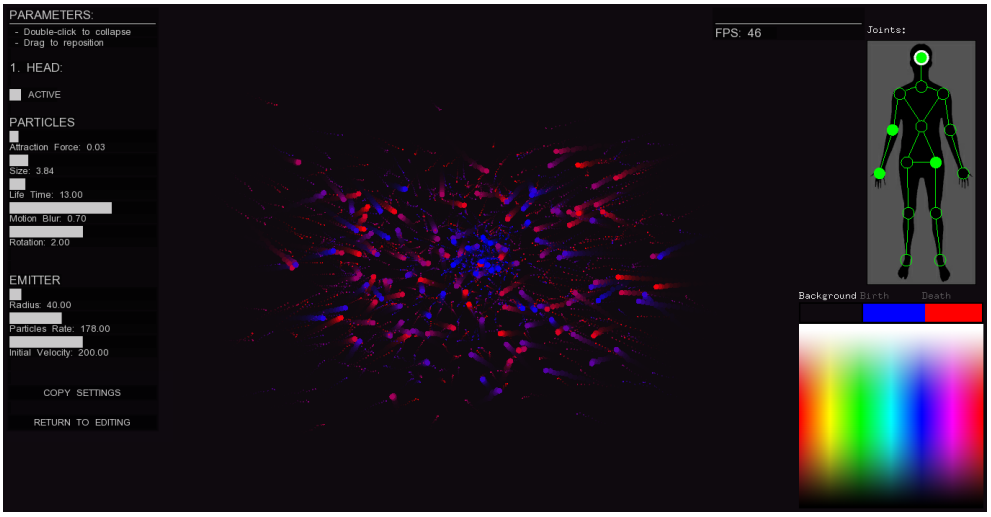



FIGURE 4.18: The parameters panel shown on the left corresponds to the currently selected joint on the joints selector panel, on the right. The displayed particles now have differences on the way they look (e.g., different sizes) and how they behave due to the differences between each joint's parametrization.

### 4.2.2 Motion Capture Algorithms

In order to actually relate the scene's elements with the body movement of the person being tracked by the Kinect camera, according to the configuration set by the user, an algorithm for detecting and storing the position of a human body in front of the camera was used. The motion capture algorithms in *MotionDesigner* were developed resorting to OpenNI and NiTE libraries.

The OpenNI libraries were used to detect if at least one Kinect sensor is connected to the computer and, if so, access its depth sensor and turn it on each time the software needs to track a person in front of the camera, typically when the users launch the projection sequence from the editing studio or when they are exploring the possible parameterizations before the live projection.

Having access to the Kinect's depth sensor, the management of the motion capture data was done resorting to the skeleton tracking algorithms already provided by NiTE middleware. To do so, when the Kinect's infrared emitter is initialized, a user tracker algorithm provided by NiTE is also initialized. This algorithm can be found in NiTE amongst others like scene segmentation, floor plane recognition and pose detection algorithms. The User Tracker algorithm finds all the active users on the scene (users being the people in front of the Kinect camera) and for each of them recognize their body boundaries and separate them from each other and from the background. This allows to distinguish the area of the depth-camera frame occupied by each performer's body.

Apart from the user tracker, NiTE also provides an algorithm for skeleton recognition. This algorithm is called at each new frame, since it needs to constantly track the position of each skeleton joint of the person in front of the camera. To manage the data gathered from this algorithm, for each detected performer in front of the Kinect camera the position data is stored in an array.

At each new frame, the array which stores the body pose data of each person in front of the Kinect sensor is accessed and, for those whose body is already recognized by the User Tracker algorithm, we access its skeleton information and

call the algorithm for recognizing each joint of that user skeleton (see Section B.2 of Appendix B for further details). By doing so, the performers' joints position and orientation in space can be obtained. These joints positions are then fed into the algorithms responsible for managing the position of the scene's elements.

The implemented GUI for editing the graphical scenes content and for managing the motion tracking properties were used in all types of graphical scenes provided in the software. In order to provide more than one scenario for the users of *MotionDesigner* to experiment with while creating their own interactive projections, two different types of graphical scenes were implemented:

1. **Particle System** - where a point in space emits a set of floating particles that track and orbit the performer's body joints;
2. **Joints Draw** - where the skeleton joints of the performer act like brushes painting on a canvas.

To find out which scenes the software would provide right from the start some interviews were conducted with a group of people from the target audience. According to the dancers and architects interviewed, these two types of graphical scenes are the most used and the most interesting types of graphical content to use in interactive performances or installations.

### 4.2.3 Particle System

A particle system is one of the interactive graphical scenes provided by *MotionDesigner* for its users to embed in their interactive projections. This type of scene was implemented since it is one of the most explored in interactive projections due to its interesting behavior and the many different and interesting shapes it assumes when interacting with the movement of a performer.

The particle system scene consists on a particles emitter, i.e., a point in space that constantly ejects particles onto the scene. The particles ejected by this emitter move independently from each other and will track a specific joint from the

skeleton(s) detected by the depth camera. Figure 4.19 illustrates a particle system composed of an emitter on the center of the screen and an attraction point.

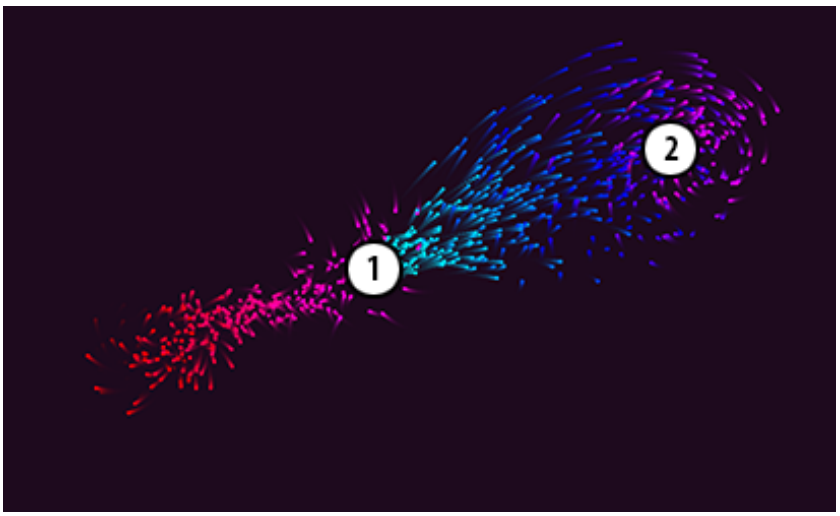



FIGURE 4.19: Typical structure of a particle system. 1 - Particles emitter; 2 - Attraction point for the particles

The parameters that characterize the particle system, i.e., the variables that would be adjustable by the final user, are [33]:

1. **Attraction Force** - the intensity with which the point of attraction pulls the particles;
2. **Particles Size** - the size of the particles radius (in pixels);
3. **Life Time** - time that the particles take to disappear from the scene (in seconds);
4. **Motion Blur** - visibility of the trail leaved by the particles motion;
5. **Rotation** - centrifugal force of the attractor point, i.e., controls the speed and orientation with which the particles orbit the target;
6. **Emitter Radius** - the size of the emitter radius (in pixels);
7. **Particles Rate** - the number of particles ejected by the emitter (in particles per second);

8. **Initial Velocity** - the maximum velocity with which the particles are ejected from the emitter.

These are the parameters that will be shown in the parameters control panel (left panel on Figure 4.18). By setting different values to each of these parameters, through the GUI elements provided on this panel, the user will affect the aspect and behavior of the particles being drawn and computed by the responsible algorithms, in real-time.

The particle system implemented is one of independently moving particles. This was decided because it would be easier to compute all the data in real-time, since each particle's movement is not affected by the surrounding particles, i.e., no collision detection, no gravitational attraction, etc. This allows us to have a larger number of particles on the screen without slowing down the computation speed.

Each particle is drawn in the screen as a 2D primitive, specifically a circle. By doing so, the drawing and managing of the rendering properties was quicker, when comparing to 3D rendering, allowing us to focus on the behavioral algorithms of the particles.

There is a single class, called `Particle`, to manage all the particles properties, since their data and functions will be reusable for each particle generated by the emitter (see Section B.3 of Appendix B). The first three functions defined after the constructor are the basic functions of any openFrameworks program. In OF typically every class has these methods implemented and they are called during the execution of the program as Figure 4.20 shows.

The arguments passed to the `update()` function of the particles are, respectively, the time passed between each calling of this function (in seconds) and the x,y and z coordinates of the destination point, i.e., the attractor's position (for prototyping we passed the mouse coordinates but later we passed the corresponding joints position).

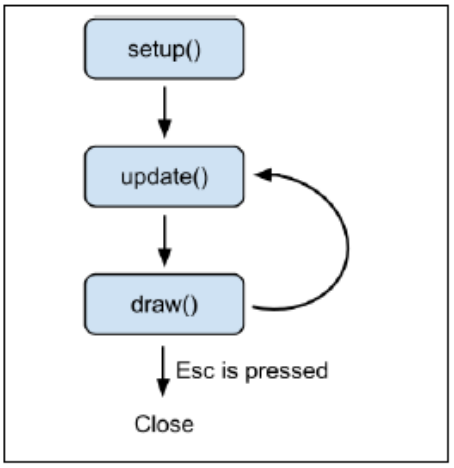



FIGURE 4.20: Basic function calling in a typical OF class. The function `setup()` is called when the class is instantiated, then the `update()` and `draw()` functions are called cyclically until the user terminates the program execution [33]

The `draw()` function has two arguments which are, respectively, the minimum and the maximum color value between which the particle will interpolate. This means that when a new particle is born it will have associated a specific color and, as its time of living approximates the instant where it is supposed to die, its color will interpolate until it reaches the value of the other color specified. This color interpolation concept was already explored by Denis Perevalov [33], but in his implementation the user can not change these values. In *MotionDesigner* the user can set, in real-time, the desired birth and death colors for the particles by using the color picker provided on the GUI.

Figure 4.21 shows a screenshot of the implemented particle system, which illustrates which parameters the users can control and to what they refer to.

Everytime a particle is born and ejected by the emitter it has associate to it a life time (time allowed for the particle to be alive), a radius size and a force value (the intensity with which the particle is pulled by the attractor), which are all parameterized by the user. The particles position and velocity are initialized with a random value within the emitter radius and the velocity radius of the particle, respectively, as depicted on Figure 4.21. This makes the particles to never be born on the exact same place nor being all ejected with the same speed.

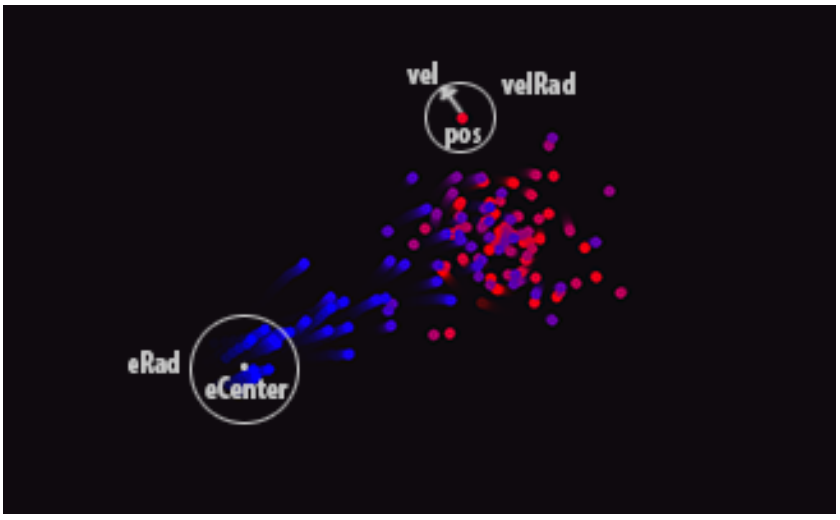



FIGURE 4.21: `eCenter` - Emitter Center; `eRadius` - Emitter radius, i.e., possible area where the particle are born; `pos` - Particle position; `vel` - Particle velocity vector; `velRad` - Particle's initial velocity limit (the particle is born with a random velocity within this limit)

At each new frame, the particles' state is updated and they are drawn to the screen according to their parameters values. For computing each particle's properties, at each new frame, their velocity vector is rotated according to the value specified by the user in the corresponding parameter, whose value ranges from -500 to 500 (positive or negative sign refers to the rotation orientation, being it counter-clockwise or clockwise, respectively). Since the particles are supposed to move on the $xy$ plane, their velocity vector is rotated on the $z$ plane (see `Particle::update()` on Section B.3 of Appendix B). The particles position is computed using the Euler method [19], which is one of the simplest numerical integration methods there is. The formula used in the Euler method is:

$$f(t0 + dt) = f(t0) + g(t0)dt \tag{4.1}$$

With Equation 4.1, each particle's position will be updated to the position it should assume regarding the velocity vector and the time passed before the last update function call. Then, the newly position coordinates are aligned towards the target, i.e., towards the particle attraction point. This is done by computing the Euclidean distance between the coordinates of the target and the particle's

current coordinates, and multiplying this difference by the intensity they are being pulled in this direction, which is also one of the parameters specified by the user. However, instead of simply multiplying the attraction force value, it is actually multiplied by a random value between half of the attraction force value and its actual value. This makes each particle to be pulled by the attraction point with different intensities, varying somewhat within the value specified by the user. This makes the particles to move in a much more interesting way as a group, which was something not explored by Perevalov's system [33].

In order for the particles to be drawn to the screen according to the values computed on the update function, simple circles primitives are drawn on the screen for each number of articles alive, according to their computed positions and radius (the latter specified by the user). Each circle is drawn filled with the color it is supposed to have in that time instant (an RGB value between the birth and death colors, specified on the color picker, according to the life time of the particle). This concept of color interpolation is illustrated in Figure 4.22.

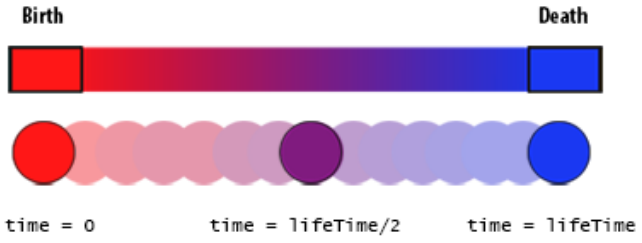



FIGURE 4.22: The color of the particles are the result of the interpolation of the Birth and Death colors (specified by the user through the color picker provided on the GUI), which is related to the distance the life time value of the particle is between 0 and the maximum life time allowed (specified by the user in the parameters panel).

For rendering the particle system as a whole, and since the user can set the intensity of the motion blur left by the particles, it was used an off-screen raster buffer, an FBO, for drawing the different particles positions over time. This FBO has the same width and height as the screen and is used for accumulating the drawings on the screen, as typically done for rendering overlapping drawing layers.

The background color specified in the color picker is set to the FBO at each new frame, so as the circles representing the particles. Therefore, the positions of the particles are drawn to the FBO accumulatively, allowing the possibility for each of them to leave a trail while moving around. Only when all the particles are drawn to the FBO and this has the rendering according to the current parameters values and colors specified by the users, the frame buffer is drawn on the screen.

Having an FBO for accumulating the particles position allows them to leave a trail while moving on the screen. But in order for the user to control the trail intensity left by the particles (motion blur) a semi-transparent rectangle with the screen dimensions is drawn inside the FBO after all the drawing operations for the particles are done. By drawing various rectangles above the particles layer (one at each new frame) the previous positions occupied by the particles will become gradually transparent until they are no longer visible. The time the particles previous positions, i.e., the particles trail, take to disappear form the screen depends on the motion blur value specified by the users on the parameters panel, e.g., if it is higher then the transparency of the rectangle will be greater and, therefore, the trail will take longer to disappear.

By using an off-screen buffer to accumulate the particles positions and a method for controlling the transparency of each drawing layer, allows the particles to have motion blur, which intensity is controlled by the user in real-time, along with other parameters like the size of the particles, the intensity with which they are attracted to the performer's body joints, their rotation speed, etc. Figure 4.23 shows a screenshot of the particle system, which shows the particles leaving a motion blur while moving on the screen.

For each particle born and ejected by the emitter there is a specific target to which they will travel. At first this target was, for every particle, the mouse coordinates (Figure 4.23 is an example of this type of implementation). Later, when the tracking was working properly, the particles would move in the direction of the body joints of the person being tracked by the Kinect. To map the particles into the positions of each body joint, each particle is born with a different target

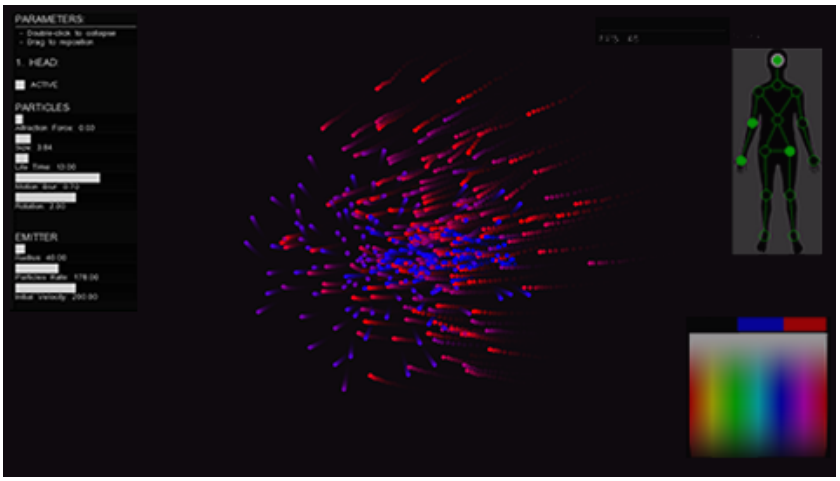



FIGURE 4.23: Visual result of the particle system implementation. Each particle leave a trail whose intensity is controlled by the user in real-time (motion blur parameter). Other parameters like the size, attraction force, rotation speed, etc., are controlled in the parameters panel on the left.

joint. Each particles has a joint index value associated to it, which ranges from 0 to 14 (to cover all the fifteen traceable joints). The value attributed to the first particle to be born is 0 and it iterates for the next particles until it reaches 14, then it resets the target to 0 for the 15th particle to be born and continues the iteration for the following particles, and so on. This allows the particles to have different target joints. The relationship between the target index and the corresponding joint can be seen on Table 4.1.

The target joint linear iteration is considered for every joint of the performer's body only if the user did not set any joint as inactive. When the user clicks on the *Active* toggle on the parameters panel, the currently selected joint will be set to the value of the toggle, e.g., if the selected joint on the joints selector panel is the right shoulder, and if its toggle on the parameters panel is off, then the right shoulder joint will not be considered as an attractor for the particles born on the emitter, making the newborn particles to ignore this joint as a target. This successfully makes the particles to travel in the direction of different points in space, which correspond to the performer's body joints traceable by the Kinect and activated on the GUI. So that the emitter does not always stay in the same position, its coordinates are also updated so it follows the performer's center of

| **Target Index** | **Target Joint** |
|---|---|
| 0 | Head |
| 1 | Neck |
| 2 | Left Shoulder |
| 3 | Right Shoulder |
| 4 | Left Elbow |
| 5 | Right Elbow |
| 6 | Left Hand |
| 7 | Right Hand |
| 8 | Torso |
| 9 | Left Hip |
| 10 | Right Hip |
| 11 | Left Knee |
| 12 | Right Knee |
| 13 | Left Foot |
| 14 | Right Foot |

TABLE 4.1: List of traceable joints and the corresponding target index. The 1st particle to be born on the emitter will trace the performer's head, the 15th particle will trace the right foot, the 16th will trace the head, and so on.

mass.

The testing and experimentation with the particle system scene, with the particles tracking the body joints of a person in front of the Kinect camera, resulted in very interesting situations which are illustrated on Figure 4.24.

### 4.2.4 Drawing With Joints

In order to have more than one scene to provide to the users and to allow them to have different materials to work within their interactive projections, another type of interactive graphical scene was implemented, which was called *Joints Draw*.

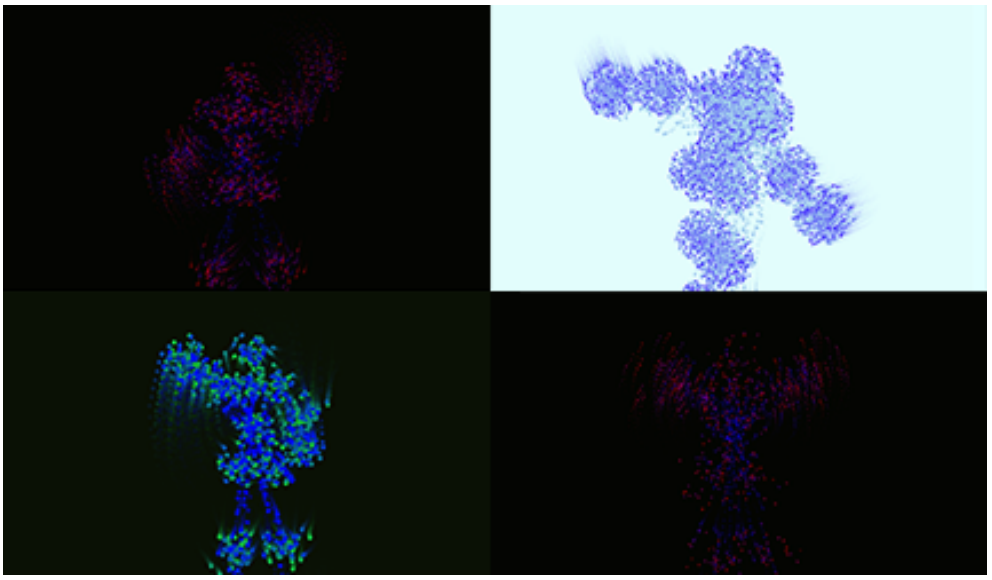


FIGURE 4.24: The particles track the performer's skeleton joints and orbit them forming an anthropomorphic swarm of particles, which is more or less recognizable as a human form depending on the current active joints and their parameterization.

In the *Joints Draw* scene the performer's skeleton joints act like brushes painting on a canvas, which is the screen. In this type of scene the parameters configurable by the user are:

1. **Brush Size** - the size of the brush radius (in pixels);
2. **Trail** - the perseverance of the trail leaved by the brush motion (how many seconds does it take for the drawing to be erased);
3. **Drawing Speed** - the intensity with which the joints move the brushes.

To allow the management of the parameterizations, behavior and rendering properties of each brush a class called `Brush` was implemented (see Section B.4 of Appendix B for further details) and instantiated fifteen times (one for each body joint traceable by the Kinect camera), storing each of them in an array, so as in the particle system implementation. The brushes are also drawn in the screen as simple circle primitives, whose size and color can be set by the user in real-time by interacting with the provided GUI (see Figure 4.25).


FIGURE 4.25: Two brushes (head and left shoulder) painting on the screen. The user can set the size of the brushes, the drawing speed and the erase time for the drawing on the parameters panel (left). Each parameterization is associated to the brush corresponding to the joint currently selected on the joints selector panel (upper right).

In order to make each traceable body joint to paint on the screen, i.e., to make each brush leave a trail, an off-screen raster buffer (FBO) was used, so as in the *Particle System* scene. The FBO is used to accumulate the drawings of each brush on the screen. It is this accumulation that makes each brush to leave a trail, which represents the drawing the performer is actually making.

As in the *Particle System* scene, it is by controlling the transparency of the semi-transparent rectangle that is drawn above the screen buffer, that the users can set the intensity of the trail leaved by the moving objects. In this case, a higher trail intensity means the drawing will take more time to be erased from the canvas. However, this time the user can not set this trail value to a value that is so low it actually does not result in a visible trail. Therefore, there is a minimum value that is allowed for the user to set on the parameters panel, which still makes the brushes to leave a trail on the screen although this trail (the drawing itself) will take a few seconds to be erased from the screen.

Figure 4.26 illustrates some results by experimenting this scene with a performer. Some moments allow a more clear reading of the performer's pose, for

instance the upper right screenshot, where the head (biggest brush) the torso and the legs are identifiable. Other moments, like the ones depicted in the bottom images, present a more abstract shape due either to the lack of joints actually painting on the screen. On the bottom left image only two joints are shown, which are not identifiable, while on the bottom right image, due to the fast movements created by the performer's body and the many active joints, the brushes' drawings overlap each other.

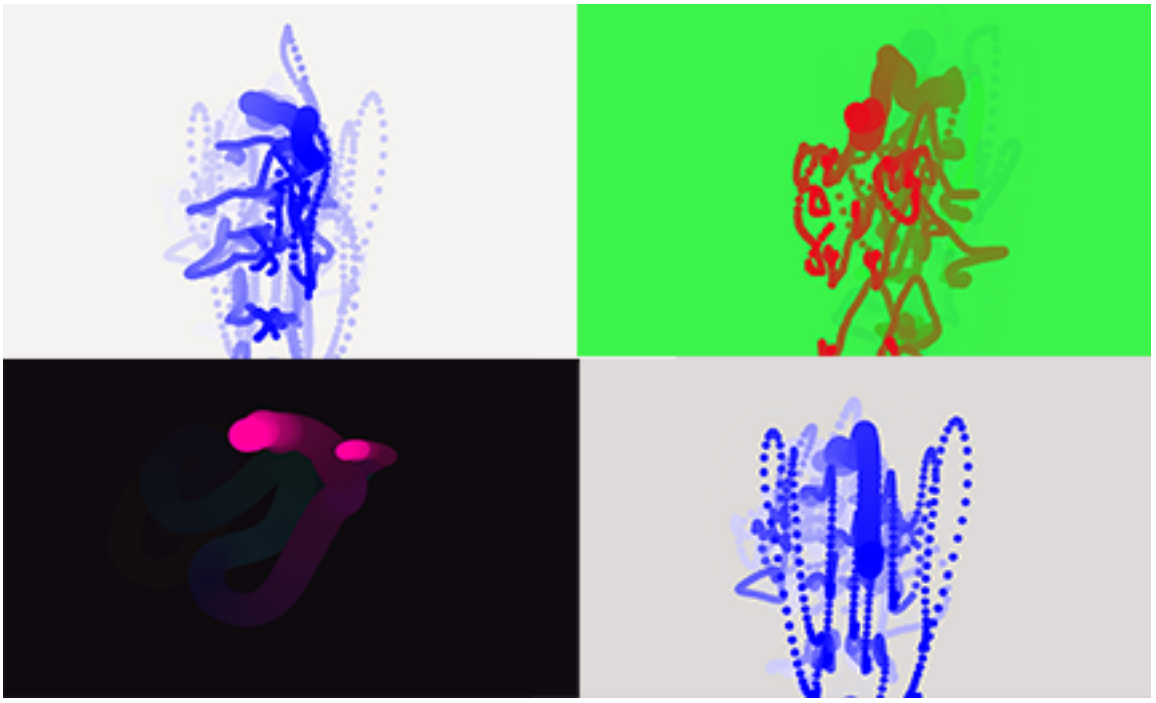


FIGURE 4.26: Some abstract shapes can be drawn on the screen by having a performer moving around and conducting each brush move on the canvas/screen. By changing the parameters values the user can set different aesthetics to drawings

## 4.3 Interactive Audio

Besides the interactive graphical scenes, *MotionDesigner* also provided its users the possibility to work with interactive audio to play along with the projection of the graphical elements.

The developed software allows the users to use a specific sound file, e.g., WAV or MP3 file, and transform its properties while it plays, according to a set of interaction rules that they set as desired. The sound properties which can be affected through the motion of the performer in front of the depth camera are the following:

1. **Volume** - the volume of the sound being played;
2. **Speed** - the speed with which the sound is being played (can be lower than the normal speed or higher);
3. **Panning** - sets how the sound is channeled through the speakers (speakers primacy).

For each of these sound properties, the user can choose which body joint of the performer will affect that property, as well as choose the position reference for that joint. This means that, after the users select a joint to affect a specific sound property, they can set one the following position references for that joint:

1. **X-Axis** - the sound is transformed according to position of the joint on the x-axis of the screen;
2. **Y-Axis** - the sound is transformed according to position of the joint in the y-axis of the screen;
3. **Distance to Marker** - the sound is transformed according to distance between the joint position and the position of a marker on the screen (which can be changed by the user);

4. **Distance to Mouse** - the sound is transformed according to distance between the joint position on the screen and the position of the mouse.

There are four different ways a single joint can transform a sound property and there are fifteen different joints that can affect that property. This opens a wide range of possibilities for the performer to interact with the sound samples being played, since there are joints that are more often moved by the performer than others, e.g., the hands have a higher tendency to be moved than the hips. By making different combinations between the joints which affect a sound property and the transformation reference for that joint, the users can have different behaviors for the soundscape of their interactive art piece, since the way the soundscape behaves do not depend only on the performer's body motion but also on the current configuration the user sets.

To allow the users to explore which joints they will want to control a specific sound sample, they can launch, on the Editing Studio, a new editing environment where the sound chosen sound sample is being played and where a GUI allows them to set the desired control settings. This is done by clicking on the Explore button associated with the sound samples aon the audio files palette of the Editing Studio, as done with the interactive graphical scenes (see Figure 4.27).


FIGURE 4.27: Explore Interactive Audio button. The sound files presented on the sounds panel display an *Explore* button for the user to run the interactive audio parameterization environment.

In order to load a specific sound file the OF class `ofSoundPlayer` was used, which allows to load a sound file of type WAV or MP3. In the case of *Motion-Designer*, WAV files are used, since these files are much faster to compute in OF,

giving the fact that MP3 files represent a compressed form of audio and so these type of files consume more CPU memory for decoding and processing their data [33].

When the interactive audio editing scene is called, the loaded sound file is immediately played. The OF sound engine will make the sound sample to be loaded instantly and played in a loop until the user exits the audio editing environment. The user can pause the sound sample playback at any time by pressing the *Space* key on the keyboard. The GUI for this audio editing environment, seen on Figure 4.28 was also developed using Reza's *ofxUI* add-on and consists on a panel with multiple drop down lists, two for each sound property so that the user can choose the desired control joint and its position reference, and text information which display the current values of each sound property.

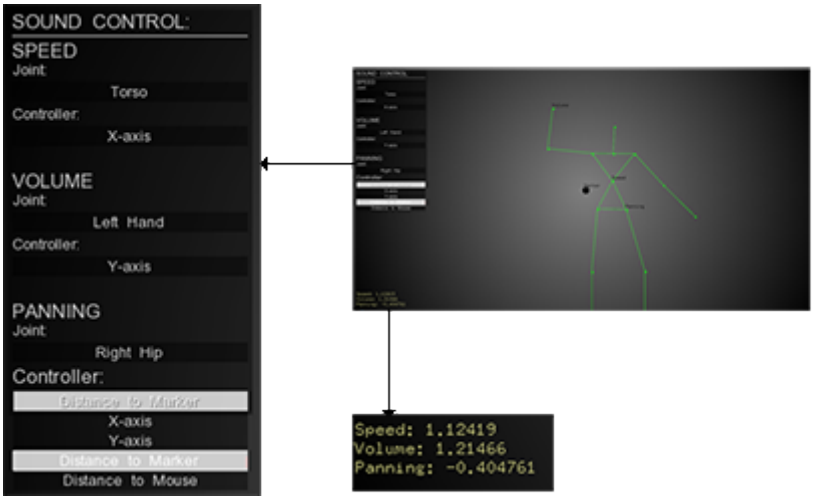



FIGURE 4.28: Interactive Audio Editing GUI. The user can set the joints responsible for transforming the properties of the sound sample being played during the projection. The panel on the left allows the user to associate a specific joint to a sound property (speed, volume and panning) and specify its position reference. On the bottom left corner the current values for each sound property are shown.

For computing the changes to be performed to the sound sample being played according to the current configuration, different operations are done. At each new frame, it is checked what are the joints and references selected on the sound control panel and, according to this data, check the corresponding joints position. For the sound speed control the corresponding joint position is read (which is known due

to the NiTE skeleton tracking algorithms) and its value is compared to the position references it should take. Only then, according to the joint position on the screen, e.g., position on the screen x-axis, the value for the speed of the sound is known, which will be a value between 0.5 and 1.5 (sound speed x0.5 or x1.5, respectively). The value to be set to the sound speed is proportional to the distance between the joint position (be it the x position, the y position, the position in relation to the marker, etc.) and the screen borders.

The proportional distance method for computing the sound speed value according to the control joint position in relation to the screen is the same for computing the other sound properties, except that each of them has different minimum and maximum values, e.g., while the sound speed can be half or the double of the normal value, the sound volume can go from 0.0 to 1.0 (which represents mute and 100% of the sound volume, respectively). The value for the sound panning ranges from -1.0 to 1.0, representing 100% of the sound being played on the left speaker or the right speaker, respectively.

Complementary to the sound properties control panel and the text display, a representation of the skeleton being tracked by the depth camera is also drawn on the screen. As seen on Figure 4.28, the performer's skeleton joints are represented through the use of small rectangles, which are inter-connected by lines to give the look of a human figure. Two circles are also drawn in the screen, representing the marker and the mouse position so that the user is always aware where these references are according to the skeleton joints position.

With the interactive audio control environment, along with the graphical scenes editing GUI implemented and accessible through the Editing Studio, the users can have real-time control over different audiovisual elements to integrate in the interactive projection they are creating. In order to reinforce and strengthen *MotionDeisgner*'s own identity, a logo was created, which can be seen in Figure 4.29.

In Figure 4.30 it can be seen a side-by-side comparison between the software's early sketches, which were functionality-driven, and its final implementation, which resulted from the interaction with the final users along the development process (details on how this interaction and its results are further discussed on Chapter 5). In this comparison it can be seen the different editing environments *MotionDesigner* provides: the Editing Studio, which is the main environment of the software and where the user can sequence and pre-configure all the audiovisual content of the projection (see section A.1 of Appendix A for the rough hand-drawn sketch); and the Interactive Scenes Editing environment, where the user can parameterize and control the projection content in real-time (for these sketches see section A.2 of Appendix A).


FIGURE 4.29: The *MotionDesigner* logo, which was created in a vector graphics editing software.

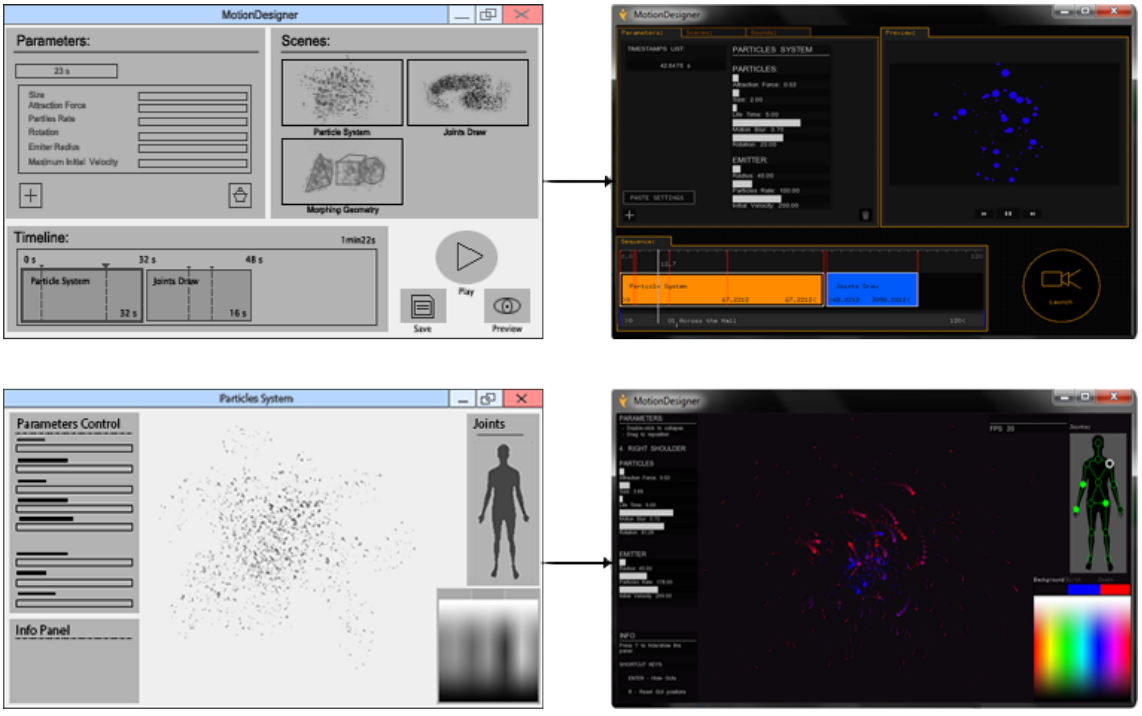



FIGURE 4.30: A side-by-side comparison between the proposed software interface as depicted on our early sketches and the final implementation.

# Chapter 5

# Evaluation and Discussion

The developed software, *MotionDesigner*, was tested with ten people from the target audience, being those choreographers, dancers, architects and multimedia artists interest in creating interactive projections for a live performance or for an installation art piece. The software was tested with these people to check how intuitive and efficient it is and to confirm if this tool is a catalyst in the creation of this type of interactive art works. The majority of these tests (six out of ten) were done at the end of the development process in order to validate its concept and efficiency, whereas the remainder were done during the development process to aid the software implementation. When the implementation was finished a build of the developed software was given to a dancer/choreographer, so that she could autonomously create an interactive projection to her liking using the developed tool.

## 5.1 Evaluation Method

To test *MotionDesigner* we conducted different sessions which consisted on presenting each interactive graphical scene to the user, as well as the editing studio environment, and ask the tester to perform a set of tasks we gave to them.

These tasks consisted on editing the content of each scene and sequence and pre-parameterize them in the timeline of the editing studio.

For gathering the ideal conditions to carry out the tests we guaranteed access to a room with 20 $m^2$ of free space and a projector installed, which, along with the Microsoft Kinect camera, would be connected to a computer running *MotionDesigner*.

The ages of the ten testers were between 19 and 33 years of age and they were professionals from different areas such as architecture, dance/performance, video art and research science. Each of these testers performed the set of tasks we proposed them without the presence of any other person in the room, apart from ourselves. This means each test session was private and the testers did not know beforehand anything about the software they were about to manipulate.

Each test session started with a brief explanation of the premise behind the creation of the tool the testers were about to use and the purpose of its implementation. Then, the testers were immediately invited to use a laptop that was running a build of our software and they were asked to perform some tasks in the Editing Studio, which is the first environment presented to the user when running *MotionDesigner*. These tasks consisted on dragging a graphical scene to the timeline (the one the user wanted), increasing the time duration of that scene, changing the instant at which the scene begins, dragging an audio file to the timeline and repeating the previous operations for the sound files. Once the testers performed these tasks successfully, they were asked to parametrize the scene or the audio sample on the timeline by adding at least one timestamp to that timeline object. Then, they were asked to change the parameters values for the newly created timestamp, delete a timestamp from the timestamps list and, finally, play the timeline sequence and preview its result on the preview player.

Once the users finished interacting with the editing studio environment, they were introduced to the interactive scenes editing interface. We asked the tester to run each of the interactive scenes by clicking on the *Explore* button (in the order they wanted) and perform another set of tasks for each of the two scenes provided.

On this part of the test one of the developers acted as the performer in front of the Kinect sensor, while the tester was operating the computer. This was done so that the content being projected had a real-time response to the movement of a third person, as it is supposed to.

For the *Particle System* scene the tester was asked to change the behavior of particles and the emitter by interacting with the parameters panel. They had freedom to change the parameters they wanted to, but we suggested them to try to change the particles size, increase the area where the particles are born, change the particles rotation, decrease the delay between the movement of the performer and the movement of the particles and decrease the number of particles in the screen. After the users interacted with the parameters panel, they were asked to change the color of the scene's background and the color the particles should have when they are about to die. Finally, it was asked that they changed the number of joints that affect the particles motion, by making the head and the hands of the performer the only joints that affect the projection content (although they had the freedom to perform other joints combination).

For the *Joints Draw* scene we asked the tester to try to deactivate the head brush, change the size of the hands brushes, changing the color of the brushes and the background, and, finally, making the drawing to be erased faster.

The evaluation process for both the editing studio and the interactive graphical scenes environments was conducted through careful observation, since we observed how each tester interacted with the software's GUI and how they performed each of the tasks from the proposed guideline. At the end of the test session, a small questionnaire was hand-out to the tester, which addressed the following questions:

1. Is the interface intuitive to be used?
2. Which of the provided parameters are the most relevant and which are dispensable?
3. Which parameters could be added to the interface?

4. Are the sliders a good and intuitive way to control the parameters values?
5. Which visual and functional aspects of the UI could be improved?
6. Do you feel you had total control over the projection content and was the creative process easier?
7. Would you prefer using a tool like this for autonomously creating and editing the projection content or rather delegate these functions to a computer programmer?
8. Did you feel any difficulty or unsuccessful carrying out your ideas using this tool?
9. Did you feel that this tool helps the creative process?

The results of these guided tests, from what we got from observation to the questionnaire answers, were diversified and allowed us to understand what was lacking in the software implementation, in terms of features and requirements, and which components of the software are more efficient and robust and which are not.

## 5.2 Results

At the end of each test session it was concluded that the tester could perform all the tasks they were proposed, regardless of the time each one of them took to do so, which was different in each case.

Since the Editing Studio was the first environment to be tested it could be seen right away that it was really intuitive for any tester to have a timeline based system for manipulating different audiovisual files. On this environment, each graphical scene was conceptually seen as separate files that the users can use and reuse, which caused many testers to make an analogy between video files and the interactive scenes. Analogously, the Editing Studio environment was compared

to the environment of any video editing software, making the users feel that the developed tool is conceptually close to a video editing software. This analogy and sense of familiarity to most of the users made its use much more effortless and intuitive, meaning that the borrowing of the video editing concepts was well done, as was supposed to.

The GUI for the interactive scenes editing environment presents the users an interface based on sliders manipulation, with which one of them referring to a parameter of the scene elements. As Jung, Doris, et al. concluded on their paper [26], the concept of parameters is something that the professionals from the arts field are, typically, not familiar with and it is sometimes difficult for them to fully understand it. However, we could see in our tests that each of the users could easily understand this concept of parameters as being characteristics of the scene elements, and that changing each parameter value was going to change the form or behavior of the elements displayed on the scene. It could also be seen that the testers made a clear correspondence between the listed parameters and the characteristics of the elements on the scene, since they altered the right parameter when attempting to change a specific characteristic of the scene elements. The reason for contradicting Jung's results may be that the people that tested *MotionDesigner* were somehow scientifically literate.

Regarding the four tests that were conducted during the development process, each test was done with a different person from the target audience. Each of these four people tested a different version of the software. These versions had different features turned on and off, i.e., a version presented some features that could not be presented in another version, which means each version had a different combination of features available. This allowed us to better understand which features are fundamental for the users. The feedback the users gave at the end or while performing the proposed tasks, by interacting with these versions, was a valuable contribute. At the end of each test the users suggested some features that they felt were needed, as well as pointed some design issues to be fixed. There were some options on the design and implementation of our system, e.g., adding or not a preview player and changing the panel distribution on the editing

studio interface, that resulted from the testers feedback or just by observing them interacting with the different versions of the system. It can be seen in Figure 5.1 that the number of suggestions, i.e., features the users felt that were missing in our implementation, decreased over time, since each version of the tested software had more features presented to them then the previous versions, e.g., the preview player, the panels organized in a tab-system, etc., covering a wider range of needs that the users feel when using the developed tool.

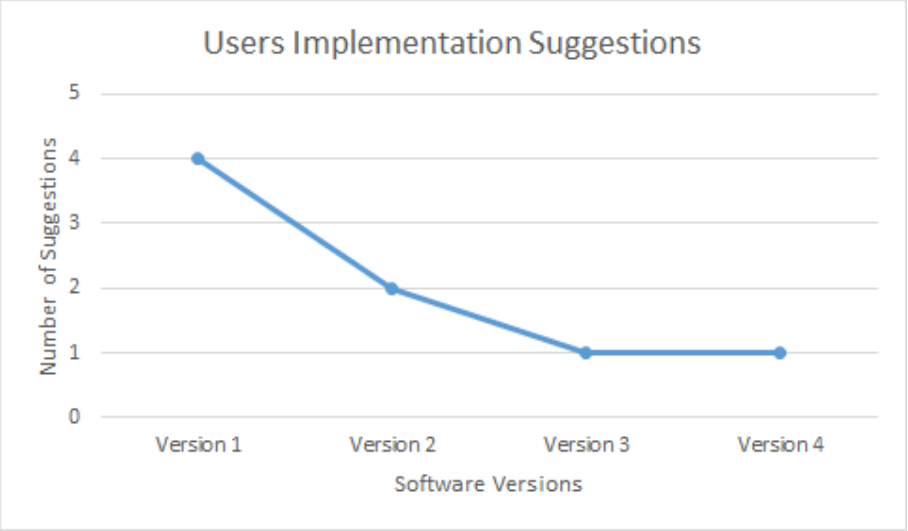



FIGURE 5.1: Number of implementation suggestions proposed by the testers according to each tested version of the software

The first version of *MotionDesigner* to be tested did not have the preview player panel and the parameterizations/timestamps system implemented on the editing studio environment. The joints selector panel was also not displayed to the users on the interactive scenes editing environment. Version 2 still did not provide the joints selector panel and iteration 3 already covered the previous features that were not available, although it lacked the sound interactivity. The fourth and last iteration to be tested was a version very similar to the final prototype version of the software (with only a few design differences).

After the final version of the system was tested with each of the six people from the target audience (those that did not tested it in an early stage) they were asked each question from the questionnaire, in order to better understand the efficiency of our implementation. The answers they gave to the questionnaire

allowed to directly know how the experience was for each tester. The results presented further are relative to these six testers that already tested the final implementation version.

The first question to be asked to the testers addressed the intuitiveness of the presented system. It was intended to know if the users felt that interacting with the system was relatively easy and if they could perform the desired tasks in an intuitive and effortless way.

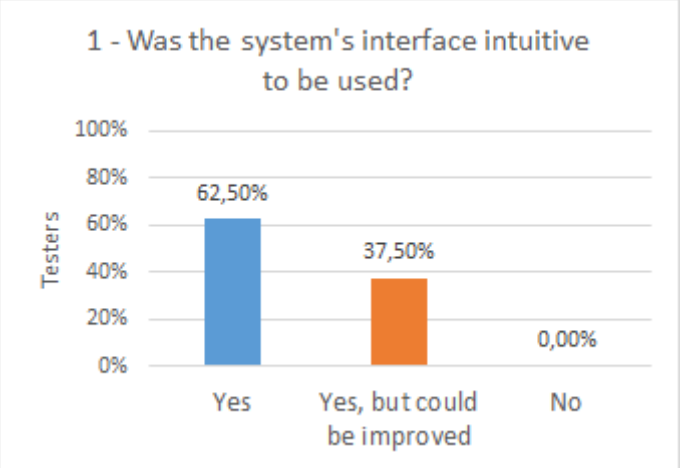



FIGURE 5.2: Graph relative to the testers feedback on the intuitiveness of the system.

It can be seen in Figure 5.2 that all the users considered that our tool was relatively intuitive to use. However, 62,50% of the testers considered that, despite of being intuitive, there are some minor changes that could be done to the GUI in order to boost even more the user experience. They suggested a few design decisions, such as changing the sliders name position from underneath to above the slider itself, as well as provide pop up text notes so the user understands what a specific button does the first time they see that interface, e.g., the *Explore* button on the scenes palette of the editing studio, etc. Other visual feedback issues like the pointing arrows that appear when hovering the border of the timeline objects (representing the fact that these are resizable elements) are examples of details that were not yet implemented when the tests were carried out and are things that some testers felt were lacking in order to have a completely effortless experience. It can also be seen by analyzing the Figure 5.2 that no tester said the developed

software was not intuitive at all and this reflects the fact that our design approach was somehow appropriate despite the issues that could be refined.

The second question from the questionnaire intended to check which parameters were the most interesting and fundamental in each scene and which were not. For both type of scenes, the *Particle System* and the *Joints Draw* scenes, all the users replied that all the parameters were relevant for affecting the scene, feeling that none of them induced redundancy on the way they affected the scene elements. Some of the testers reinforced that, for all the parameters, by changing the values of each one of them we could directly see the consequence of that change and that this consequence was obvious and coherent giving the new parameter value. These real-time results was something that pleased the testers a lot, since it catalyzes the action-reaction process when they are testing the ideas they have for that projection scene.

On the next question, the users were asked which graphical scenes could have more controllable parameters and which parameters could these be.

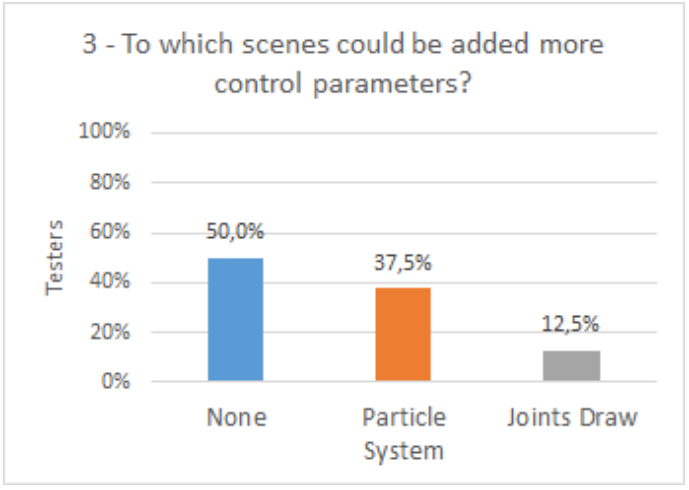



FIGURE 5.3: Graph relative to the testers feedback on which scenes could have more control parameters beyond those which are provided.

As can be seen on Figure 5.3, half of the testers did not have anything to add to any scene in terms of controllable parameters. However, the other half of the testers referred to the *Particle System* scene and/or the *Joints Draw* scene. It can be concluded that the particle system scene is the one with more fertile ground,

since, besides being the scene with more control parameters (eight, against three of the joints draw scene) it had more testers suggesting new control parameters than the joints draw scene (approximately three times the number of testers). From these suggestions the testers referred new options like controlling the speed at which the particles are born on the emitter, setting the position to the emitter so that it does not always follow the performer's center of mass, adding a jitter controller to induce instability to the particles behavior or even choosing to represent the particles as filled or unfilled circles. For the joints draw scene, a tester suggested things like the possibility to draw straight and/or dashed lines. These are all new control parameters that could be added to each respective scene, allowing the expansion of the user's degree of freedom when manipulating the contents of the projection.

After confirming that the concept of parameters was something easy to grasp for each of the testers, it was necessary to check if they also considered the sliders an easy and intuitive way to control those parameters.

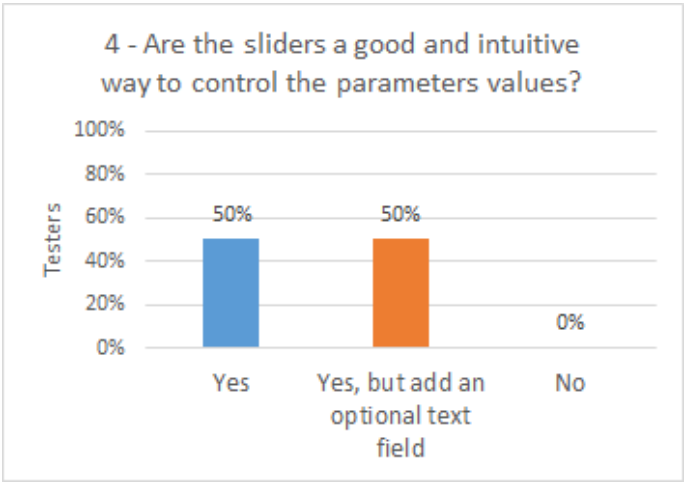



FIGURE 5.4: Graph relative to the testers feedback on the intuitiveness and practicability of having a sliders-based interface for controlling the parameters of the scene elements.

Through the results which are illustrated in Figure 5.4, it can be concluded that the implementation of a sliders based panel was a good interface design decision, since every tester considered the interaction with this interface element efficient. The sliders allowed the users to go from one value to another much higher or much

lower value in a one second operation. Despite the fact that all of the testers felt the sliders were a great interface element to control the parameters values, half of the interviewed said that they could also benefit from a text field for setting more precise values, which can be something frustrating to do when controlling the slider. The combination of these two elements could make the interaction with the GUI parameters panel much more efficient, despite having the slider as the main controller.

Still focusing on user interface polishing, the testers were asked which GUI elements could be visually or functionally improved. These GUI elements refer to each interface panel form the interactive scenes editing environment (parameters panel, info panel, color picker and joints selector) and from the editing studio (parameterizations panel, scenes and audio panels, preview player and timeline).

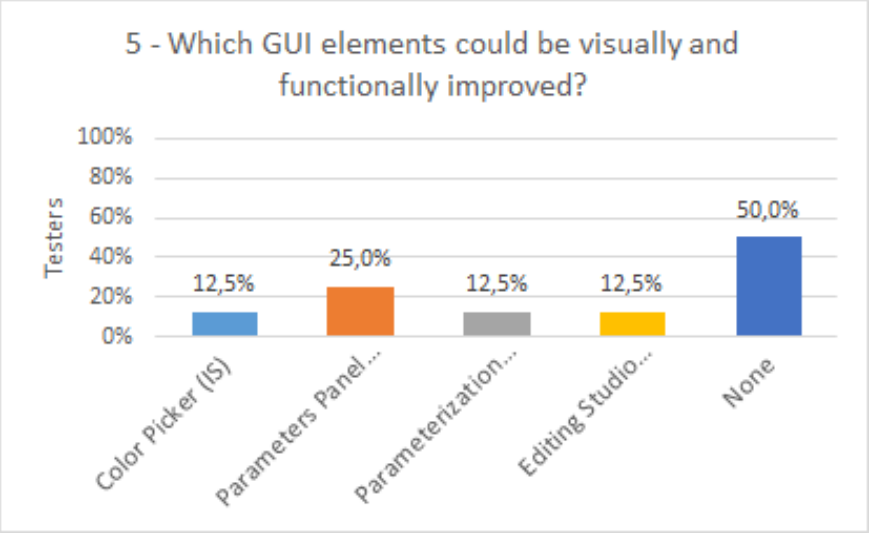



FIGURE 5.5: Graph relative to the testers feedback on the improvement of the GUI elements of each *MotionDesigner* environment

In this poll, half of the testers said that all the GUI elements were satisfactory enough for them and did not need any improvement. However, the other half referred to different GUI elements from both the interactive scenes and the editing studio environments. A small percentage of the users, 12.5%, referred to the color picker saying that its visual representation could be improved, namely the display text for the color containers names and the way the containers are differentiated from the background. The same number of testers referred to two

different elements from the other environment, the editing studio. One of these elements was the parameterizations panel, and it was suggested that the drop-down list where we can see the list of timestamps and pick one of them was immediately identifiable as so. This was posteriorly fixed by auto-opening the drop down list when the users open this panel for the first time and every time they add a new timestamp.

Another design issue referred by the testers was the general design of the editing studio, specifically the color scheme, claiming that it could not be appellative to most of the common users. A GUI element that had more suggestions, 25% of the testers, was the parameters panel from the interactive scenes editing environment. They suggested improvements like changing the color scheme for the sliders, i.e., swapping the colors between the unfilled and the filled part of the slider. These observations were taken in consideration but since we had to change the source code of the *ofxUI* add-on, which was used to implement the sliders, and since this was suggested by a considerably small number of users, these changes were not implemented.

After asking the users about the software interface, it was needed to know how they felt about the degree of freedom this tool gives them during the creative process. In order to do so, they were asked if during the time they were using *MotionDesigner* they felt they had total control over the projection content and if the experimentation process (e.g., for representing the graphical scenes) was easier having a tool like this, which is specifically oriented for the creative person's use without needing the technical person or the computer specialist. All the testers said they felt this tool helped them in creating the interactive content they idealized and that it gives them autonomy in that process.

Besides asking about the degree of freedom this tool provides to its users, we went further in the question and also asked if they prefer, during the creative process, to autonomously dictate, manipulate and sequence the content of the interactive projection or if they prefer to delegate all the technical aspects to a programmer or computer specialist. To this question all the testers, with no

exception, told that they would prefer to use a tool like this since it gives them autonomy to experiment their ideas as they come by during the creative process, which is something really desirable by most of the creative people engaging in a creation of their own.

To reinforce the testers argument on preferring to have a tool like Motion-Designer to autonomously create their interactive projections, a set of graphical scenes were proposed to the testes so that they could reimagine them. We would then implement the tester's idea right away so that he/she could measure how much time is consumed during that task and if still feels stimulated after a while. What was concluded is that everyone felt it took too long for the changes to be made through coding and that while we were still implementing the change the users wanted to make they already felt the need to experiment other ideas and not loose much time with one. The testers reinforced that with a tool oriented for the creative people, they can test different possibilities at a much faster pace and without having to depend on third-party skills. This was a great concept validation for system herein presented.

After confirming that the ideas materialization is a fundamental process of the creative process, we asked the users if they felt any difficulty in carrying out their own ideas for the interactive projection they were supposedly creating with tour software. As can be seen on Figure 5.6, all the users felt successful on achieving the results they wanted when creating their own interactive projections. A considerable number of the testers also said they felt this tool could make them reach new ideas, since they can not only implement in real-time the ideas that emerge but also, through experimentation, arrive to new and different ideas that were not foreseen. This is a very useful aspect since the artists do not always know what they wants and since a set of graphical scenes is already provided for them to experiment with, new ideas can emerge and the creative process will hopefully be easier and productive.

In order to confirm if the developed software really facilitates the creative process of such interactive works, the testers were directly asked if they felt this

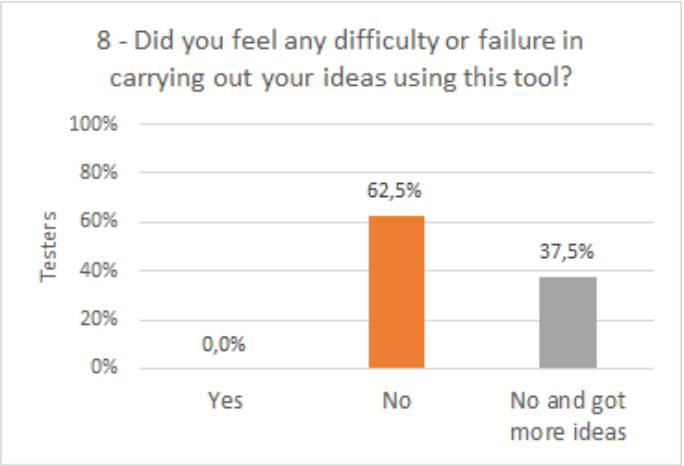



FIGURE 5.6: Graph relative to the testers feedback on materializing their ideas for an interactive projection using *MotionDesigner*

process facilitation, to which all of them answered positively, as seen on the results presented on Figure 5.7. However, a quarter percent of the testers said that they felt the tool had a complicated learning curve, since it is harder for the user to master the manipulation of each system component. Despite this, they immediately stated that after some practice this tool is something really efficient and productive. They also suggested making a brief introduction tutorial when running the software for the first time, which is something considered for further software iterations.

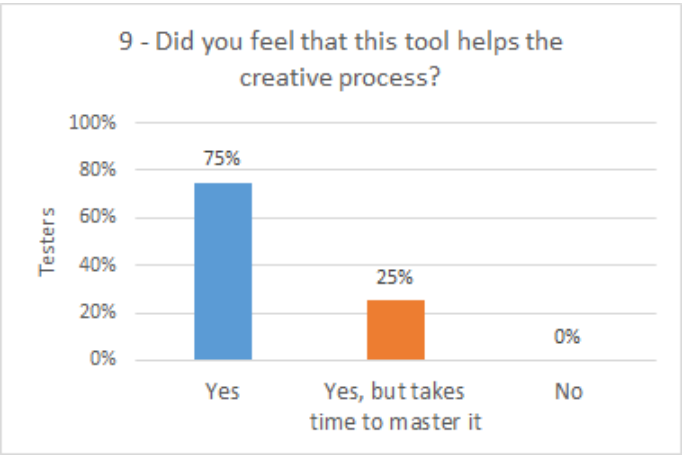



FIGURE 5.7: Graph relative to the testers feedback on materializing their ideas for an interactive projection using *MotionDesigner*

After finishing the questionnaire the users had freedom to suggest other features or simply give an overall feedback. Most of the users did not cover any issue that they did not addressed already when answering the questionnaire. However, two testers said they wanted to try acting as the performer in front of the depth-camera and interact directly with the projection content. This is a very interesting scenario that was not foreseen when outlining the tests. In this scenario the tester was directly interacting with the projection content, e.g., the particle system scene, and exploring the shapes those scene elements could assume, while asking one of us to operate the computer and change the parameters values to their liking. This is another way for the artists to explore, in real-time, the possible representations of the scene elements and find an aesthetic that pleases them, but was something that we did not expected and found as a very interesting scenario.

One of the testers, which is a dance degree student, was chosen to act as the performer and she also showed interest in participating in the creation of a small interactive performance. Therefore, the developed tool was used to create a small dance performance, in which both the particle system scene and the joints draw scene were used as the projection content. Figures 5.8 and 5.9 show some moments of the performance rehearsal with the dancer acting as the performer and one of us as the conductor of the performance. All the projection aesthetic was found by following the dancer's artistic vision.

Through the evaluation results a very positive conclusion can be made, since they validate the concept of the proposed system and confirm that the developed tool is a significant aid in the creation of interactive projections for art performances or installations, allowing the creative person to test many different and interesting scenarios. A great majority of the testers also said that they would acquire a copy of *MotionDesigner* and actually use it in the creation of future artistic works, which was something that reinforced the potential of the developed tool.

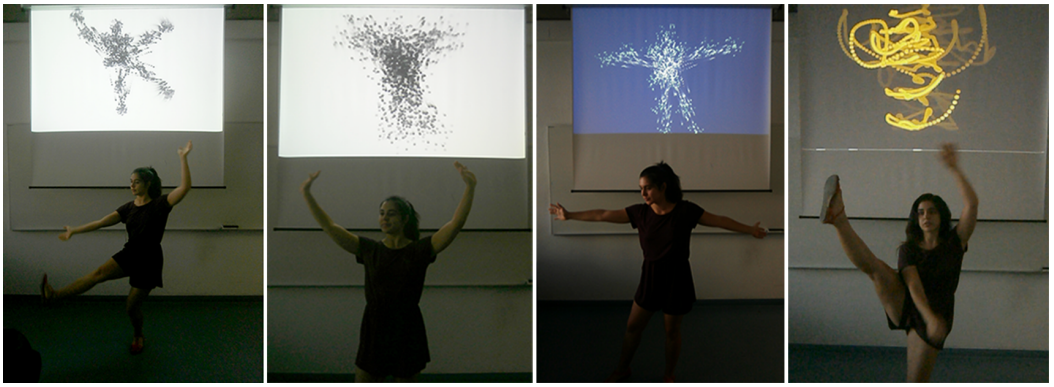


FIGURE 5.8: Rehearsal moments for the small interactive performance that was prepared. The dancer is performing a choreography while the graphics projected on the wall react to her movement in real-time. On the first three images the *Particle System* scene is being projected and on the last one is the *Joints Draw* scene. The user can mirror the projection content in relation to the body pose if wanted, as seen on the first image.

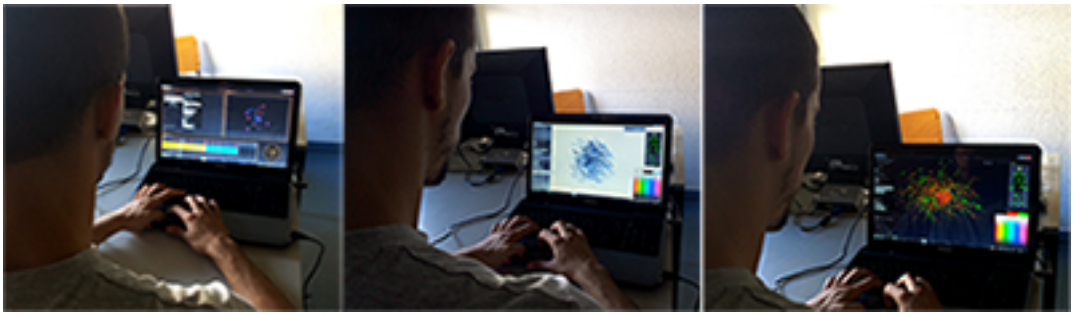


FIGURE 5.9: The user sequences the scenes he is going to use in the projection and sets a few initial parameterizations (left image), then he controls the representation and behavior of the scenes by changing the parameters values in real-time.

# Chapter 6

# Conclusions and Future Work

## 6.1 Conclusions

The system herein proposed intends to help creative people to create an interactive projection using RGB-D cameras and give them freedom and autonomy to control all the audiovisual elements in real-time. Using the developed tool, *MotionDesigner*, the user is be able to dictate which graphical elements are going to be projected and their order of appearance in the projection, as well as editing its visual representation and behavior through a high-level interaction with the system, made possible due to a GUI oriented for the artist.

The literature survey highlighted a gap on the existing tools that aim to aid the artists to autonomously create an interactive projection for art performances or installations, which is the scarcity of materials given within these tools (typically they have a single type of graphical scene to be manipulated) and the lack of control over the sounds played during the projection. Therefore, instead of providing just one case, the developed tool provides two different interactive graphical scenes for the users to embed in the projection, providing an environment where they can sequence these scenes and edit them according to a given aesthetic logic. The users also have the possibility to use the desired sound files and relate them with the movement of the performer. Ultimately, the developed software covers both

graphical and audio interactivity and gives the user the possibility to parameterize the audiovisual content in real-time, either during the live projection or before projecting the audiovisual sequence (pre-parameterizations on the editing studio). This pre-parameterization feature was not found in any other tool that has the same focus as the one herein presented and it was something that pleased the users, since by using timestamps/markers on the timeline the person creating the projection can act alone as both the conductor and the performer.

Based on the evaluation tests results and the feedback given by different people from the target audience, it can be concluded that the kind of setup herein proposed and all the features developed worked successfully on achieving its goal. All the users from the target audience that tested *MotionDesigner*, from architects to dancers/choreographers, achieved the results they wanted and recognized it as a catalyst for the creative process. The interviewees from the target audience considered both interactive graphical scenes provided in the software as two types of material they would like to use in an interactive projection due to the many different results they can obtain by manipulating each of them.

The testers in general considered the developed tool as a help for achieving new ideas during the creative process, since through experimenting with the provided materials and their possible configurations they can arrive to scenarios not imagined before. Adding to the creative possibilities the developed tool opens for its users, its interface design showed to be very intuitive and the sliders-based control and the timeline-based environment revealed to be a very simple and efficient method for the users to quickly parameterize the scenes elements and sequence them over time, respectively.

The real-time interactivity between the user (the creator and conductor of the interactive projection) and the projection content showed to be one of the strongest valencies of the presented system. Various testers referred that one of the best things about the developed tool is that they can change the parameters that characterize the interactive scenes elements and the result of those changes are seen immediately. Complementary, the users can pre-parameterize each scene they

are going to project, through the timestamps system implemented in the editing studio environment. This pre-parameterization feature allowed each user to be the curator of the projection as well as the performer that affects the projection, which is a scenario that, to the extent of our knowledge, was not covered in any other tool before.

Through the feedback given by the artists and architects that tested *MotionDeisgner*, it can be concluded that, despite all the features and material that could be added further in the future, the developed tool already provides the users sufficient autonomy to develop an interactive projection. The users that tested the software considered that this single tool already provides them a significant degree of freedom to control the projection audiovisual content. The creative people involved in the work herein presented referred to *MotionDesigner* as a great catalyst and facilitator of their own work when creating an interactive performance with audiovisuals reacting to a performer's movement in real-time. Its simple interface, according to the users opinions, allows them to easily and intuitively choose what is going to be projected, when these elements are revealed to the audience, how they are represented and how the interaction between the performer and the projection unfolds over time, which reinforces the advantage of using a tool as the one developed. Although the degree of freedom given to the users can always be maximized, a tool with the design and features herein presented can already significantly help the artists to create their own interactive projections to embed in their art works.

## 6.2 Future Work

Despite the positive impact that *MotionDesigner* had amongst the people from the target audience, there are small improvements that could be made, whether functional or merely visual enhancements, which could expand the efficiency of the proposed tool and the comfort it brings to its users. These improvements, which we pretend to implement in the near future, are:

- Allowing specific body poses to be the cue for starting or ending a scene or a sound sample from the timeline (rather than relying solely on time);
- A basic sound synthesizer module or implementing OSC routers to allow the communication between *MotionDesigner* and third-party software for music production, e.g., Ableton's Live [13];
- A Save/Load Project feature on the editing studio environment, allowing the user to save the current timeline content and status, in order to continue editing them later;
- An interface panel for controlling the RGB-D cameras so that, if the users connect more than one Kinect, they can choose which of the cameras is collecting the performer's motion data in a given time interval;
- A flag for activating/deactivating all the traceable joints at once.

There are also some questions that the work herein presented raised and that may induce a conceptual leap, namely:

- Why did some artists struggle with the concept of parameters using other tools [26] and not with *MotionDesigner*?
- If the users had the possibility to embed their own graphic material (e.g., image and video files) would it give them a higher sense of control over the projection content?
- Which creative possibilities arise by allowing the performer's pose to trigger other audiovisual elements on the scene?

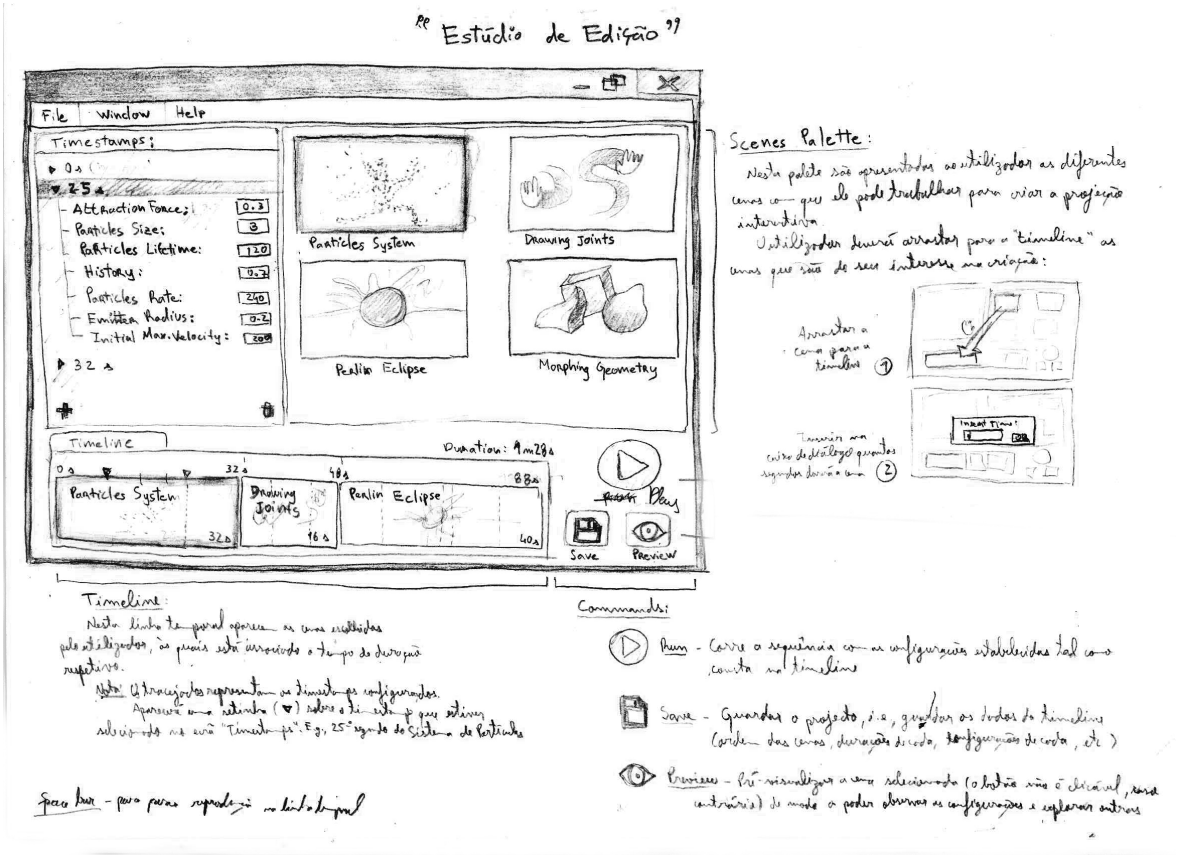
We believe that answering these questions will allow the artists to have an even more prepared tool for them to use when creating their own interactive audiovisual work as well as possibly broaden the context in which the developed tool can be used.

# Appendices

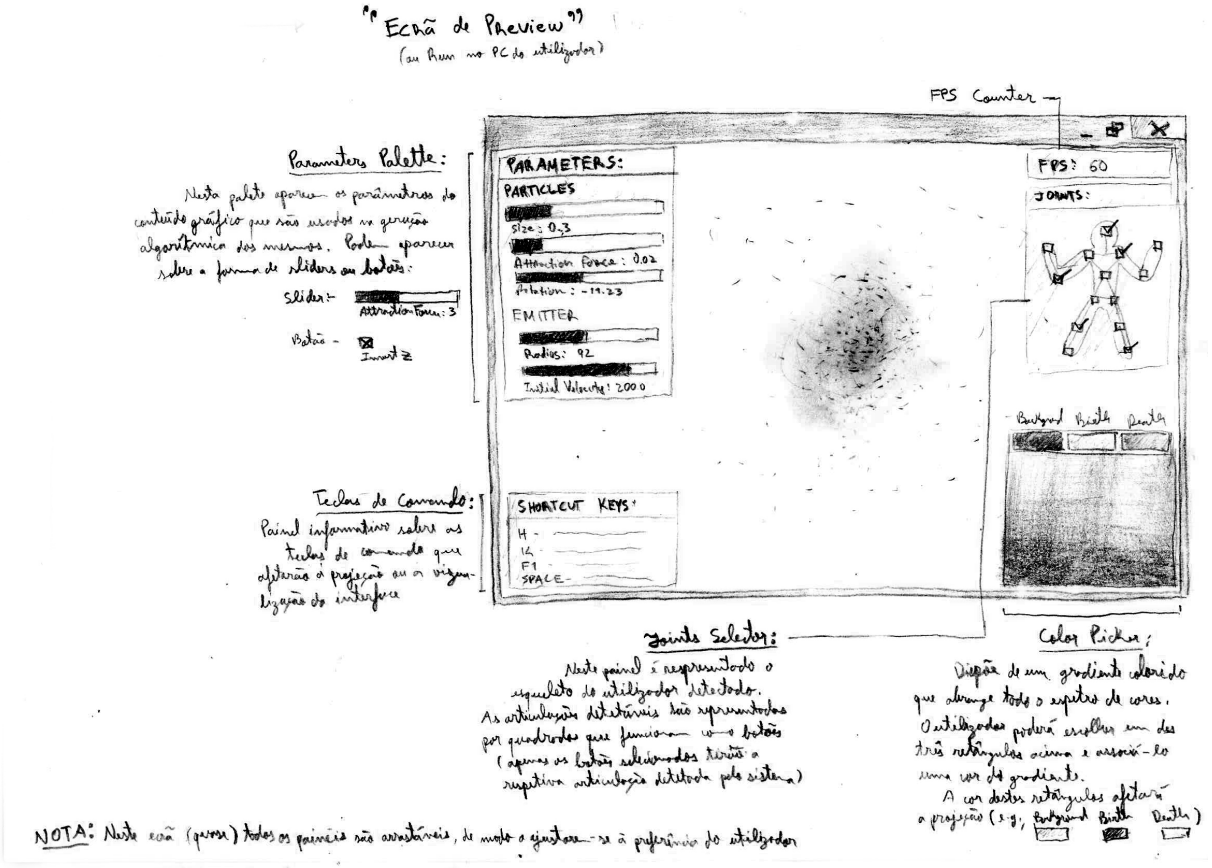
# Appendix A

# GUI Design Sketches

## A.1 Editing Studio



## A.2 Interactive Scenes



# Appendix B

# Source Code Snippets

## B.1 Interactive Scenes GUI

Slider.h:

```
string title;
ofRectangle rect;
float *value;
float minV, maxV;
```

Interface.h:

```
class Interface {
public:
  void setup();
  void draw();
  void addSlider( string title, float *value, float minV, float maxV );

  vector<Slider> slider;    //Array of sliders
  int selected;     //Index of selected slider
};
```

Interface.cpp:

```
void Interface::draw(){
  for (int i=0; i < slider.size(); i++) {
    Slider &s = slider[i];
    ofRectangle r = s.rect;
    ofFill();
```

```
    ofSetColor( 255, 255, 255 );
    ofRect( r );
    ofSetColor( 0, 0, 0 );
    ofNoFill();
    ofRect( r );
    ofFill();
    float w = ofMap( *s.value, s.minV, s.maxV, 0, r.width );
    ofRect( r.x, r.y + 15, w, r.height - 15 );
    ofDrawBitmapString( s.title + " " + ofToString( *s.value, 2 ), r.x + 5,
     r.y + 12 );
  }
}
```

ColorPicker.h:

```
class ColorPicker{
public:
  ofFbo fbo;
  float rectHeight;
  float margin;
  ofImage img;
  ofColor color1, color2, color3, textColor;
  ofColor getColor1(), getColor2(), getColor3();
  float colorPickerIndex;
};
```

ColorPicker.cpp:

```
void ColorPicker::setup(){
  img.loadImage("color-picker.png");
  rectHeight = 25;
  margin = 5;
  fbo.allocate(img.width, img.height + rectHeight + margin, GL_RGB32F_ARB);
  color1 = ofColor(15,10,15);
  color2 = ofColor(ofColor::red);
  color3 = ofColor(ofColor::blue);
  textColor = ofColor(ofColor::black);
  colorPickerIndex = 0;
}

void ColorPicker::draw(){
  fbo.begin();
  ofEnableAlphaBlending();
  //Draw color gradient image
  ofSetColor(255,255,255);
  img.draw(0, rectHeight + margin);
  //Draw left container
```

```
  ofSetColor(color1);
  ofFill();
  ofRect(2, 2, img.getWidth()/3, rectHeight);
  //Draw right container
  ofSetColor(color2);
  ofFill();
  ofRect(img.getWidth() - (img.getWidth()/3)-2, 2, img.getWidth()/3, rectHeight);
  //Draw middle container
  ofSetColor(color3);
  ofFill();
  ofRect(4 + img.getWidth()/3, 2, img.getWidth()/3, rectHeight);
  ofDisableAlphaBlending();
  fbo.end();

  glEnable(GL_BLEND);
  glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
  ofSetColor(255,255,255);
  fbo.draw(ofGetWidth() - fbo.getWidth()-20, ofGetHeight() - fbo.getHeight()-20);
}
```

Interface.cpp:

```
void Interface::setupGUI(float *force, float *size, float *lifeTime,
 float *history, float *rotate, float *eRad, float *bornRate, float *velRad){
  gui = new ofxUISuperCanvas("PARAMETERS:");

  sP.force = force;
  sP.size = size;
  sP.lifeTime = lifeTime;
  sP.history = history;
  sP.rotate = rotate;
  sP.eRad = eRad;
  sP.bornRate = bornRate;
  sP.velRad = velRad;

  gui->addLabel("PARTICLES");
  gui->addSlider("Attraction Force",0.0,0.5,*sP.force)->
    setTriggerType(OFX_UI_TRIGGER_ALL);
  gui->addSlider("Size",0.0,30.0,*sP.size_PS)->setTriggerType(OFX_UI_TRIGGER_ALL);
  gui->addSlider("Life Time", 0.0, 120.0, *sP.lifeTime_PS)->
    setTriggerType(OFX_UI_TRIGGER_ALL);
  gui->addSlider("Motion Blur", 0.0, 1.0, *sP.history)->
    setTriggerType(OFX_UI_TRIGGER_ALL);
  gui->addSlider("Rotation", -500, 500, *sP.rotate)->
    setTriggerType(OFX_UI_TRIGGER_ALL);
  gui->addLabel("EMITTER");
  gui->addSlider("Radius",0.0,500.0,*sP.eRad)->setTriggerType(OFX_UI_TRIGGER_ALL);
  gui->addSlider("Particles Rate", 0.0, 500.0, *sP.bornRate)->
```

```
    setTriggerType(OFX_UI_TRIGGER_ALL);
  gui->addSlider("Initial Velocity", 0.0, 400.0, *sP.velRad)->
    setTriggerType(OFX_UI_TRIGGER_ALL);
  gui->autoSizeToFitWidgets();
  ofAddListener(gui1->newGUIEvent, this, &Interface::guiEvent);
}
```

## B.2 NiTE

testApp.cpp:

```
void testApp::update(){
  . . .
  const nite::Array<nite::UserData>& users = userTrackerFrame.getUsers();
  for (int i = 0; i < users.getSize(); i++){
    const nite::UserData& user = users[i];
    updateUserState(user,userTrackerFrame.getTimestamp());
    if (user.isNew())
      userTracker.startSkeletonTracking(user.getId());
    else
      if (user.getSkeleton().getState() == nite::SKELETON_TRACKED)  {
        const nite::SkeletonJoint& head =
          user.getSkeleton().getJoint(nite::JOINT_HEAD);
        const nite::SkeletonJoint& neck =
        user.getSkeleton().getJoint(nite::JOINT_NECK);
        const nite::SkeletonJoint& handL =
        user.getSkeleton().getJoint(nite::JOINT_LEFT_HAND);
        . . .
        const nite::SkeletonJoint& footL =
        user.getSkeleton().getJoint(nite::JOINT_LEFT_FOOT);
        const nite::SkeletonJoint& footR =
        user.getSkeleton().getJoint(nite::JOINT_RIGHT_FOOT);

        headPos = ofPoint(head.getPosition().x + ofGetWidth()/2,
          -(head.getPosition().y - ofGetHeight()/2),
          -head.getPosition().z/param.Zintensity);
        neckPos = ofPoint(neck.getPosition().x + ofGetWidth()/2,
          -(neck.getPosition().y - ofGetHeight()/2),
          -neck.getPosition().z/param.Zintensity);
        handLPos = ofPoint(handL.getPosition().x + ofGetWidth()/2,
          -(handL.getPosition().y - ofGetHeight()/2),
          -handL.getPosition().z/param.Zintensity);
        . . .
        footLPos = ofPoint(footL.getPosition().x + ofGetWidth()/2,
```

```
          -(footL.getPosition().y - ofGetHeight()/2),
          -footL.getPosition().z/param.Zintensity);
        footRPos = ofPoint(footR.getPosition().x + ofGetWidth()/2,
          -(footR.getPosition().y - ofGetHeight()/2),
          -footR.getPosition().z/param.Zintensity);

        //Update the particles position
        for (int i = 0; i < particles.size(); i++){
          switch (particles[i].getTargetJoint()){
          case 0:
            destX = headPos.x;
            destY = -headPos.y ;
            destZ = headPos.z;
            break;
            . . .
          case 14:
            destX = footRPos.x;
            destY = -footRPos.y;
            destZ = footRPos.z;
            break;
          }
          particles[i].update(dt, destX, destY, destZ);
      }
  }
}
```

## B.3 Particle System

Particle.h

```
\#pragma once
\#include "ofMain.h"

class Particle{
public:
Particle();     //Constructor
void setup();     //Start particle
void update(float dt, float destX, float destY, float destZ); //Compute physics
void draw(ofColor pColor1, ofColor pColor2);  //Draw particle

ofPoint pos;      //Current position
ofPoint vel;      //Current velocity
float time;     //Time since birth
float lifeTime;     //Maximum life time
```

```
float size;     //Maximum particle size
bool live;      //Is particle alive
float force;      //Intensity of attraction
bool tracking;      //Is particle tracking the attractor
};
```

Particle.cpp

```
void Params::setup(){
eCenter = ofPoint(ofGetWidth()/2, ofGetHeight()/2, 1);
eRad = 40;
bornRate = 178;
bornCount = 0;
velRad = 200;
lifeTime = 5.0;
rotate = 2;
size = 1;
force = 0.03;
tracking = true;
}
```

Particle.cpp:

```
void Particle::setup(){
pos = param.eCenter + randomPointInCircle(param.eRad);
vel = randomPointInCircle(param.velRad);

time = 0;
lifeTime = param.lifeTime_PS;
live = true;
size = param.size_PS;
force = param.force;
tracking = param.tracking;
}
```

```
void Particle::update(float dt, float destX, float destY, float destZ){
if(live){
vel.rotate(0, 0, param.rotate * dt);
pos += vel * dt;
if(tracking){
pos.x += (destX - pos.x) * ofRandom(force/2, force);
pos.y += (-destY - pos.y) * ofRandom(force/2, force);
pos.z += (destZ - pos.z) * ofRandom(force/2, force);
}

time += dt;
if(time >= lifeTime)
```

```
live = false;
}
}
```

testApp.cpp:

```
ofFbo _fbo;
float _time0;

float trail;
Params param;
deque<Particle> particles;

void testApp::setup(){
ofSetFrameRate(60);
int w = ofGetWindowWidth();
int h = ofGetWindowHeight();
_fbo.allocate(w, h, GL_RGB32F_ARB);

_fbo.begin();
ofBackground(10,10,10);
_fbo.end();

_time0 = ofGetElapsedTimef();
destX = ofGetWidth()/2;
destY = -ofGetHeight()/2;
destZ = 1;
param.setup();
trail = 0.7;
}


void testApp::update(){
float time = ofGetElapsedTimef();
float dt = ofClamp(time - _time0, 0, 0.1);
_time0 = time;

int i = 0;
int bornN = 0;
while(i < particles.size()){
if (!particles[i].live)
particles.erase(particles.begin()+i);
else
i++;
}

param.bornCount += dt * param.bornRate;
if(param.bornCount >= 1){
```

```
bornN = int (param.bornCount);
param.bornCount -= bornN;
for (int i = 0; i < bornN; i++){
Particle newP;
newP.setup();
particles.push_back(newP);
}
}
for(int = 0; particles.size(); i++)
particles[i].update(dt, mouseX, mouseY, 1);
}

void testApp::draw(){
float alpha = 0;
ofBackgroundGradient(ofColor::gray,ofColor(_colorPicker.getColor1()),
OF_GRADIENT_CIRCULAR);

//1. Drawing to buffer
_fbo.begin();
ofEnableAlphaBlending();
alpha = (1 - trail) * 255;
ofSetColor(255, 255, 255, alpha);
ofFill();
ofRect(0, 0, ofGetScreenWidth(), ofGetScreenHeight());
ofDisableAlphaBlending();

//Draw the particles
ofFill();
for (int i = 0; i < particles.size(); i++)
particles[i].draw(ofColor::blue, ofColor::red);
_fbo.end();

//2. Draw the buffer on the screen
ofSetColor(255, 255, 255);
_fbo.draw(0, 0);

//3. Draw the interface on the screen
interf.setup();
interf.addSlider( "Force", &param.force, 0.0, 0.1);
interf.addSlider( "Particles Size", &param.size, 0, 30);
interf.addSlider( "Life Time", &param.lifeTime, 0, 30 );
interf.addSlider( "History", &history, 0, 1 );

interf.addSlider( "Particles Rate", &param.bornRate, 0, 500 );

interf.addSlider( "Emitter Radius", &param.eRad, 0, 500 );
interf.addSlider( "Max. Init. Velocity", &param.velRad, 0, 400 );
interf.addSlider( "Rotation", &param.rotate, -500, 500 );
```

```
drawInterface = true;
}
```

## B.4 Joints Draw

Params.cpp:

```
void Params::setup(){
size_JD = 20;
speed = 0.1;
lifeTime_JD = 50.0;
drawing = true;
}
```

testApp.cpp:

```
void testApp::setup(){}
  \\Setup Params and Interface
  . . .
  //Set up brushes
  float nBrushes = 15;
  for (int i = 0; i< nBrushes; i++){
    Brush newB;
    newB.setup();
    _brushes.push_back(newB);
    cout << "Created brush no. " << i << "\n";
  }

  //Setup NiTE
  . . .
}

void testApp::update(){
  //Gather joints position data through NiTE algorithms
  . . .
  for (int i = 0; i < _brushes.size(); i++){
    switch (_brushes[i].getTargetJoint()){
    case 0:
      destX = headPos.x;
      destY = -headPos.y ;
      destZ = headPos.z;
      break;
    . . .
```

```
    case 14:
      destX = footRPos.x;
      destY = -footRPos.y ;
      destZ = footRPos.z;
      break;
    }
    _brushes[i].update(dt, destX, destY, destZ);
  }
}
```

# Bibliography


[1] Cinder. http://libcinder.org/, 2015. [Online; accessed 19-January-2015].

[2] Developing with Kinect. https://www.microsoft.com/en-us/kinectforwindows/develop/default.aspx, 2015. [Online; accessed 26-August-2015].

[3] Digital Art Museum. http://digitalartmuseum.org/history/, 2015. [Online; accessed 19-January-2015].

[4] Hack the Art World. http://hacktheartworld.com/, 2015. [Online; accessed 19-January-2015].

[5] Max is a visual programming language for media « Cycling 74. https://cycling74.com/products/max/, 2015. [Online; accessed 19-January-2015].

[6] Microsoft Kinect. http://www.xbox.com/kinect, 2015. [Online; accessed 25-August-2015].

[7] Mind the Dots! - Preparing for live on Vimeo. https://vimeo.com/28374769, 2015. [Online; accessed 21-September-2015].

[8] NiTE 2.2.0.11 | OpenNI. http://openni.ru/files/nite/, 2015. [Online; accessed 27-August-2015].

[9] openFrameworks. http://www.openframeworks.cc/, 2015. [Online; accessed 19-January-2015].

[10] OpenNI Programmer's Guide. http://com.occipital.openni.s3.amazonaws.com/OpenNI_Programmers_Guide.pdf, 2015. [Online; accessed 27-August-2015].

[11] Pixel - extrains on Vimeo. https://vimeo.com/114767889, 2015. [Online; accessed 21-September-2015].

[12] Processing.org. https://processing.org/, 2015. [Online; accessed 19-January-2015].

[13] What is Live? Learn more about Ableton's music making software | Ableton. https://www.ableton.com/en/live/, 2015. [Online; accessed 29-September-2015].

[14] Reza Ali. ofxUI: A User Interface Addon for OF - REZA ALI. http://www.syedrezaali.com/ofxui/, 2015. [Online; accessed 29-August-2015].

[15] Jordan Backhus. Dancer Bends Light in Stunning Projection-Mapped Performance | The Creators Project. http://thecreatorsproject.vice.com/blog/dancer-bends-light-in-stunning-projection-mapped-performance?utm_source=tcpfbus, 2015. [Online; accessed 21-September-2015].

[16] Helen Bailey, James Hewison, and Martin Turner. Choreographic morphologies: digital visualization and spatio-temporal structure in dance and the implications for performance and documentation. In *Proceedings of the International Events in Visual Arts (EVA Conference), London*, pages 9–18, 2008.

[17] Joey Bargsten. Gesture and Performance with Kinect, Quartz Composer, and PureData. http://fau3711.pbworks.com/w/file/fetch/76805993/Gesture_and_Performance_with_Kinect_Quartz_and_PD.pdf, 2013. [Online; accessed 19-January-2015].

[18] Tamara Berg, Debaleena Chattopadhyay, Margaret Schedel, and Timothy Vallier. Interactive music: Human motion initiated music generation using

skeletal tracking by kinect. In *Proc. Conf. Soc. Electro-Acoustic Music United States*, 2012.

[19] John Charles Butcher. *The numerical analysis of ordinary differential equations: Runge-Kutta and general linear methods*. Wiley-Interscience, 1987.

[20] Edwin Catmull. *How Pixar fosters collective creativity*. Harvard Business School Publishing, 2008.

[21] Jonathan Deutscher, Andrew Blake, and Ian Reid. Articulated body motion capture by annealed particle filtering. In *Proceedings of the IEEE on Computer Vision and Pattern Recognition*, volume 2, pages 126–133. IEEE, 2000.

[22] Benjamin Glover. Real Time Interactive Technology in Dance using Kinect. `http://benglover.net/wp-content/uploads/2014/03/Interactive-Technology-in-Dance-Project-Report-by-Ben-Glover.pdf`, 2013. [Online; accessed 19-January-2015].

[23] Berto Gonzalez, Erin Carroll, and Celine Latulipe. Dance-inspired technology, technology-inspired dance. In *Proceedings of the 7th Nordic Conference on Human-Computer Interaction: Making Sense Through Design*, pages 398–407. ACM, 2012.

[24] Pete Hellicar and Joel Gethin Lewis. Divided By Zero, Hellicar and Lewis Ltd. `http://www.hellicarandlewis.com/divide-by-zero/`, 2015. [Online; accessed 19-January-2015].

[25] Sultanullah Jadoon, Salman Faiz Solehria, and Mubashir Qayum. Optimized selection sort algorithm is faster than insertion sort algorithm: a comparative study. *International Journal of Electrical & Computer Sciences IJECS-IJENS*, 11(02):19–24, 2011.

[26] Doris Jung, Marie Hermo Jensen, Simon Laing, and Jeremy Mayall. . cyclic.: an interactive performance combining dance, graphics, music and kinect-technology. In *Proceedings of the 13th International Conference of the NZ*

*Chapter of the ACM's Special Interest Group on Human-Computer Interaction*, pages 36–43. ACM, 2012.

[27] Kourosh Khoshelham and Sander Oude Elberink. Accuracy and resolution of kinect depth data for indoor mapping applications. *Sensors*, 12(2):1437–1454, 2012.

[28] Gene Kogan. genekogan/ofxSecondWindow - GitHub. `https://github.com/genekogan/ofxSecondWindow`, 2015. [Online; accessed 29-August-2015].

[29] Celine Latulipe and Sybil Huskey. Dance. draw: exquisite interaction. In *Proceedings of the 22nd British HCI Group Annual Conference on People and Computers: Culture, Creativity, Interaction-Volume 2*, pages 47–51. British Computer Society, 2008.

[30] W Scott Meador, Eric M Kurt, and Kevin R O'Neal. Virtual performance and collaboration with improvisational dance. In *ACM SIGGRAPH 2003 Sketches & Applications*, pages 1–1. ACM, 2003.

[31] W Scott Meador, Timothy J Rogers, Kevin O'Neal, Eric Kurt, and Carol Cunningham. Mixing dance realities: collaborative development of live-motion capture in a performing arts environment. *Computers in Entertainment (CIE)*, 2(2):12–12, 2004.

[32] Kyle Orland. Microsoft Announces Windows Kinect SDK For Spring Release. `http://www.gamasutra.com/view/news/33136/Microsoft_Announces_Windows_Kinect_SDK_For_Spring_Release.php`, 2011. [Online; accessed 25-August-2015].

[33] Denis Perevalov. *Mastering openFrameworks: Creative Coding Demystified*. Packt Publishing Ltd, 2013.

[34] Alex Pham. E3: Microsoft shows off gesture control technology for xbox 360. `http://latimesblogs.latimes.com/technology/2009/06/microsofte3.html`, 2009. [Online; accessed 25-August-2015].

[35] Danilo Gasques Rodrigues, Emily Grenader, Fernando da Silva Nos, Marcel de Sena Dall'Agnol, Troels E Hansen, and Nadir Weibel. Motiondraw: a tool for enhancing art and performance using kinect. In *CHI'13 Extended Abstracts on Human Factors in Computing Systems*, pages 1197–1202. ACM, 2013.

[36] Anna Trifonova, Letizia Jaccheri, and Kristin Bergaust. Software engineering issues in interactive installation art. *International Journal of Arts and Technology*, 1(1):43–65, 2008.

[37] R Hevner von Alan, Salvatore T March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, 28(1):75–105, 2004.

[38] Zhengyou Zhang. Microsoft kinect sensor and its effect. *MultiMedia, IEEE*, 19(2):4–10, 2012.