Design and implementation of an interactive graphics generation system

Fabià Serra Arrizabalaga

BACHELOR'S THESIS / 2014-2015

Bachelor's degree in Computer Science
Bachelor's degree in Audiovisual Systems Engineering

THESIS ADVISOR

Dr. Narcís Parés, Department of Information and Communication Technologies

To my family and friends,

Acknowledgements

I wish to thank professor Shlomo Dubnov for giving me the chance to come to University of California, San Diego and collaborate with his group CREL (Center for Research in Entertainment and Learning) and work inside the Calit2 division (California Institute for Telecommunications and Information Technology). It has been a great experience, personally and professionally.

Also acknowledge my advisor Narcís Parés for all his support throughout the course of the project and being the first to introduce me to the openFrameworks toolkit. His patience and all the advices he has given me, thanks to his great knowledge creating interactive systems and project development, has been an immense help in the fulfillment of this project.

I also want to give a big thanks to the openFrameworks creators and its community for their constant enthusiasm, all their help in the forums and for developing this great toolkit. Also to the Open Source Community, being able to read well written code and get inspired by all the different projects.

Finally I also want to thank my parents for their unconditional support during all these years of education, without them nothing of this would have been possible. Also my brother Octavi, his crazy talent and hard working effort has always been a big inspiration to me.

Abstract

In modern dance it is becoming more common to use projected visual imagery to enhance the performance and explore new artistic paths. A challenge is how to generate the images in real-time and integrate them with the movements of the dancer. In this project I have designed and implemented a real-time interactive system with the openFrameworks toolkit that generates computer graphics based on human motions and that can be used in dance performances. Using a Microsoft Kinect depth sensing camera and efficient computer vision algorithms, the system extracts motion data in real-time that is then used as input controls for the graphics generation. The graphics generated include particle systems with different behaviors, abstract representation of the silhouette of the dancers and two dimensional fluids simulation. By means of a user-friendly interface we can adjust the parameters that control the motion capture, change the graphics projected and alter their behavior in real-time. The system has been tested in real dance performances and has fulfilled the requirements that were specified for that user context.

Resum

En dança moderna és cada vegada més comú utilitzar imatges projectades per a millorar la experiència de l'espectacle i explorar nous camins artistics. Un repte és com generar aquestes imatges en temps real i integrar-les als moviments del ballarí. En aquest projecte he dissenyat i implementat un sistema interactiu a temps real utilitzant la llibreria openFrameworks que genera gràfics en ordinador a partir del moviment humà i que pot ser utilitzat en espectacles de dança. Utilitzant el sensor Microsoft Kinect i algoritmes eficients de visió artificial, el sistema extreu informació del moviment en temps real que serveix com a control d'entrada per a la generació dels gràfics. Els gràfics que es poden generar inclouen sistemes de partícules amb diferents comportaments, representació abstracta de la silueta dels ballarins i un simulador de fluids en dos dimensions. A través d'una interfície gràfica es poden ajustar els paràmetres que controlen la captura del moviment, canviar els gràfics que es projecten i alterar el seu comportament en temps real. El sistema ha estat testejat en situacions reals i ha complert els requeriments que s'havien plantejat per aquest context.

Contents

1 INTRODUCTION **1**
- 1.1 Personal Motivation . . . 1
- 1.2 Problem Definition . . . 2
- 1.3 Approach . . . 3
- 1.4 Contributions . . . 3
- 1.5 Outline . . . 4

2 BACKGROUND **5**
- 2.1 Motion capture . . . 5
  - 2.1.1 Mechanical . . . 5
  - 2.1.2 Magnetic . . . 6
  - 2.1.3 Inertial . . . 6
  - 2.1.4 Optical . . . 6
- 2.2 Related applications . . . 9
- 2.3 Software . . . 13
  - 2.3.1 openFrameworks . . . 14
- 2.4 Conclusions . . . 16

3 REAL-TIME MOTION CAPTURE **17**
- 3.1 Introduction . . . 17
- 3.2 Requirements . . . 18
  - 3.2.1 Functional Requirements . . . 19
  - 3.2.2 Non-functional Requirements . . . 19
- 3.3 Image processing . . . 20
- 3.4 Contour Detection . . . 21
- 3.5 Tracking . . . 24
- 3.6 Optical Flow . . . 25
- 3.7 Velocity Mask . . . 27
- 3.8 Conclusions . . . 29

4 GENERATION OF GRAPHICS IN REAL-TIME **31**
4.1 Introduction . . . 31
4.2 Requirements . . . 33
4.2.1 Functional Requirements . . . 33
4.2.2 Non-functional Requirements . . . 34
4.3 Fluid Solver and Simulation . . . 34
4.3.1 Theory . . . 35
4.3.2 Implementation . . . 36
4.4 Particle System Generation . . . 39
4.4.1 Theory . . . 40
4.4.2 Single Particle . . . 41
4.4.3 Particle System . . . 43
4.4.4 Emitter Particles . . . 54
4.4.5 Grid Particles . . . 56
4.4.6 Boids Particles . . . 57
4.4.7 Animations Particles . . . 59
4.4.8 Fluid particles . . . 60
4.5 Silhouette Representation . . . 62
4.6 Conclusions . . . 64

5 *CREA*: SYSTEM PROTOTYPE **65**
5.1 Introduction . . . 65
5.2 Requirements . . . 66
5.2.1 Functional Requirements . . . 67
5.2.2 Non-functional Requirements . . . 67
5.3 Overall System Architecture . . . 68
5.4 Control Interface . . . 69
5.5 Cue List . . . 71
5.6 Gesture Follower . . . 72
5.7 Conclusions . . . 74

6 EVALUATION AND TESTING **77**
6.1 Introduction . . . 77
6.2 User Testing of the Prototype . . . 78
6.3 Discussion . . . 80

7 CONCLUSIONS **83**
7.1 Future work . . . 84

A *CREA* USER'S GUIDE **89**
A.1 Requirements . . . 89

A.2 Installation 89
A.3 MAIN MENU 92
A.4 BASICS + KINECT 92
A.4.1 BASICS 92
A.4.2 KINECT 93
A.5 GESTURES + CUE LIST 95
A.5.1 GESTURE SEQUENCE 95
A.5.2 CUE MAPPING 96
A.5.3 GESTURE FOLLOWER 96
A.5.4 CUE LIST 96
A.6 FLOW + FLUID SOLVER 97
A.6.1 OPTICAL FLOW 97
A.6.2 VELOCITY MASK 98
A.6.3 FLUID SOLVER 98
A.6.4 FLUID CONTOUR 100
A.7 FLUID 2 + CONTOUR 100
A.7.1 FLUID MARKER 100
A.7.2 FLUID PARTICLES 102
A.7.3 SILHOUETTE CONTOUR 103
A.8 PARTICLE EMITTER 103
A.8.1 PARTICLE EMITTER 103
A.9 PARTICLE GRID 106
A.10 PARTICLE BOIDS 109
A.11 PARTICLE ANIMATIONS 111

List of Figures

1.1 *CREA* functionality block diagram 2

2.1 Microsoft Kinect. Source Wikipedia[1] 8

3.1 Block diagram of the motion capture part of the system. 18
3.2 Steps to get the contour from IR Data Stream and Depth Map. . . 21
3.3 Conditions of the border following starting point 22
3.4 Binary image representation with surfaces S_2, S_4 and S_5 the ones we want to extract its boundaries. 22
3.5 Illustration of the process of extracting the contour from a binary image. The circled pixels are the starting points of border following. 23
3.6 Optical Flow texture 27
3.7 Difference in the result of applying a contour detection algorithm to the absolute difference between two consecutive frames and using the velocity mask algorithm. 28

4.1 Block diagram of the graphics generation part of the system. . . . 32
4.2 State fields of the fluid simulation stored in textures. 37
4.3 Silhouette Contour Input textures for the Fluid Solver and density state field texture. 38
4.4 IR Markers Input textures for Fluid Solver and density state field texture. 39
4.5 Attract and Repulse interaction 51
4.6 Emitter particles possibilities 55
4.7 Flocking behavior basic rules[2]. 59
4.8 Diagram of the fluid particles textures 61
4.9 Polygon with a non-convex vertex 62
4.10 Silhouette representation graphic possibilities. 63

5.1 Block diagram of the *CREA* system 66
5.2 Cue List panel in the control interface. 72
5.3 Snapshots of using the Gesture Follower in *CREA* 73

5.4 *CREA* UML Class Diagram with the basic class functionalities. . . 75

6.1 Illustration of the setups used in the evaluations. 78
6.2 Pictures from the evaluations of *CREA*. 81

A.1 Main menu and basics panel 93
A.2 Microsoft Kinect panels 95
A.3 Gestures and Cue List panels 97
A.4 Optical Flow and Velocity Mask panels 98
A.5 Fluid Solver and Fluid Contour panels 101
A.6 Fluid Marker, Fluid Particles and Silhouette Contour panels 104
A.7 Emitter Particles panel 107

Chapter 1

INTRODUCTION

This chapter introduces my personal motivation for the project, the problem I am attempting to solve, how this problem has been addressed, the main contributions that I believe I have made and finally there is an outline of all the chapters in the document.

1.1 Personal Motivation

I am doing a double degree program in Audiovisual Systems Engineering and Computer Science at the Polytechnic School of Universitat Pompeu Fabra in Barcelona.

Since I was born there has been a personal computer in my home and I was really young when I started playing with it; soon I became really interested on computers and technology in general. On the other hand, my mother, who is a painter, taught my brother and me from a very young age that painting and drawing was fun and entertaining.

From that background, I guess it was really easy to be interested in anything that would relate digital technologies with art. I have always liked doing projects in my spare time related to animation, film recording, digital modeling and, more lately, programming projects.

When I started doing this double degree program, which is designed to be done in five years, I already had in mind that I wanted to do a year abroad; I asked the possibility of doing all the required classes in four years and leaving both end of degree projects for the last year, doing only one big project that would be

meaningful for both degrees. As soon as they told me this was okay, I decided to search for universities abroad that did research in areas that I had interest in. USA and specially California was one of the first places in the list.

From all the emails I sent, there was two universities who accepted me and I decided to go to University of California in San Diego. In this university there is a world renowned computer graphics group and the professor with whom I contacted, Shlomo Dubnov, leads a group (Center for Research in Entertainment and Learning, CREL[1]) that combines the research of digital media technologies with arts, so I thought this was a perfect choice.

1.2 Problem Definition

This project is the result of a specific problem that comes from the context of the work of [Dubnov et al., 2014]. Their aim was to create an interactive performance system for floor and Aerial Dance that enhances the artistic expression by means of creating an augmented reality reactive space. With the use of a gesture following algorithm based on a simplified Hidden Markov Model (HMM), they were able to recognize moves from the performer that are similar to gestures that were prerecorded at a training phase, with the idea of using this data to trigger the projected media [Wang and Dubnov, 2015].

The objective of my project has been to explore the real-time generation of graphics using efficient algorithms and to put together a complete standalone system of use in dance performances that integrates these real-time graphics with existing components for the needed motion capture functionality and the rendering of the graphics. The block diagram of the overall system is depicted in Figure 1.1.

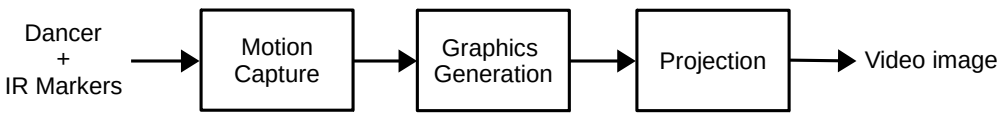


Figure 1.1: *CREA* functionality block diagram

[1] http://crel.calit2.net/about

1.3 Approach

The first step was to study the state of the art in motion capture and graphics generation for live performances to have an idea of the tools used in developing similar systems, both in terms of hardware and software, and decide which algorithms to use in order to get useful data from motion capture and generate effective computer graphics in real-time based on this data.

The next step was to define the list of requirements of the system and from those requirements we conducted an object-oriented design of the software to design the architecture of the system that would meet all the requirements.

Having the system architecture we implemented the application, which we call *CREA*. We tested it in real situations emulating dance performances to evaluate the results obtained and checked wether it met all the initial requirements.

1.4 Contributions

In this project we believe we have made the following contributions.

We have created a standalone system that captures relevant human motion data in real-time and generates graphics also in real-time responding to this data. The system is robust, easy to configure and has an easy way to manage the graphics behavior through the use of a control interface. Moreover, with this interface we can adapt the system to be used in different physical spaces by calibrating the parameters of the motion capture.

We have reviewed relevant background for our project, identifying some creative applications that have been developed in the last few years and that relates to real-time interaction in the context of live performances.

As part of the overall system we have developed one component for motion capture. This component extracts movement data from a performer in a way that it can be reused for other applications beyond the current one. Furthermore, to create more natural looking interactions we use a fluid solver that uses the motion capture to simulate fluid behaviors.

Another core component of the system is the graphics generation. The system integrates various efficient algorithms within the field of particle systems, abstract representation of the silhouette of the dancers and two dimensional fluids simulation. These graphics can be used as a basis to create more complex systems.

Another contribution is that the system has been evaluated in a real context and that we have proved its usefulness in the particular scenario of dance performances.

Finally I want to emphasize that we are contributing to the open source and openFrameworks community by making the software publicly available under an open source license.

1.5 Outline

The rest of this report is organized as follows. Chapter 2 reviews the background of the thesis, describing the state of the art in the field and related applications that have been done. Chapter 3 talks about the motion capture algorithms used for getting motion data from the Microsoft Kinect and that have been integrated in the overall system. Chapter 4 describes the algorithms that have been developed for generating the real-time graphics based on the motion capture data. Chapter 5 presents the complete real-time system that has been developed, *CREA*, describing the design and implementation of both the control interface and the architecture of the system. Chapter 6 explains the evaluation and testing that was carried out to validate the prototype and define the final system. Chapter 7 includes the conclusions, summarizing the contributions and identifying some future work. This document also includes an appendix, which is the user guide of *CREA*, explaining how to install the software and a manual describing the functionalities of the control interface.

Chapter 2

BACKGROUND

This chapter covers some relevant background used to develop this project. It starts talking about different motion capture techniques, then it continues with presenting some similar applications to the one we have developed and it finishes with an overview of softwares that are used in this kind of applications.

2.1 Motion capture

Human motion capture is an increasingly active research area in computer vision since it is used in many different fields. Motion capture describes the process of recording movement and translating that movement onto a digital model.

There are various techniques to do motion capture but some of the most common are mechanical, magnetic, inertial and optical.

2.1.1 Mechanical

Mechanical motion capture is done through the use of an exoskeleton that a performer wears. Each joint of the skeleton has a sensor that whenever the performer moves it records the value of movement. Knowing the relative position of the different sensors, it rebuilds his movements using software. This technique offers high precision and it is not influenced by external factors (such as lights and cameras).

The problem with this technology is that the exoskeletons are sometimes heavy and they use wired connections to transmit data to the computer, limiting the

movements the performer can do. Also the sensors in the joints have to be often calibrated.

2.1.2 Magnetic

Magnetic motion capture uses an array of magnetic receivers placed on the body of the performer that track its location with respect to a static magnetic transmitter that generates low-frequency magnetic field. The advantage of this method is that it can be slightly cheaper than other methods and the data captured is quite accurate. The main drawback is that any metal object or other performers in the scene interfere with the magnetic field and distorts the results. The markers also require repeated readjustment and recalibration.

2.1.3 Inertial

Inertial motion capture uses a set of small inertial sensors that measure acceleration, orientation, angle of incline and other characteristics. With all this data it rebuilds the movements of the performer translating the rotations to a skeleton model in the software. The advantage of this method is that it is not influenced by external factors and that it is very quick and easy to set up. Its disadvantage is that positional data from the sensors is not accurate.

A very good example of this method is MOTIONER[1], an open source inertial motion capture project developed by YCAM InterLab.

2.1.4 Optical

Optical motion capture employs optical and computer vision technologies to detect changes in the images recorded by several synchronized cameras that triangulate the information between them. Data acquisition is traditionally acquired using reflective dots attached to particular locations of the performer's body. However, more recent systems are able to generate accurate motion data by tracking surface features identified dynamically for each particular subject, without the need of reflective dots (markerless motion capture).

The advantage of this method is that the performer feels free to move; there are no cables connecting their body to the equipment, making it really useful for dance

[1] http://interlab.ycam.jp/en/projects/ram/motioner

performances. The data received can be very accurate but it requires a prior well-done calibration between the different cameras. With the development of depth cameras such as Microsoft Kinect, there is no need to use multiple cameras, decreasing the cost of the overall system. The main drawback of this method is that it is prone to light interference and occlusions, causing loss of data.

There exist different kinds of cameras and making the right choice is really important[2] to get the desired final result.

Given the advantages of this motion capture technique, we decided to use this method for this project, using a Microsoft Kinect depth camera to capture the motion; its price, easy to use, decent motion quality and the fact that we wanted to create a tool that could be used by as many people as possible, made the Kinect the first choice. It is an affordable device that is very easy to get and many people already has.

Microsoft Kinect

Microsoft Kinect (see 2.1) is a camera sensor developed by Microsoft announced in 2009 as 'Project Natal' and released in November 2010 as a product for the Xbox 360. It is a combination of hardware and software that allows you to recognize motion by using a sophisticated depth-sensing camera and an infrared scanner.

It is conformed by four basic hardware components:

1. **Color VGA video camera:** Video camera that detects the red, green and blue color components with a resolution of 640x480 pixels and a frame rate of 30 fps.
2. **Infrared emitter:** Projects a speckle pattern of invisible near-infrared light.
3. **Depth sensor:** Sensor that provides us with a depth map that has information about the 3D scene the camera sees. It does that by analyzing the speckle pattern of the infrared emitter (the sensor is an infrared sensitive camera), measuring the time after each dot of the pattern reflects off the objects. The depth range is from about 0.5 meters to 6 meters, although it can go up to 9 meters with less resolution and more noise.
4. **Multi-array microphone:** Array of four microphones to isolate the voices from other background noises and provide with voice recognition capabilities.

[2] http://www.creativeapplications.net/tutorials/guide-to-camera-types-for-interactive-installations/

Figure 2.1: Microsoft Kinect. Source Wikipedia[3]

Through the use of complex algorithms based on machine learning the Kinect can infer the body position of the users present in the field of view and create a skeleton model following the user movements, although the details of this technique are not yet publicly available.

In the date of the release of the Kinect, there were not any drivers available to make use of it in computer applications. However, just a few days after it was released, a group of hackers started the OpenKinect project[4] to create an open source free library that would enable the Kinect to be used with any operating system. This library, called *libfreenect*, gives you access to the basic Kinect device's features: depth stream, IR stream, color (RGB) stream, motor control, LED control and accelerometer.

About a month later, on December 9th, PrimeSense (manufacturer of the PrimeSensor camera reference design used by Microsoft to create the Kinect) acknowledging the interest and achievements of the open source community, decided to open source its own driver and framework API (*OpenNI*), and to release binaries for their NITE skeletal tracking module. This external module is not open source and the license doesn't permit its commercial use.

One year later, Microsoft released the official SDK, which only works on Windows, providing with all the advanced processing features like scene segmentation, skeleton tracking, hand detection and tracking, etc.

In this project, in order to support the open source community, we decided to use *libfreenect*, although at some point we considered the option of using *OpenNI*. This was in order to get the skeletal tracking feature that *libfreenect* doesn't provide. However, this option was discarded right after considering the effect this feature would have in a dance performance. Having worked with the skeletal tracking in previous projects, I already had experience with it and its limitations. In order to recognize the skeleton of the user this has to stand in front of the Kinect sensor facing it; in many cases the performers will come to the stage by the sides of the interactive space and we need to get motion information also on

[3] http://commons.wikimedia.org/wiki/File%3AXbox-360-Kinect-Standalone.png
[4] http://openkinect.org/wiki/Main_Page

these situations. On top of that, the tracking is not reliable with fast movements and non natural human poses; since dance movements can be really fast and contain lots of non natural human poses, the skeleton data would have lots of noise or it wouldn't give any data at all, creating very inconsistent results with the graphics interaction.

As a result, we decided to use only the depth map and the infrared data stream features because they provide with data of the interactive area during all the time, being able to balance the amount of motion data we want at every frame by applying computer vision algorithms.

From the depth map we can get the silhouette of the performer and from the infrared data stream, using some infrared light reflecting tapes we can get precise information of the position of these. The depth map is a grayscale image on which every color of gray indicates a different depth distance from the sensor. Closer surfaces are lighter; further surfaces are darker. This makes it easy to separate the objects/performers in the scene from the background and get just the silhouette data. The infrared data stream gives us the image of the infrared light pattern that the Kinect emits, having the great advantage of being able to capture images even in low light conditions. Infrared light reflecting tape makes the infrared pattern dots bounce the tape off; the infrared sensing camera sees that as if the tape was emitting infrared light and hence it appears as a bright area in the whole surface of the tape. Thanks to this property we can segment the image to get only the bright areas. We call these tapes *IR Markers* and we can either place them on the dancer's body (e.g., wrists and ankles) or in objects.

In [Andersen et al., 2012] there's a very detailed description of the technical functionality of the Kinect and [Borenstein, 2012] provides with great interactive application examples and tricks in developing with the Kinect camera.

2.2 Related applications

The use of media technology is not a new thing to the stage. Its use has been growing exponentially over the past three decades and we are now in a time where the boundary between technology and arts are beginning to blur.

Ever since the motion picture recording of Loïe Fuller's Serpentine Dance in 1895, dance and media technology developed an intimate relationship. In [Boucher, 2011] there is a precise description of how the digital technologies have been gaining ground among dance and related arts, describing various ways in which virtual

dance can be understood, its relation with motion-capture, virtual reality, computer animation and interactivity.

Technology has changed the way we understand and make art, providing artists with new tools for expression. Right now, technology is a fundamental force in the development and evolution of art. The new forms of art allow more human interaction than ever before and they start to question the distinction between physical and virtual, looking outside of what's perceived as "traditional".

The company of Adrien Mondot and Claire Bardainne have been creating numerous projects[5] since 2004 with technology performing a big role on them. They have developed their own software, eMotion[6], to create interactive animations for live stage performances[7]. The application is based on the creation of interactive motions of objects (e.g., particles, images, videos, text) that follow real world physics laws, making the animations very natural looking. External controllers such as Wiimote, OSC captors, sudden motion sensors and midi inputs can be used to define forces and rules that interact with the objects. The documentation is very basic and it is quite hard to learn how to use without enough time exploring the interface. It is closed software for non-commercial use only and there is no way to check the algorithms used for the different simulations.

One of their latest performances, Pixel[8], is a great example of exploiting the use of technology in a dance context. The use of two axes projection (background and floor) creates an illusion space that makes the audience feel totally immersed on the performance, forgetting the border between real and virtual. The graphics generated have a minimalist design that uses very simple shapes (lines, dots and particles) to create a subtle balance between the virtual projected graphics and the dance so that one doesn't overshadow the other.

During the performances, they never put sensors in the dancers[9] but control the digital materials with their hands – using iPads, Leap Motion and Wacom tablets. That plus the use of already recorded videos projected on the stage, makes the system very robust to any kind of artifacts or non desired effects you might get using live motion sensors.

One great example of using live motion sensors is Eclipse/Blue[10], a performance directed by Daito Manabe on which using a high-speed infrared camera they are

[5] http://www.am-cb.net/projets/
[6] http://www.am-cb.net/emotion/
[7] https://vimeo.com/3528787
[8] http://www.am-cb.net/projets/pixel-cie-kafig/
[9] http://thecreatorsproject.vice.com/blog/watch-dancers-wander-through-a-digital-dream-world
[10] http://www.daito.ws/en/work/nosaj_thing_eclipse_blue.html

able to track the dancers movements and create a dynamic virtual environment. In this project they also make use of very simple graphics so the audience doesn't get distracted with them and they only help in making the dancers movements feel larger and more emotive.

This last example was inspired by the works of Klaus Obermaier[11], a media-artist, director, choreographer and composer who has been creating innovative works within the field of performing arts and new media since more than two decades. He is regarded as the uncrowned king of the use of digital technologies to create interactive dance performances. One of his most famous works is Apparition[12], a captivating dance and media performance that uses a very similar technology as Eclipse/Blue to create an immersive kinetic space.

On the opposite side of these interactive projects, there is the use of media technologies to create a sequence of graphics specifically for a particular choreography. The performers meticulously rehearse their movements in order to be synchronized with the art work so it seems like if it was an interactive show. The graphics have to be continually adapted to get the perfect timing and match the projection with the dancers. This is the technique most singers and live events use for their stage visuals, like Beyoncé on the half time Super Bowl show[13]. Another example using this technique is the Japanese troupe Enra[14], who create spectacular live performances that are so precisely rehearsed that makes it difficult to believe is non-interactive. Pleiades[15] is one of their most inspiring works and they use very similar graphics to the ones of Eclipse/Blue and Pixel.

This technique has the advantage that the visual graphics can be as complicate and stunning as we please with the certitude that there won't be any unexpected behaviors. Nonetheless, it has the great disadvantage that it needs lots of time to rehearse the movements and that the minimum synchronization failure is reflected (see minute 1:50 of Beyoncé's video[13]).

All of the projects mentioned so far are closed software and there is almost non information about the algorithms and techniques used.

Reactor for Awareness in Motion[16] is a research open source project, created by YCAM InterLab, that aims to develop tools for dance creation and education. They have created a C++ creative coding toolkit to create virtual environments

[11] http://www.exile.at/
[12] https://www.youtube.com/watch?v=-wVq41Bi2yE
[13] https://www.youtube.com/watch?v=qp2cBXvuDf8
[14] http://enra.jp/
[15] https://www.youtube.com/watch?v=0813gcZ1Uw8
[16] http://interlab.ycam.jp/en/projects/ram

for dancers (RAM Dance Toolkit). The toolkit contains a GUI and functions to access, recognize, and process motion data to support creation of various environmental conditions and gives realtime feedbacks to dancers using code in an easy way. The toolkit uses openFrameworks (see 2.3.1), which means users can use functions from both RAM Dance Toolkit and openFrameworks. As mentioned in section 2.1.3, they created their own inertial motion capture system named MOTIONER, but the system also works with optical motion capture systems such as the Kinect depth camera.

Many projects from the open-source community provide with very inspiring examples in graphics generation and help in learning best practices in programming them effectively. The Processing community (see section 2.3) is one of the most active among them and people share their projects in an online platform named openProcessing[17]. The openFrameworks community is also very active and there are many open source examples in creating interactive applications[18,19,20].

ZuZor is the project of [Dubnov et al., 2014] mentioned before in section 1.2. In this project they use the infrared data stream and the depth map from the Kinect to get the three dimensional position of some infrared reflecting tape (i.e., markers) they put on dancers. Moreover, they also use the depth map to extract the silhouette of the dancer. These functionalities are implemented using Max/MSP/Jitter (KVL Kinect Tracker[21]) and they use the Open Sound Control (OSC) protocol to communicate with Quartz Composer (see section 2.3) to generate some very basic graphics that respond to the data from the Kinect[22]. They only use the three dimensional position of the markers as the input data (not velocity) and the silhouette data is only used to draw it as it is, like a casted shadow. Because of this reason, the graphics don't seem to respond to the motion of the dancer (i.e., quick movements have the same outcome as slow movements).

Even though this project has been developed within the same group that created ZuZor, we have developed the system from scratch without using any code from that project.

[17] http://www.openprocessing.org/

[18] https://github.com/openframeworks/openFrameworks/wiki/Tutorials,
-Examples-and-Documentation

[19] https://github.com/kylemcdonald/openFrameworksDemos

[20] https://github.com/ofZach/algo2012

[21] https://cycling74.com/forums/topic/sharing-kinect-depth-sensitive-blob-tracker-3d-bg-subtraction-etc/

[22] https://www.youtube.com/watch?v=2vg_DBDXrvQ

2.3 Software

There are quite a few different creative programming environments that let you create real-time interactive and audiovisual performances.

One of the most used worldwide for professional performances is Isadora[23]; a proprietary interactive media presentation tool for Mac OS X and Microsoft Windows that allows you to create real-time interactive performances in a really simple and easy way, requiring no programming knowledge at all. The license for the software costs around 300$.

EyesWeb[24] is a Windows based environment designed to assist in the development of interactive digital multimedia applications. It supports a wide number of input devices (e.g., motion capture systems, cameras, audio) from which it extracts various information. The power of EyesWeb is its computer vision processing tools; a body emotional expression analyzer extracts a collection of expressive features from human full-body movement that can be later used to generate different outputs. It has an intuitive visual programming language that allows even non-technical users to develop with it. It is free to download but is not open source.

EyeCon[25] is another tool to facilitate the creation of interactive performances and installations by using a video feed from the performance area. We can draw lines, fields and other elements on top of the video image and when the system detects some movement over the elements drawn, an event message is triggered. These events can be assigned to output different media or send a message to an external application through a communication protocol. Other features of this software include tracking of the objects in the video image and access to information about their shape. It only works in Windows computers and the license for a normal user is 300$.

Other more general tools for easily developing interactive audiovisual applications are Quartz Composer (Mac OS X), VVVV (Windows), Max/MSP/Jitter (Windows and OS X), Pure Data (Multi Platform), Processing (Multi Platform), Cinder (Windows and OS X) and openFrameworks (Multi Platform).

I decided to use openFrameworks because I already had experience with it and I wanted an open-source cross-platform toolkit with text-based programming (Quartz Composer, VVVV, Max/MSP/Jitter and Pure data are node-based visual programming). However, choosing one toolkit doesn't mean you must stick with it, all

[23] http://troikatronix.com/isadora/about/
[24] http://www.infomus.org/eyesweb_ita.php
[25] http://www.frieder-weiss.de/eyecon/

of them can be used interchangeably and use one or another according to its strengths.

2.3.1 openFrameworks

OpenFrameworks[26] is an open source toolkit designed to provide a simple framework for creative and artistic expression, sometimes referred as "creative coding". Released by Zachary Lieberman in 2005 and actively developed by Theodore Watson, Arturo Castro and Zachary itself, it is today one of the main creative coding platforms.

> The main purpose of openFrameworks is to provide users with an easy access to multimedia, computer vision, networking and other capabilities in C++ by gluing many open libraries into one package. Namely, it acts as a wrapper for libraries such as OpenGL, FreeImage, and OpenCV [Perevalov, 2013].

Really good resources to learn openFrameworks and get ideas are [Perevalov, 2013], [Noble, 2012] and [openFrameworks community, 2015]. The last reference is a work in progress book from the openFrameworks community but it is already a good resource to learn the basics and some advanced concepts.

Even though the core of openFrameworks has a bunch of powerful capabilities, it does not contain everything. That is why the community has been creating different external libraries, called 'addons', that add all of these extra capabilities.

The addons used in this project are the following:

ofxKinect

Created by Theodore Watson, ofxKinect[27] is a wrapper for the *libfreenect* library[28] to work with the Microsoft Kinect camera inside openFrameworks projects.

[26] http://openframeworks.cc/
[27] https://github.com/ofTheo/ofxKinect
[28] https://github.com/OpenKinect/libfreenect

ofxCv

Created by Kyle McDonald, ofxCv[29] is an alternative approach to wrap the OpenCV library, being the other alternative the wrapper that comes with the openFrameworks core, ofxOpenCV.

OpenCV[30] is an open source library that contains hundreds of algorithms for image processing and analysis, including object detection, tracking and recognition, image enhancement and correction.

ofxFlowTools

Created by Mathias Oostrik, ofxFlowTools[31] is an addon that combines fluid simulation, optical flow and particles using GLSL shaders. This allows to create realistic fluids from a live camera input in a very computationally effective way, taking advantage of the GPU parallelism.

ofxUI

Created by Reza Ali, ofxUI[32] easily allows for the creation of minimally designed and user-friendly GUIs.

ofxXmlSettings

This addon is now part of the core of openFrameworks and it comes inside the addons folder when we download openFrameworks. It allows for reading and writing xml files.

ofxSecondWindow

Created by Gene Kogan, ofxSecondWindow[33] allows to create and draw to multiple separate windows, being able to have one window to draw the graphics and another for the Graphical User Interface.

[29] https://github.com/kylemcdonald/ofxCv
[30] http://opencv.org/
[31] https://github.com/moostrik/ofxFlowTools
[32] https://github.com/rezaali/ofxUI
[33] https://github.com/genekogan/ofxSecondWindow

2.4 Conclusions

In this chapter we presented some relevant background that we used to develop this project. We haven't given much specific background about graphics generation but given the importance of this topic we give more details in chapter 4.

Chapter 3

REAL-TIME MOTION CAPTURE

This chapter explains the algorithms used for getting the motion data from the Microsoft Kinect depth sensor. First it introduces the problem of motion capture and the list of functional and non-functional requirements that were considered from this part of the system. Next it describes briefly the maths behind the algorithms and how they are used inside *CREA*. It ends up with a short conclusion.

3.1 Introduction

Extracting precise information from gesture data is crucial if we want the graphics to dialogue with the dancer and interact with him in a natural way; making the graphics participant of the dance and not a mere decoration. More than the beauty and complexity of the graphics, the most important thing is that the graphics use the motion data correctly in a way that we can establish an adequate dialogue between the graphics and the dancers movements.

As explained already in Chapter 2, we decided to use only the depth map and the infrared data stream features from the Kinect.

Figure 3.1 summarizes the functionality of this part of the system and the sections explained in this chapter. The depth map and the infrared data stream images are both processed to obtain a binary image with only the relevant surfaces we want to track. These images are then fed to a contour detection algorithm that results in a set of points defining the binary surfaces. In the case of the depth map this data is then used to estimate the optical flow and to obtain the areas of motion

between two consecutive frames (Velocity Mask). In the case of the infrared data stream the *IR Markers* positions are fed to a tracking algorithm in order to get their velocity.

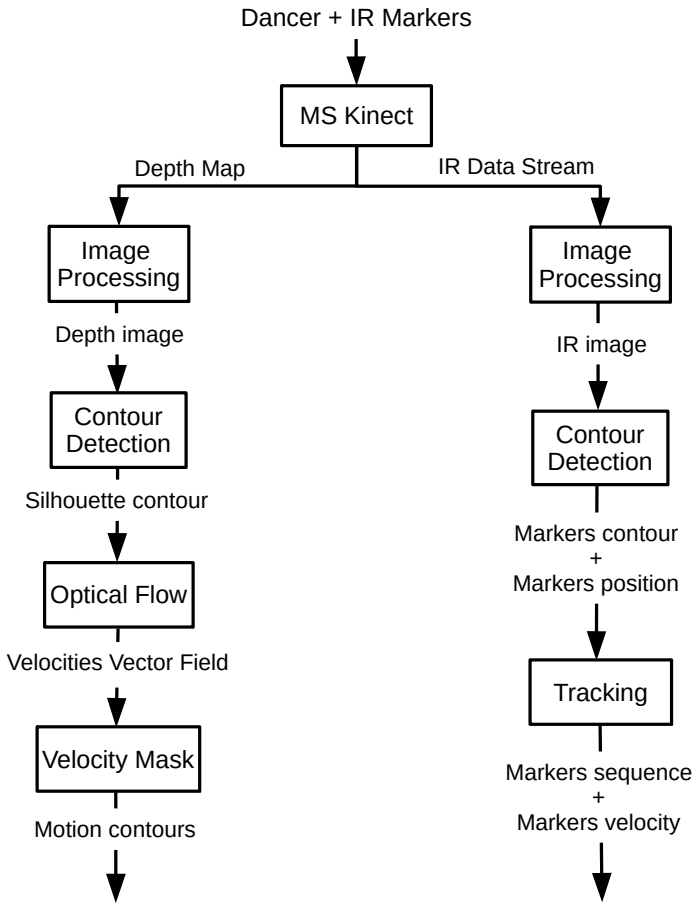


Figure 3.1: Block diagram of the motion capture part of the system.

3.2 Requirements

Following what we have mentioned in the introduction with respect to the features we decided to use from the Microsoft Kinect, we created the list of functionalities that the motion capture system would have to accomplish and the constrains in order to satisfy the proper functioning of the overall system in the proposed context. First we explain the functional requirements and then we continue with the non-functional ones.

3.2.1 Functional Requirements

The functional requirements are described as specific functions or behaviors that the system must accomplish. The list of functional requirements that we defined for the motion capture system is the following:

1. **Background subtraction**
   The motion capture system must extract the regions of interest (performers) from the background for further processing.

2. **Motion of the performer**
   The motion capture system must obtain information about the motion of the performer, both quantitively (how much it moves) and spatially (where it moves).

3. **Information about the *IR Markers***
   The motion capture system must get the position at all times of the infrared reflecting tapes used as markers. It also needs to get the velocity of the *IR Markers* by following its position over time.

3.2.2 Non-functional Requirements

The non-functional requirements specify criteria that can be used to judge the operation of the system, rather than specific functions they describe how the system should work. The list of non-functional requirements we defined for the motion capture system is the following:

1. **Real-time**
   The motion capture system shall guarantee response within 20 milliseconds from the actual motion.

2. **Fault Tolerance**
   The motion capture system shall be tolerant to the inherent noise of the Microsoft Kinect data.

3. **Crash recovery**
   The motion capture system shall be able to restart in case of an unexpected error.

4. **Portability**
   The motion capture system shall be able to adapt to different physical environments, with different lighting conditions and setups.

5. **Usability**
   The motion capture system shall be easy to use from the other modules of the system.

3.3 Image processing

Through the use of image processing techniques we can extract the relevant surfaces from the depth map and the IR data stream and reduce its noise so we get cleaner data. The first step is to segment the images to separate the relevant data from the non-relevant information (i.e., background). The simplest method of image segmentation is called thresholding, which given a grayscale image and a threshold value it results in a binary image (i.e., mask). Basically it replaces with a black pixel each of the pixels in the image that its intensity is below the threshold and with a white pixel if its intensity is greater. Our goal is to get the areas of interest in white and the rest in black and the approach is a bit different for both inputs.

In the depth map we apply two thresholds (near threshold and far threshold) to isolate the objects that are in a specific range, say only the objects between one meter and three meters; sometimes we will want to discard the objects closer to a certain distance. In order to do that we multiply the two binary images (i.e., bitwise AND operation) resulting from the two thresholding operations, getting a final mask with only the pixels that are common in both. However, for the near threshold if we apply a normal thresholding we will get the closer surfaces white and the rest in black; we need to get the exact opposite so the closer surfaces are black and when we multiply both images these get discarded. The solution is to invert the depth map before applying the near threshold (see Figure 3.2).

On the other hand, for the IR data stream we just apply one threshold to separate the bright areas (i.e., *IR Markers*) from the rest, since in this image there is not a gradient of intensities defining the distance from the sensor; higher intensity just depends on how much infrared light the marker reflects.

Having both binary images (*IR Mask* and *Contour Mask*), we apply the two most common morphology operators: Erosion and Dilation. First an erosion to delete small white dots that might result from the inherent noise of the Kinect and then a dilation to fill holes. After that we blur the image with a Gaussian blur to smooth the surface and reduce noise.

As a last step we crop the binary images to cut undesired parts from the interaction space. Sometimes we will have a small space and there will be objects (e.g., walls,

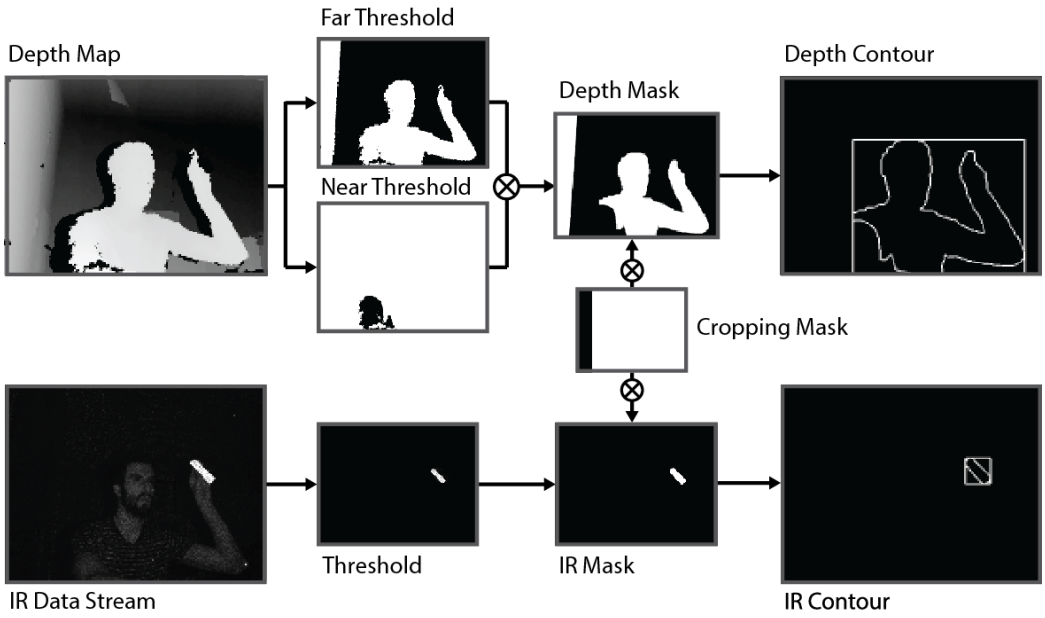


Figure 3.2: Steps to get the contour from IR Data Stream and Depth Map.

floor, ceiling, furniture) that we do not want the system to track or bright lights in the scene emitting infrared light that disturb the depth sensor and creates noise. These undesired areas can be blacken out so they do not affect the interaction by cropping the image, although the interaction space will be smaller. The process for cropping is really simple: (1) create a mask of the same size as the image we want to crop (640×480 pixels), (2) assign 0's to the pixels we want to crop and 1's to the rest and (3) multiply the image by this mask.

All these image processing operations are done using the OpenCV library through the *ofxCv* addon and they are summarized in Figure 3.2.

3.4 Contour Detection

Having the two binary images from the previous step (*IR Mask* and *Contour Mask*), we apply a contour detection algorithm to get the set of points defining the boundary of all the white pixels in the image, namely the silhouette and *IR Markers* boundaries. This technique is known as border following or contour tracing and the algorithm OpenCV uses is [Suzuki and Abe, 1985].

Given a binary image, the algorithm applies a raster scan and it interrupts the scan whenever a pixel (i, j) satisfies one of the conditions in Figure 3.3. When this

is true it means that in that pixel there is a beginning of a border, either an outer border or a hole border (see Figure 3.4):

(a) Outer Border **(b)** Hole Border

Figure 3.3: Conditions of pixel (i, j) to consider is a border following starting point.

From this pixel the algorithm follows the border and assigns a uniquely identifiable number to all the pixels conforming the border; this identification number is set to negative if the pixel satisfies the second condition (Figure 3.3b). After having followed the entire border it saves the border information (if it is a hole or an outer border) and it resumes the raster scan. At every new border it encounters it compares it with the previous border and depending on wether this is a hole or an outer border and if the previous was a hole or an outer border it creates the structure of objects (as shown in the right hand side of Figure 3.5). When the raster scan reaches the lower right corner of the picture the algorithm stops, having the boundaries of the objects defined by the sequential numbers assigned to the borders.

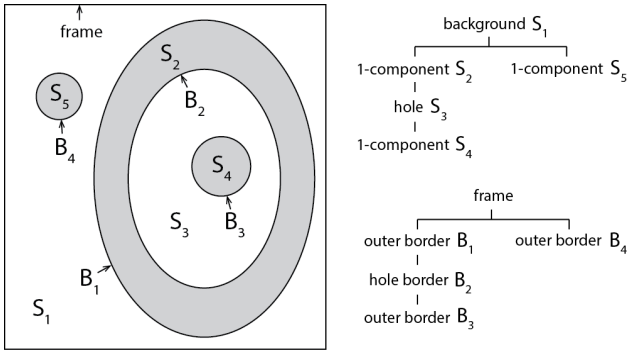


Figure 3.4: Binary image representation with surfaces S_2, S_4 and S_5 the ones we want to extract its boundaries.

Figure 3.5 illustrates an example of applying the algorithm to a simple binary image. These are the steps:

1. When the raster scan reaches the circled pixel in *(a)* it finds it satisfies the condition of figure 3.3a, meaning this is the starting point of an outer border.

2. The sequential number 2 is assigned to the whole border and the pixels that satisfy condition of figure 3.3b are set to -2.

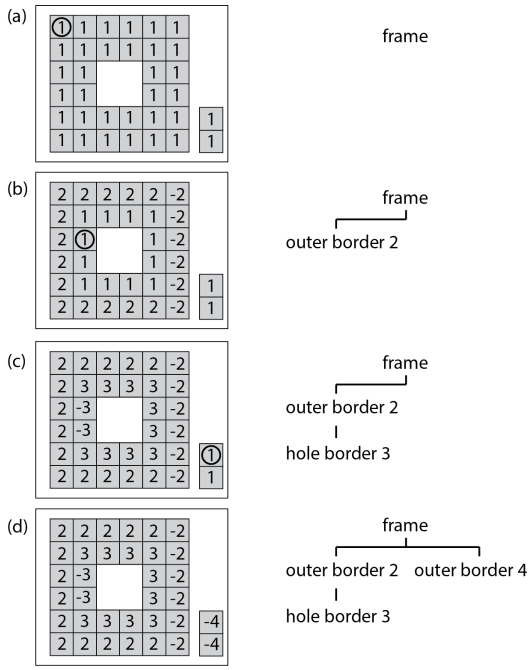


Figure 3.5: Illustration of the process of extracting the contour from a binary image. The circled pixels are the starting points of border following.

3. By comparing this border with the latest encountered border (i.e., frame) we find out that its parent border is *frame*.

4. The raster scan is resumed and it gets interrupted by the circled pixel in *(b)*, which satisfies condition of a hole border (figure 3.3b).

5. The border is followed and we assign the sequential number 3 or -3 in the same manner as before.

6. Since this new border is a hole border and the previous encountered border was an outer border, we find out that this is his parent border.

7. Resuming the raster scan this does not get interrupted until the circled pixel in *(c)*, which we find out is an outer border of a different object.

8. The algorithm ends when it scans the bottom-right pixel.

Applying this algorithm to the binary images we get the set of points conforming the silhouette of the user (i.e., *silhouette contour*) and the area of the *IR Markers* (see Figure 3.2). By getting the centroid of the set of points conforming the *IR Markers* (average position of all the points) we get a unique point describing the position of the marker, which we later use to track its position over time.

3.5 Tracking

In order to track the position of the markers over time we apply a simple tracking algorithm that comes with the *ofxCv* addon. Having a list of the positions of the markers in the actual frame and a list of the positions in the previous frame, the algorithm finds the best match by comparing all the distances between them. To identify each of the markers in the list it assigns a uniquely identifying number as a label. If there are more markers in the actual frame than in the previous, it creates new labels for them. On the other hand, if there are less markers in the actual frame than in the previous the algorithm assigns the marker labels that result in the best match and the ones that remain unmatched start a counter of the number of frames that have been unmatched and are stored in a special vector of 'dead' labels. In the following frames the actual markers positions are also compared with these unmatched labels positions and if they are closer than the other options, they get assigned to the marker; the counter is put to zero again and the label is deleted from the 'dead' vector. If the counter surpasses a specified value ('persistence'), the label is deleted from the list of possible marker labels and is put in the vector of 'dead' labels forever.

One important value of this tracking algorithm is 'maximumDistance', which determines how far a marker can move until the tracker considers it a new one. This is done by filtering the different distances between the markers at every frame, eliminating those that are higher than 'maximumDistance'.

In terms of the implementation we create an $N \times M$ matrix, where N is the number of markers in the actual frame and M the number of markers in the previous frame, containing the distances (filtered with 'maximumDistance') between all the markers. Each row of the matrix is then sorted by distance and iterating through all the rows we obtain the best match for each of the markers. For all the unmatched markers from the actual frame (rows not assigned to any column) we assign a new identifying label. In contrast, the unmatched markers from the previous frame (columns not assigned to any row) are put to the 'dead' vector and depending if they are young (counter is less than 'persistence') we just increment their counter or they are deleted from the list of previous markers (its corresponding column in the matrix is deleted). As a last step, the position of the marker is interpolated with its previous position so we get a smoother movement between frames. Having the sequence of positions of the markers over time, it is really simple to get their velocity: subtract the position from the previous frame to the actual.

This tracking algorithm is not very robust and if the paths of the *IR Markers* get crossed quickly its labels might swap. However, we can tolerate this error for the purpose of our application.

3.6 Optical Flow

Optical flow is defined as the pattern of apparent motion of objects by the relative motion between an observer (in this case the Kinect sensor) and the scene. There are a many different methods to estimate this motion but the basic idea is to given a set of points in an image/frame find those same points in another image/frame. At the end we get a vector field describing the motion of all the single points from the first frame to the second. This approach fits perfectly in order to estimate the motion from the *silhouette contours* over time.

Following this idea we first implemented a very straightforward method using the *silhouette contour*; for each point of the silhouette in the actual frame we compute the distance with every other point in the previous frame and we take the one that is closest. By subtracting both points we estimate the velocity vector in that position. We end up with a set of vectors around the *silhouette contour* describing the change of position from one frame to the other. This approach is implemented in *CREA* and it works but it is not being used; the estimation of the optical flow fulfills the same requirements with better results.

The methods to estimate the optical flow are normally classified in two different types: dense techniques and sparse techniques. The former gives the flow estimation for all the pixels in the image while the latter only tracks a few pixels from specific features (e.g., object edges, corners and other structures well localized in the image). Dense techniques are slower and are not good for large displacements but they can give more accuracy in the overall motion. For the purpose of *CREA* we decided it was better to use a dense method since we wanted to get flow information on the whole interaction area, which sparse techniques do not provide since it only tracks specific features.

Having said that, we decided to use the optical flow dense estimation method provided by OpenCV, Gunner Farneback's Optical Flow [Farnebäck, 2003]. The problem with this approach was that the latest stable version of openFrameworks (0.8.4), which is the one used for this project, comes with an older OpenCV version (2.3.1) that does not support GPU computational capabilities, having to estimate the optical flow entirely using the CPU. As a result, in order to estimate the optical flow in real-time we had to scale down the depth image and compute the flow on this lower resolution image. With this approach we were getting a decent framerate and the interaction with the graphics generated was very good for a few hundred particles.

Once after we started adding graphics more computationally expensive to the system (e.g., fluid solver in section 4.3) we realized that the computation of the opti-

cal flow was consuming most of the CPU, making the application work very slow when running everything together. As a solution, we started looking for optical flow implementations based on the GPU. One of the options was to built the latest OpenCV version (which does already include optical flow GPU support), link it with openFrameworks and make direct calls to the library. However, we found out that the *ofxFlowTools* addon mentioned in Chapter 2 already implemented an optical flow working in the GPU, making it easier to install for other users that would like to extend *CREA* and a cleaner code than the other approach.

This implementation uses a gradient-based method, which also provides a dense optical flow. This method assumes that the brightness of any point in the image remains constant (or changes very slowly) to estimate the image flow at every position in the image. Since we are using the binary depth image as input to the optical flow and this has constant brightness throughout all the frames, this fits very well. This assumption of constant brightness can be written mathematically as:

$$I(\vec{x}, t) = I(\vec{x} + \vec{u}, t + 1) \tag{3.1}$$

Where $I(\vec{x}, t)$ is the intensity of the image as a function of space $\vec{x}$ and time t and $\vec{u}$ is the flow velocity we want to estimate.

By means of a Taylor series about $\vec{x}$ we can expand equation 3.1 and ignoring all the terms higher than first order we get this equation:

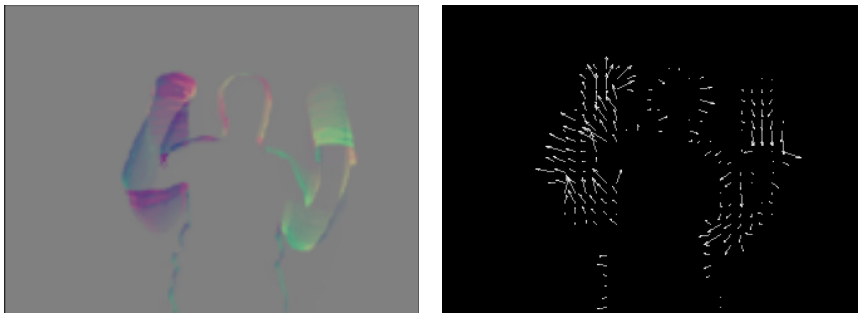
$$\nabla I(\vec{x}, t) \cdot \vec{u} + I_t(\vec{x}, t) = 0 \tag{3.2}$$

Where $\nabla I(\vec{x}, t)$ and $I_t(\vec{x}, t)$ are the spatial gradient and the partial derivative with respect to t of the image I, and $\vec{u}$ is the two dimensional optical flow vector.

Due to its nature, whenever we cannot estimate $I_t(\vec{x}, t)$ we can replace it by computing the difference of I in two consecutive frames:

$$I_t(\vec{x}, t) \equiv I(\vec{x}, t + 1) - I(\vec{x}, t) \tag{3.3}$$

Given this, the implementation of a gradient-based optical flow is quite straightforward. Since the computation is identical for every point in the space we can take advantage of the parallelism of the GPU to estimate the velocity simultaneously for every pixel. The implementation we use does that by calling a fragment shader written in GLSL (OpenGL Shading Language) with the texture of the actual frame ($I(\vec{x}, t)$) and the one in the previous ($I(\vec{x}, t - 1)$) as arguments. By



(a) Optical Flow texture **(b)** Optical Flow velocity vectors

Figure 3.6: Optical Flow vector field texture with its corresponding velocity vectors representation.

solving equation 3.2 it estimates the flow velocity for every pixel and stores this information in another texture (the magnitude in 'x' is stored in the red channel and the magnitude in 'y' in the green channel, see Figure 3.6).

On top of that, this implementation includes another shader to blur the velocity field temporally and spatially. To blur temporally it aggregates the flow textures at every frame in one same buffer, slowly erasing the buffer by drawing a rectangle of the same size of the textures with a certain opacity. If this rectangle is drawn with maximum opacity the buffer gets deleted at every frame and there is not any temporal blur (the buffer only contains the actual flow texture). The opacity of this rectangle is directly proportional to a 'decay' value; if 'decay' is high the opacity is high too and there is no temporal blur. In order to blur spatially, it applies a weighted average on each vector of the flow field texture with its neighbors, being able to set the amount of blur by setting how many neighbors to use to compute the average ('radius').

Although this algorithm runs very fast, since there is no need to have much precision on the flow field for our purposes, we decided to use the same approach as we had done previously for the optical flow based on the CPU. The optical flow is estimated using a lower resolution of the depth image (image scaled down by four). This makes the algorithm run faster and saves resources for computing more complex graphics.

3.7 Velocity Mask

The velocity mask is an image that contains the areas where there has been motion by using the optical flow field. Given an input image it returns this same

image with only the pixels which position in the flow field correspond to a high velocity.

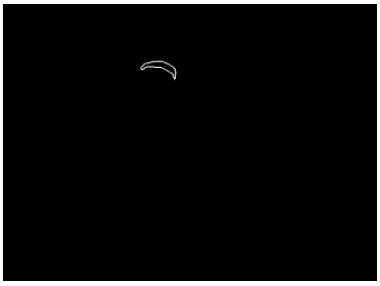
The approach is quite simple and it is implemented using a fragment shader executed on the GPU, which means that for every pixel in the screen it does an independent operation. Using the optical flow texture (see Figure 3.6a) what we do is set a higher opacity in the input image pixel as higher the velocity is in that same pixel position in the flow texture. Basically we take the pixel value in the input image, take the same pixel in the flow texture (i.e., velocity) and set the magnitude of this second pixel (i.e., magnitude of the velocity) as the opacity of the first. This results in an image revealing the original pixels of the input image (higher opacity than the rest) where there is motion.

This algorithm is really similar in concept as getting the absolute difference of two consecutive frames, which also results in the areas where there has been a change highlighted. However, with the velocity mask we have more control over the motion areas by controlling the optical flow parameters and the result is more precise than just taking the frame difference (see Figure 3.7).

(a) Absolute Difference

(b) Absolute Difference Contour Detection

(c) Velocity Mask



(d) Velocity Mask Contour Detection

Figure 3.7: Difference in the result of applying a contour detection algorithm to the absolute difference between two consecutive frames and using the velocity mask algorithm.

Having these areas of motion highlighted we can then apply the contour detection algorithm explained in section 3.4 to get the set of contour points defining these regions (see Figure 3.7); we call them *motion contours*.

3.8 Conclusions

In this chapter we have described our work on motion capture, covering all the algorithms that have been used in the *CREA* system.

The algorithms used for motion capture work very well in most of the environments and the latency is low enough for the purpose of this project. Moreover, the output data we obtain is quite precise and describes pretty accurately the motion of the performers, offering a great variety of possibilities for generating graphics that respond to this data. Having said that, we consider we have fulfilled satisfactorily the proposed requirements from the beginning of the chapter.

The main problem of the motion capture is due to the inherent technology Microsoft Kinect uses for getting the depth map. The depth map has quite a lot of noise/jitter and makes the algorithms that are based on this data slightly less robust. Sometimes this problem will be caused by infrared emitting lights in the visual field that interfere with the depth sensor. One simple way to reduce this problem is by using infrared filters/glass materials that filters/reflects outside the infrared lights and lowers the interference with the depth sensor. [Camplani et al., 2013] proposes a more complex solution to this problem. By using spatial and temporal information from the depth map it obtains the missing values in the noisy areas and creates smoother edges and more homogeneous regions. This results in a more stable depth map with less jittering.

On the other hand, the new version of the Microsoft Kinect has been already released and it is way more accurate and precise than the previous model. The depth sensor is more stable with interference of infrared lights and it has better resolution on the depth map and infrared data stream. The drivers for this new sensor are still in a development phase and there are still several issues that have to be fixed. Moreover, it only supports Microsoft Windows. As a result, this solution it is not yet very practicable in the context of this project at the time we write this document but it is something to bear in mind in the very near future. Besides this, there are a large number of dedicated cameras that can be used to improve the accuracy of the motion capture[1].

In addition, there are other more precise motion capture techniques (see section 2.1) that can provide with accurate skeletal data from the performers. This information would allow us to create particular graphics that respond better to the human pose and to specific motions. This could also be achieved through the use of the EyeCon and EyesWeb technologies mentioned in section 2.3. [Piana et al., 2014] describes various ways of extracting features from body movements and recog-

[1] http://www.creativeapplications.net/tutorials/guide-to-camera-types-for-interactive-installations/

nition of emotions in real-time using optical motion capture, which are used in EyesWeb.

Chapter 4

GENERATION OF GRAPHICS IN REAL-TIME

This chapter explains the algorithms used for generating the real-time graphics based on the data from the motion capture. First it introduces the problem of graphics generation and the list of functional and non-functional requirements that was considered from this part of the system. Next it describes briefly the maths behind the algorithms and how they are used inside *CREA*. The chapter ends up with a short conclusion.

4.1 Introduction

There are many different algorithms and methods to generate interactive graphics. The idea from the very beginning was to focus on two-dimensional abstract and minimalistic graphics, using basic geometric elements such as lines and circles, that would simulate different behaviors based on the motion capture data. Many of the ideas for the graphics come from the projects mentioned in Chapter 2 and we tried to imitate some of their characteristics. Given that most of them use particle systems to create various effects we decided to focus on this graphic technique, providing several features that would simulate different behaviors.

We also decided to implement a two-dimensional fluid solver on top of the particle systems after a couple of projects[1,2] from the digital artist Memo Atken. The simulation of realistic animations based on physical models such as fluids, are

[1] http://www.memo.tv/reincarnation/
[2] http://www.memo.tv/waves/

things that are already very beautiful to see just by themselves. Given the fact that the motion capture used is not very robust and has noise, creating graphics that do not need constant nor precise motion data input to look beautiful is a really important concept. Besides that, the fluid solver can also be used as an interaction medium with the other graphics, enriching the system by creating a very natural dialogue with the dancer.

Figure 4.1 summarizes the functionality of this part of the system and the sections explained in this chapter. The output data from the motion capture (i.e., *silhouette contour*, optical flow vector field, *IR Markers* position, *IR Markers* velocity...) is fed to the three graphical generation blocks: (1) Fluid Simulation, (2) Particles System Generation and (3) Silhouette Representation. In a step in between the Motion Capture and the graphics generation blocks the motion data is used as an input to a two dimensional Fluid Solver. This solver outputs the fluid behavior data over time which is then used as an extra input to the graphical generation blocks; being able to use it to interact with the graphics in a very organic and natural way. The Fluid Solver and Fluid Simulation are very related and this is why we explain them together in the same section.

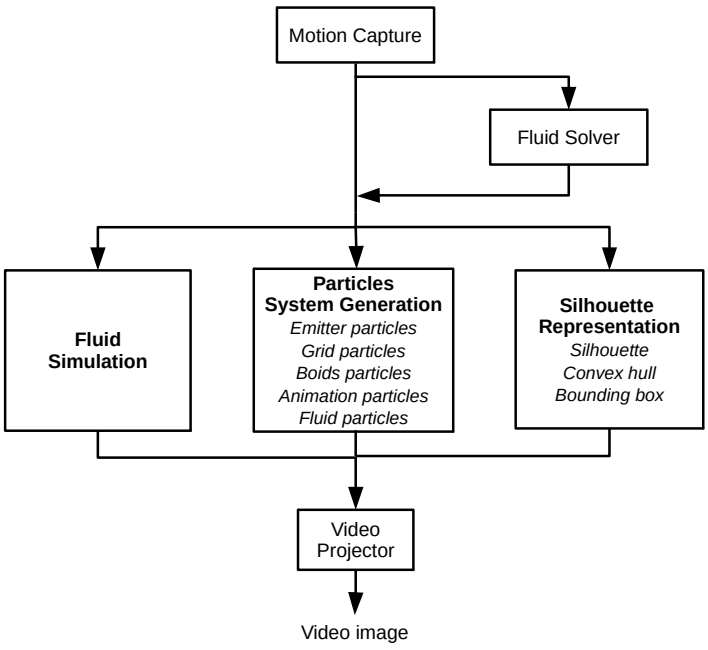


Figure 4.1: Block diagram of the graphics generation part of the system.

4.2 Requirements

Following what we have mentioned in the introduction with respect to the graphics, we created the list of functionalities that the graphics generation module would have to accomplish and the constrains in order to satisfy the proper functioning of the overall system in the proposed context. First we explain the functional requirements and then we continue with the non-functional ones.

4.2.1 Functional Requirements

The list of functional requirements that we defined for the graphics generation system is the following:

1. **Use motion capture**
   The graphics generation system must use the motion capture data as input to the graphics generation algorithms. The graphics must interact with the motion data and also use it as a generating source.

2. **Fluid Solver**
   The graphics generation system must include a two-dimensional fluid solver that uses the motion capture data as control input. This fluid solver must be accessible from the other graphic generation techniques in order to use that as a way to create fluid behaviors.

3. **Fluid Simulation**
   The graphics generation system must use the fluid solver as an input to simulate fluid looking graphics, such as water and fire.

4. **Particle Systems**
   The graphics generation system must create particles with different behaviors. These particles must follow a set of defined rules so they behave as a single system. The particles must simulate effects such as flocking, rain, snow and explosion animations.

5. **User representation**
   The graphics generation system must represent in a few different ways the silhouette of the user extracted from the motion capture.

4.2.2 Non-functional Requirements

The list of non-functional requirements we defined for the graphics generation system is the following:

1. **Fault Tolerance**
   The graphics generation system shall be tolerant to the noise coming from the motion capture data.
2. **Real-time**
   The graphics generation system shall guarantee response within 15 milliseconds from the input data.
3. **Crash recovery**
   The graphics generation system shall be able to restart in case of an unexpected error in the generation algorithms.
4. **Extinsibility**
   The graphics generation system shall take future growth into consideration and be easy to extend its functionalities.

4.3 Fluid Solver and Simulation

In order to implement a fluid solver we started reading a number of references on computer simulation of fluid dynamics and on different types of implementations ([Harris, 2004], [Stam, 1999], [Stam, 2003], [West, 2008] and [Rideout, 2010]). Meanwhile we also learned about the basics of the OpenGL Shading Language ([Rost and Licea-Kane, 2009], [Karluk et al., 2013] and [Vivo, 2015]) in order to understand how to program the fluid solver using the GPU capabilities. However, after some time learning about fluids we discovered that two addons for openFrameworks already implemented a two-dimensional fluid simulation. These addons are already really optimized so we decided to use them instead of reprogramming everything.

These two addons are: (1) ofxFluid[3] and (2) ofxFlowTools[4]. First we started using (1) for its simplicity (only 2 files compared to 47 from (2)) but after some time adapting it to the system we realized we were constantly checking (2) code as a reference, so we decided to try (2). This implements an optical flow written in GLSL (see section 3.6), it has way more flexibility on the fluid parameters and

[3] https://github.com/patriciogonzalezvivo/ofxFluid
[4] https://github.com/moostrik/ofxFlowTools

ready to use features to use a live camera input to interact with the fluid. The operating principle and approach is the same in both, they are based on [Stam, 1999] method, which is not accurate enough for most engineering applications but it is ideal for interactive computer graphical applications. This method is explained in the following section.

4.3.1 Theory

To simulate the behavior of a fluid we need to get a mathematical representation of the state of the fluid at any given time. The most important quantity is the velocity, because it determines how the fluid and the objects inside the fluid move; we express the velocity over time as a vector field (i.e., velocity vector for every point in space) at each time step. There are many different models that have been developed to simulate the fluid dynamics but the model we use is based on the Navier-Stokes equations, which is by far the most used equations among all the actual fluid solvers. We won't explain the physics behind these equations since they are quite complex and there are many sources that already explain them really well, we will just try to explain briefly what they say. Two excellent books to introduce and understand the fundamental concepts of computational fluid dynamics are [Chorin and Marsden, 2000] and [Abbott, 1997].

The basic idea is to determine the velocity field at each time step by solving a set of equations that describe the evolution of a fluid under a variety of forces. These set of equations are the Navier-Stokes equations, which given a velocity field $\mathbf{u}$ and a scalar pressure field p for the initial time, they describe how the fluid changes over time:

$$\nabla \cdot \mathbf{u} = 0 \tag{4.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p - \nu \nabla^2 \mathbf{u} + F \tag{4.2}$$

Where ν is the kinematic viscosity of the fluid, ρ is its density and F englobes all the external forces acting on the fluid.

These equations come from assuming that the fluid is incompressible, which means that the density of any subregion of the fluid doesn't change during its motion through space. The equivalent of this is that the divergence of the flow velocity is zero, which is what equation 4.1 says.

Using a mathematical result known as Helmholtz-Hodge Decomposition[5] we can

[5] http://en.wikipedia.org/wiki/Projection_method_(fluid_dynamics)

derive equations 4.1 and 4.2 to a single equation that is easier to solve, without the pressure p term:

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{P}(-(\mathbf{u} \cdot \nabla)u + \nu \nabla^2 \mathbf{u} + F) \tag{4.3}$$

Where $\mathbf{P}$ is called the projection operator and it is used to project a vector field onto a divergence free vector field that satisfies the incompressibility of the fluid (divergence zero at all points in the field $\nabla \cdot \mathbf{u} = 0$).

Equation 4.3 is solved by sequentially solving each of the terms inside the parenthesis on the right hand side: (1) advect[6] the field and transport the velocities and other quantities around the space ($-(\mathbf{u} \cdot \nabla)\mathbf{u}$), (2) diffuse the field due to the viscous friction within the fluid ($\nu \nabla^2 \mathbf{u}$), (3) add all the external forces applied to the fluid (F) and (4) force the velocity to conserve mass applying the projection operator ($\mathbf{P}$). The details on how to solve each of these steps numerically and how to apply the projection operator is described precisely in [Stam, 1999] and [Harris, 2004].

On top of this there are other terms that can be added in the function to extend the features of the fluid simulation, such as (1) buoyancy and (2) vorticity confinement. Buoyancy is the force that acts on the fluid due to the changes of temperature. We simply need to add a new scalar field (i.e., temperature) that adds a force where the local temperature is higher than the given ambient temperature. Vorticity is a measure of spinning and rotation of the fluid and vorticity confinement what makes the fluid preserve better this rotation over time. This is done by measuring where vorticity is being lost and injecting rotational motion in these areas so they keep spinning longer.

4.3.2 Implementation

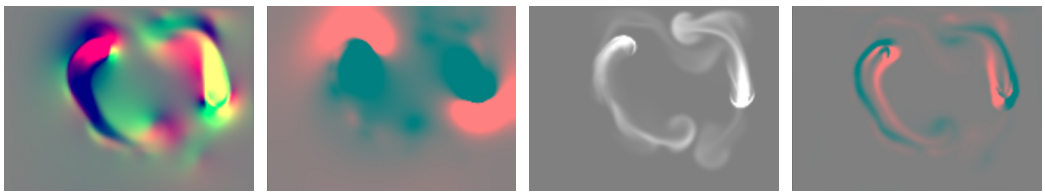
In terms of the implementation there is a two-dimensional grid of cells storing different data types that describe the state of the fluid (e.g., velocity vector, density, pressure, temperature, vorticity and any other substances that we want to propagate with the fluid flow). The process is to keep solving the equations for the different substances at each time step and update the values of the cells.

If this was implemented using the CPU there would be a pair of nested loops iterating over each cell in the grid to perform the same computation over and over and update its values. However, since the operations at each cell are identical, we can take advantage of the parallelism of the GPU and do these computations

[6] http://en.wikipedia.org/wiki/Advection

simultaneously; instead of using grids of cells we use textures that store the data in its pixels (e.g., the 'x' magnitude of the velocity can be stored in the red channel, the 'y' magnitude in the green channel and the blue and alpha channels can be used to store the temperature or any other pressure scalar, see Figure 4.2) and run fragment shaders to update its values. For each of the steps that solve equation 4.3 and that we have explained in the previous section there is a fragment shader that solves the equation in the specific pixel. The code of these fragment shaders is simply the transcription of the mathematical formulas (see [Harris, 2004] for more details). The two most important textures to represent the state of the fluid are the fluid velocity (see Figure 4.2a) and the scalar pressure field (see Figure 4.2b).

To draw a simulation of the fluid we just need to draw the texture containing the color density carried by the fluid (see Figure 4.3d and Figure 4.4d) but we can use any of the other textures to do anything we can imagine (e.g., use the velocity field to move objects, draw the velocity vector field as a grid of vectors, use the temperature texture to draw smoke densities).



(a) Velocity **(b)** Pressure **(c)** Temperature **(d)** Vorticity

Figure 4.2: State fields of the fluid simulation stored in textures.

The behavior of the fluid can be controlled by altering the values of the state field textures. This is done by creating new textures and summing them to the fluid state textures before solving the equations. As an example, if we create a texture with a pixel in the red channel with value 100 and the rest to 0 and we sum it to the velocity state field texture (see Figure 4.2a), the solver will interpret this as an horizontal force of magnitude 100 being applied in that pixel position. We have created two different methods to control the behavior of the fluid, one using the *silhouette contour* data and the other using the *IR Markers*.

Silhouette Contour Input

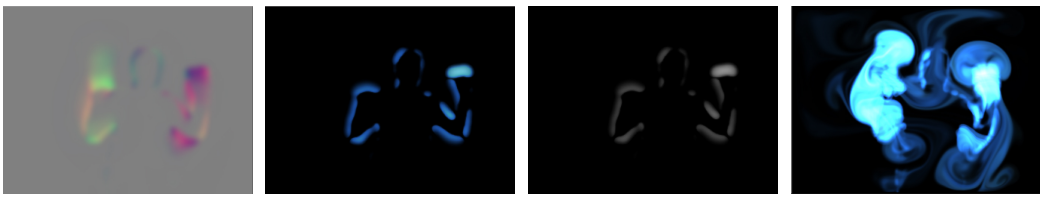
Uses the motion capture data that comes from the depth map to control the fluid. To change the acceleration of the velocity field of the fluid we use the optical flow texture (see Figure 4.3a); summing the pixel values of this texture to the fluid

velocity field (see Figure 4.2a) and multiplying the former by a scalar to control how much it effects to the flow.

To add density to the fluid we sum the velocity mask (see section 3.7) to the fluid density texture (see Figure 4.3d), adding density only the areas where there is motion. This density is the one that defines the color of the fluid and since the velocity mask is white by default, our hack was to tint the binary depth image with the desired color before using it to compute the velocity mask (see Figure 4.3b). To tint the image we use a framebuffer[7] and below there is the code that does that, with an extra method that changes the color randomly over time using Perlin Noise:

```
// tint binary depth image (silhouette)
coloredDepthFbo.begin(); // start to draw in the framebuffer
  ofPushStyle();
  ofClear(255, 255, 255, 0); // clear buffer
  if(vMaskRandomColor){ // tint image with random colors using perlin noise
    vMaskColor.setHsb(ofMap(ofNoise(ofGetElapsedTimef()*0.3), 0.1, 0.9, 0, 255,
    true), 255, 255);
    vMaskRed = vMaskColor.r;
    vMaskGreen = vMaskColor.g;
    vMaskBlue = vMaskColor.b;
  }
  // tint image with specified color
  vMaskColor.set(vMaskRed, vMaskGreen, vMaskBlue, vMaskOpacity);
  ofSetColor(vMaskColor); // set draw color
  depthImage.draw(0, 0, width, height); // draw binary image in the buffer
  ofPopStyle();
coloredDepthFbo.end();
```

Finally we also inject temperature in the areas of motion by using the luminance channel from the velocity mask texture (see Figure 4.3c), being the pixel intensities the temperature in that area. This generates buoyancy due to the differences in temperature.



(a) Optical Flow **(b)** Density **(c)** Temperature **(d)** Density State Field

Figure 4.3: Silhouette Contour Input textures for the Fluid Solver and density state field texture.

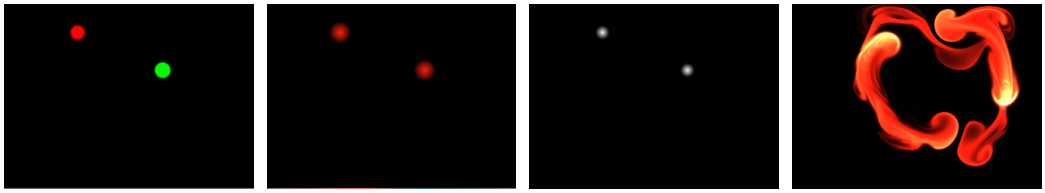
[7] http://openframeworks.cc/documentation/gl/ofFbo.html

IR Markers Input

Uses the data from the *IR Markers* to control the fluid by using a very similar procedure as the one described for the *Silhouette Contour Input*. To change the acceleration of the fluid flow we use the velocity of the marker; creating a texture with circles in the *IR Markers* positions encoding the velocity data (i.e., the red channel contains the magnitude in 'x' of the marker velocity and the green channel the magnitude in 'y') and summing the texture to the fluid velocities texture. The bigger the radius of the circles the largest the region that will be affected by the marker velocity (see Figure 4.4a).

For injecting color we follow the exact same idea but the circles are drawn with the color we want to add to the density texture (see Figure 4.4b). By blurring the edges of the circles the injected density is more subtile and smooth.

Finally for injecting temperature and generating buoyancy we just draw the circles with its color intensity as the desired temperature (see Figure 4.4c).



(a) Velocity **(b)** Density **(c)** Temperature **(d)** Density State Field

Figure 4.4: IR Markers Input textures for Fluid Solver and density state field texture.

4.4 Particle System Generation

A particle system is a really common technique in computer graphics that uses a large number of independent objects to simulate different effects and behaviors. The idea is to be able to manage the number of particles in a flexible way (create and delete the particles) and update their attributes over time according to a set of rules (e.g., gravity, friction, forces).

Chapter 3 of [Perevalov, 2013] and Chapter 4 of [Shiffman, 2012] describe how to implement a basic particle system with some simple behavior functionalities.

First we explain how the equations that describe the motion of the particles are derived and how the class of a single particle was implemented from those equations. Next we explain how the particle system class was implemented, which is

the one that manages the particles and makes them behave as a single element; describing how the motion capture data is used to interact in different ways with the particles. Finally we explain the five different particle system behaviors created: (1) Emitter particles, (2) Grid particles, (3) Boids particles, (4) Animations particles and (5) Fluids particles. Although these share the same basic structure, the set of rules that control the particles are very different and that is why they are explained separately.

4.4.1 Theory

In order to describe the motion of a particle we need to save its position and velocity and apply forces to adjust its acceleration. Given these attributes in the current time step, the challenge is to know how to predict the values in the next time step. The formula that dictates the motion of objects is Newton's second law:

$$\vec{F} = m\vec{a} = m\frac{\partial \vec{v}}{\partial t} = m\frac{\partial^2 \vec{x}}{\partial t^2} \tag{4.4}$$

Where $\vec{F}$ is the force, m is the mass, $\vec{a}$ is the acceleration, $\vec{v}$ is the velocity, $\vec{x}$ is the position and t is time.

Knowing that the velocity is the derivative of acceleration and position the derivative of velocity, we can separate Newton's second law in two first order differential equations and solve them following Euler's method. The solution is an approximation of how position and velocity change over time given the actual values:

$$\vec{v'} = \vec{v} + \vec{a}\Delta t \tag{4.5}$$

$$\vec{x'} = \vec{x} + \vec{v}\Delta t \tag{4.6}$$

Where $\vec{v'}$ is the new velocity, $\vec{v}$ is the actual velocity, $\vec{a}$ is the acceleration, $\vec{x'}$ is the new position, $\vec{x}$ the actual position and Δt the time step (time between frames).

There are other methods to approximate the solution of differential equations that are more precise (e.g., Verlet integration, Runge-Kutta method) but for the purpose of this system physical accuracy is secondary.

4.4.2 Single Particle

Following the previous equations we can implement the basic functionality to describe the motion of a single particle over time. The steps that we do at every frame are: (1) compute the acceleration of the particle by summing all the external forces acting on it divided by its mass (Newton's second law), (2) add the acceleration multiplied by the time step to the velocity, (3) add the velocity multiplied by the time step to the position and (4) restart the forces. In order to make this motion more realistic we also add a 'friction' variable that makes the velocity of the particle decay. This variable ranges from 0.9 to 1.0 and multiplying it by the velocity after summing the acceleration makes the particle slow down in a very natural way:

```
    acc = frc/mass;      // Newton's second law F = m*a
    vel += acc*dt;       // Euler's method
    vel *= friction;     // Decay velocity
    pos += vel*dt;
    frc.set(0, 0);       // Restart force
```

In order to control the motion of the particle we have to apply external forces in the direction we want the particle to move, changing its acceleration. The method that does that is the following:

```
void Particle::addForce(ofPoint force){
  frc += force;
}
```

Besides this, we decided it would be helpful to have a couple of methods more to make the particle attract to or repulse from a specific source point. The method that adds a repulsion force from a point is the following:

```
void Particle::addRepulsionForce(ofPoint posOfForce, float radiusSqrd, float
    scale){

  // (1) calculate the direction to force source and distance
  ofPoint dir         = pos - posOfForce;
  float distSqrd      = pos.squareDistance(posOfForce);

  // (2) if close enough update force
  if (distSqrd < radiusSqrd){
    float pct = 1 - (distSqrd / radiusSqrd);  // stronger on the inside
    dir.normalize();
    frc += dir * scale * pct;
  }
}
```

This method applies a repulsion force with magnitude 'scale' to the particle if the point 'posOfForce' is closer to a distance 'radiusSqrd'. First we get the direction of the repulsion force (i.e., vector pointing from the point towards the particle), by subtracting the position of the point to the position of the particle. Then we

compute the square distance between the force source and the particle and check if this distance is smaller than the minimum repulsion distance. If the particle is close enough to the source of force, we get the ratio of this distance with the minimum repulsion distance ('pct') and multiply it by the direction (normalized) of the force and the strength of the repulsion force. This ratio makes the particle repulse faster if it is closer to the repulsion point.

The attraction force method ('addAttractionForce') is exactly the same with the only difference that the direction of the force is inverted.

The fact that we use the square distance and not the distance itself is that the square root calculation is much more processor-intensive than multiplying; the use of square roots inside nested loops really makes the frame rate slow down. Due to this we use square distances and square lengths all the time throughout the code.

We can extend the particle attributes adding as many variables as we want. These are some of the extra control variables that the particle class includes:

id Float number that uniquely identifies the particle.

iniPos Position of the particle when it borns.

age Time that the particle has lived.

lifetime How much time the particle will live.

radius Actual size of the particle.

iniRadius Radius of the particle when it borns.

color Color of the particle.

opacity Opacity of the particle.

With these extra variables we can have much more control on the behavior of the particle over time. Make the particle decrease size throughout its lifespan, change its color, opacity or make it return to its original position.

Another very useful function created was to add some randomness to the motion of the particle using Perlin noise[8]; this is a pseudo-random noise that creates a smoothly interpolated sequence of values that is very frequently used in computer graphics to simulate various organic effects. The great advantage of this noise function is that it can be defined in any dimension and it always gives a consistent value for a particular location, creating a very natural looking evolution of the random values.

[8] http://en.wikipedia.org/wiki/Perlin_noise

This is the code of the function in question, which adds a random vector with magnitude 'turbulence' to the external forces acting on the particle:

```
void Particle::addNoise(float turbulence){
  // Perlin noise
  float angle = ofSignedNoise(id*0.001f, pos.x * 0.005f,  pos.y * 0.005f,
    ofGetElapsedTimef() * 0.1f) * 20.0f;
  ofPoint noiseVector(cos(angle), sin(angle));
  if(!immortal) frc += noiseVector * turbulence * age; // if immortal this
    doesn't affect, age == 0
  else frc += noiseVector * turbulence;
}
```

Since the force is in two dimensions we need to get a vector also in two dimensions that can be summed to the force. In order to get that we treat the noise as an angle and get the 'x' and 'y' offset, which define the coordinates of a vector that smoothly changes over time. Having that we just multiply this vector by the 'turbulence' variable, to control the amount of noise, and by the age of the particle so the effect increases as the particle gets older.

To compute the noise we use the 'id' of the particle, its current position and the elapsed time since the application started. By using the identifier 'id' we get a unique noise result for each particle and the elapsed time makes the noise function output different values even when the particle rests in the same position, since the elapsed time increments gradually (if we give the same arguments to the Perlin noise function it outputs the same value every time).

The 'ofSignedNoise' built-in noise function from openFrameworks outputs a value between -1.0 and 1.0, having to multiply this by a scalar if we want this value to represent an angle in the range of all the radians in a circle. The noise is multiplied by 20.0 and not by π because the output range does not result in an even distribution (often the results stay betwen -0.25 and 0.25), thus the random motion of the particles tend to favor a specific direction. By multiplying by 20.0 the angle range is greater and the distribution is more even.

4.4.3 Particle System

The particle system class manages the collection of particles and defines the set of rules we want the particles to follow over time. In order to do that it has a dynamic vector with all the particles and various methods to update their attributes and define their behavior. There are many different behaviors that we can simulate with particle systems but we decided to separate these in five different groups: (1) Emitter particles, (2) Grid particles, (3) Boids particles, (4) Animations particles and (5) Fluid particles. First we explain the general features that these types have in common and next the particularities of each of the categories.

The approach to make the particles interact is the same for the five groups but some interactions are only meaningful for certain groups. The interaction can either be between particles or from an input source (*IR Markers* and *Silhouette Contour*). The interactions we have implemented are: (1) Particle-particle repulsion, (2) Particle-particle connection, (3) Flow interaction, (4) Fluid interaction, (5) Repulse interaction, (6) Attract interaction, (7) Seek interaction, (8) Gravity interaction and (9) Bounce interaction.

When using the *IR Markers* as input, since there can be more than one marker being tracked, we need to have a way to make the particles decide which marker to interact with. As a result, we implemented a method to compute the closest marker to the particle:

```
irMarker* ParticleSystem::getClosestMarker(const Particle &particle,
    vector<irMarker> &markers, float interactionRadiusSqrd){

  irMarker* closestMarker = NULL;
  float minDistSqrd = interactionRadiusSqrd;

  // Get closest marker to particle
  for(int markerIndex = 0; markerIndex < markers.size(); markerIndex++){
    if (!markers[markerIndex].hasDisappeared){
      float markerDistSqrd =
    particle.pos.squareDistance(markers[markerIndex].smoothPos);
      if(markerDistSqrd < minDistSqrd){
        minDistSqrd = markerDistSqrd;
        closestMarker = &markers[markerIndex];
      }
    }
  }

  return closestMarker;
}
```

Basically it computes the square distance between the particle position and the different markers and returns a pointer to the closest if the particle is within the interaction radius area of the marker. Having the pointer to the closest marker we can apply any change of behavior we can think of based on the attributes stored in the marker class (e.g., position and velocity of the marker).

For the *Silhouette Contour* we also need to get the closest point from the different silhouettes detected in the interaction space. The approach is very similar:

```
ofPoint ParticleSystem::getClosestPointInContour(const Particle& particle, const
    Contour& contour, bool onlyInside, unsigned int* contourIdx){

  ofPoint closestPoint(-1, -1);
  float minDistSqrd = 99999999;

  // Get closest point to particle from the different contours
  for(unsigned int i = 0; i < contour.contours.size(); i++){
    if(!onlyInside || contour.contours[i].inside(particle.pos)){
      ofPoint candidatePoint = contour.contours[i].getClosestPoint(particle.pos);
      float pointDistSqrd = particle.pos.squareDistance(candidatePoint);
```

```
      if(pointDistSqrd < minDistSqrd){
        minDistSqrd = pointDistSqrd;
        closestPoint = candidatePoint;
        if(contourIdx != NULL) *contourIdx = i; // save contour index
      }
    }
  }

  return closestPoint;
}
```

If the variable 'onlyInside' is equal to true the method returns the closest point only if the particle is inside the *silhouette contour*, being able to use this so we only interact with the particles inside the silhouette. Otherwise if the variable is equal to false, it returns the closest point to the particle no matter where this is. Moreover, the method also stores the index of the contour in the 'contourIdx' pointer so we can use all the set of points conforming the *silhouette contour* to interact with the particle (see Bounce Interaction).

Particle-particle repulsion

This adds a repulsion force to the particles when they get too close. The most basic implementation would be to add two loops that for every particle compares how close is this one to all the others and applies a repulsion force if the distance is smaller than a threshold:

```
void ParticleSystem::repulseParticles(){
  float repulseDistSqrd = repulseDist*repulseDist;
  for(int i = 0; i < particles.size(); i++){
    for(int j = 0; j < particles.size(); j++){
      if(j!=i) particles[i]->addRepulsionForce(particles[j].pos,
    repulseDistSqrd, 30.0);
    }
  }
}
```

The complexity of this approach is $O(n^2)$ since each particle is compared with all the others, being really slow if we have a few hundreds of particles.

One improvement is to pass by reference the particle from where we apply the repulsion force and add the same force in the opposite direction also to this particle. We created a modified version of the method 'addRepulsionForce' explained before:

```
void Particle::addRepulsionForce(Particle &p, float radiusSqrd, float scale){

  // (1) make a vector of where the particle p is:
  ofPoint posOfForce;
  posOfForce.set(p.pos.x, p.pos.y);

```

```
  // (2) calculate the direction to particle and distance
  ofPoint dir         = pos - posOfForce;
  float distSqrd      = pos.squareDistance(posOfForce);

  // (3) if close enough update both forces
  if (distSqrd < radiusSqrd){
    float pct = 1 - (distSqrd / radiusSqrd);  // stronger on the inside
    dir.normalize();
    frc   += dir * scale * pct;
    p.frc -= dir * scale * pct;
  }
}
```

Using this method we only need to do half of the iterations:

```
void ParticleSystem::repulseParticles(){
  float repulseDistSqrd = repulseDist*repulseDist;
  for(int i = 1; i < particles.size(); i++){
    for(int j = 0; j < i; j++){
      particles[i]->addRepulsionForce(*particles[j], repulseDistSqrd, 30.0);
    }
  }
}
```

Another big improvement is to sort the vector of particles spatially at each iteration so we only check the particles that are close to each other. In this case we decided to sort the vector by the 'x' coordinate, having the vector ordered by how the particles appear from left to right in the space. Having the vector ordered we only have to compare each particle with the ones that are within the horizontal range of the minimum repulsion distance. Because of that, we start to compare the particle with the one right next to it (`j = i-1`) and stop the loop when the distance between them is greater than the minimum:

```
bool comparisonFunction(Particle * a, Particle * b) {
  return a->pos.x < b->pos.x;
}

sort(particles.begin(), particles.end(), comparisonFunction);

void ParticleSystem::repulseParticles(){
  float repulseDistSqrd = repulseDist*repulseDist;
  for(int i = 1; i < particles.size(); i++){
    for(int j = i-1; j >= 0; j--){
      if (fabs(particles[i]->pos.x - particles[j]->pos.x) > repulseDist){
        break; // stop second loop and jump to next particle
      }
      particles[i]->addRepulsionForce( *particles[j], repulseDistSqrd, 30.0);
    }
  }
}
```

There are better techniques to make this comparisons even more efficiently, all of them based on spatially organizing the particles so the comparisons are constrained to the closer particles (see section 4.6).

Particle-particle connection

This draws a line connecting the particles if they are closer than a certain distance. Basically it compares each particle with all the others and if they are closer than a threshold it draws a connecting line in between. In order to compare each particle with the others efficiently we use the same technique described in *Particle-particle repulsion*:

```
if(drawConnections){
  ofPushStyle();
  float connectDistSqrd = connectDist*connectDist;
  ofSetColor(ofColor(red, green, blue), opacity);
  ofSetLineWidth(connectWidth);
  for(int i = 0; i < particles.size(); i++){
    for(int j = i-1; j >= 0; j--){
      if (fabs(particles[j]->pos.x - particles[i]->pos.x) > connectDist){
        break; // stop second loop and jump to next particle
      }
      if(particles[i]->pos.squareDistance(particles[j]->pos) < connectDistSqrd){
        ofLine(particles[i]->pos, particles[j]->pos);
      }
    }
  }
  ofPopStyle();
}
```

Flow Interaction

The approach for this interaction is different for the two possible inputs: *Silhouette Contour* and *IR Markers*.

For the case of the *Silhouette Contour* input this interaction is based on the output of the optical flow vector field (see section 3.6), which is stored in a texture with the red channel being the horizontal offset and the green channel the vertical. The principle is to apply to every particle the velocity that corresponds to the same position in the flow field; adding this velocity as an external force that acts on the particle. In order to do that we had to implement a method to get the velocity given the position of the particle.

Since for computing the optical flow we use a lower resolution image of the Kinect depth map, the first step to get the velocity in a specific coordinate is to scale the coordinate system down so the coordinates of the particles match the coordinates of the optical flow texture. After that, we simply get the color of the texture in that coordinate and assign the red channel to the x coordinate and the green channel to the y coordinate of the velocity vector:

```
ofVec2f Contour::getFlowOffset(ofPoint p){
  ofPoint p_ = p/scaleFactor; // scale point to match flow texture
```

```
  ofVec2f offset(0,0);

  if(rescaledRect.inside(p_)){ // if point is inside flow texture size
    offset.x = flowPixels[((int)p_.y*flowWidth+(int)p_.x)*3 + 0]; // r
    offset.y = flowPixels[((int)p_.y*flowWidth+(int)p_.x)*3 + 1]; // g
  }

  return offset;
}
```

The only secret of this function is how to read the pixels of the texture by its coordinates. Whenever we want to manipulate the pixel data of a texture (ofTexture), which resides in memory in the GPU, we first need to transfer the texture data to the CPU (ofPixels), like we do here in this part of code:

```
ofFloatPixels flowPixels;
// Save optical flow texture and get its pixels to read velocities
ofTexture flowTexture = opticalFlow.getOpticalFlowDecay();
flowTexture.readToPixels(flowPixels);
```

When we do this we are saving the texture as a one dimensional array of pixel values, storing each row of the texture next to each other. Hence, if we want to access the 10th pixel of the 2nd row of the texture we would simply multiply by 2 the width of the texture and sum 10 (flowPixels[2*textureWidth+10]). Since the flow texture has three channels (RGB), for each pixel there are three values, stored one after the other in the array, having to multiply by 3 the previous result to access the pixel in that coordinate. Then we just sum 1 or 2 to read the second and third channel respectively.

The reason why we use 'ofFloatPixels' instead of 'ofPixels' is that the latter stores its values in 1 byte (unsigned char) and the former in 4 bytes (float) and the flow texture data is also stored in 4 bytes. We explicitly mention this since it was a common error in the code and it took us some time to realize what was the error.

Having the velocity vector in the particle position we just apply it as an external force acting on the particle:

```
unsigned int contourIdx = -1;
ofPoint closestPointInContour;
if(particleMode == BOIDS && seekInteraction) closestPointInContour =
    getClosestPointInContour(*particles[i], contour, false, &contourIdx);
else closestPointInContour = getClosestPointInContour(*particles[i], contour,
    true, &contourIdx);

if(flowInteraction){
  ofPoint frc = contour.getFlowOffset(particles[i]->pos);
  particles[i]->addForce(frc*interactionForce);
}
```

For the case of *IR Markers* this interaction is based on the velocity of the marker. We simply get the velocity of the closest marker within the interaction area and

add this velocity as a force that acts on the particle; multiplying this velocity by the ratio of how far is the marker from the particle in comparison with the interaction area (pct) makes the particles closer to the marker move faster and results in a smoother transition between the interacted particles and the ones that are not:

```
// Get closest marker to particle
irMarker* closestMarker;
if(particleMode == BOIDS) closestMarker = getClosestMarker(*particles[i],
    markers);
else closestMarker = getClosestMarker(*particles[i], markers,
    interactionRadiusSqrd);

if(closestMarker != NULL){
  if(flowInteraction){
    float markerDistSqrd =
    particles[i]->pos.squareDistance(closestMarker->smoothPos);
    float pct = 1 - (markerDistSqrd / interactionRadiusSqrd); // stronger on the
    inside
    particles[i]->addForce(closestMarker->velocity*pct*interactionForce);
  }
}
```

Fluid Interaction

This interaction is based on the output of the fluid vector field. The implementation to get the velocity vector from the fluid velocities is the exact same as for the Flow interaction, with the only difference that here we use the fluid velocity texture instead of the optical flow texture and this has four channels (RGBA) instead of three:

```
ofVec2f Fluid::getFluidOffset(ofPoint p){
  ofPoint p_ = p/scaleFactor;
  ofVec2f offset(0,0);

  if(rescaledRect.inside(p_)){
    offset.x = fluidVelocities[((int)p_.y*flowWidth+(int)p_.x)*4 + 0]; // r
    offset.y = fluidVelocities[((int)p_.y*flowWidth+(int)p_.x)*4 + 1]; // g
  }

  return offset;
}
```

Also in this interaction there is no difference between *IR Marker* and *Silhouette Contour* input since it uses directly the fluid velocities texture resulting from the fluid solver. Being that said, if we want to follow the velocities that come from the marker we will need to choose the *IR Marker* input as the interacting force for the fluid solver and the same thing for the *Silhouette Contour*.

Gravity Interaction

This interaction activates a gravity force on all the particles that are "touched". To do that we simply activate a variable named 'isTouched' for every particle that gets disturbed by the motion data and then apply a force downwards to all the particles with this variable equal to true. In order to make this more natural we add an horizontal random force when we touch the particle so they do not fall in the same place if they are horizontally aligned (in the case of *Grid Particles* this effect is very obvious):

```
unsigned int contourIdx = -1;
ofPoint closestPointInContour;
if(particleMode == BOIDS && seekInteraction) // get closest point to particle
  closestPointInContour = getClosestPointInContour(*particles[i], contour,
    false, &contourIdx);
else // get closest point to particle only if particle inside contour
  closestPointInContour = getClosestPointInContour(*particles[i], contour, true,
    &contourIdx);
// if particle is being touched
if(closestPointInContour != ofPoint(-1, -1)){
  if(gravityInteraction){
    particles[i]->addForce(ofPoint(ofRandom(-100, 100), 500.0) *
    particles[i]->mass);
    particles[i]->isTouched = true;
  }
}
// if particle not being touched but was touched before
else if(gravityInteraction && particles[i]->isTouched){
  particles[i]->addForce(ofPoint(0.0, 500.0) * particles[i]->mass);
}
```

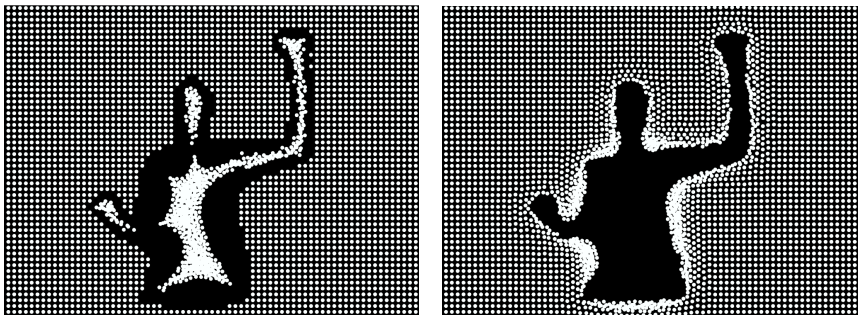
Multiplying the force by the mass of the particle prevents the acceleration to depend on the mass, meaning that regardless of the size of the particle (in the code the mass is proportional to the size of the particle) they are all going to fall with the same acceleration and hit the floor at the same time, which is true for gravity forces in absence of air resistance.

Repulse Interaction

This interaction applies a repulsion force to all the particles that are closer to a certain distance from the interaction point. In the case of *IR Markers* it takes the closest marker position to apply the repulsion force and for the *Silhouette Contour* the closest point from the set of contour points.

For the *IR Markers* input we use the method 'addRepulsionForce' explained before, with the marker position as the repulsion source.

For the *Silhouette Contour* we use the method 'addAttractionForce' instead because otherwise the result is a bit confusing and we believe this way makes more



(a) Attract Interaction **(b)** Repulse Interaction

Figure 4.5: Image showing the resulting attraction and repelling interactions, using a repulsion force to simulate the attraction and a attraction force for the repulsion.

sense to the user. With the attraction force all the particles that are inside the contour are attracted to its closest point in the contour, having all the particles move around the silhouette and none inside. However, with the repulsion force the particles inside the silhouette try to move away from the contour points and they all remain inside the contour. Because of this reason, the attraction force looks as if it was repelling the particles and the repulsion force as if it was attracting them (see Figure 4.5).

Attract Interaction

This interaction applies an attraction force to all the particles closer to a certain distance from the interactive point. The approach is the same as for the repulse interaction but with the force inverted, using the method 'addAttractionForce' for the *IR Markers* and 'addRepulsionForce' for the *Silhouette Contour*.

Seek Interaction

This interaction makes the particles steer towards a target, for which we use the closest IR Marker position and the closest point in the *Silhouette Contour* as targets. The implementation is really similar as the 'addAttractionForce', with the difference that here all the particles steer towards the target no matter what is the distance. The 'interactionRadius' makes the particles slow down when they are inside the area while in the attraction force only the particles inside the area are attracted:

```
1 void Particle::seek(ofPoint target, float radiusSqrd, float scale){
2     // (1) calculate the direction to target & length
3     ofPoint dirToTarget = target - pos;
```

```
    float distSqrd = pos.squareDistance(target);

    // (2) scale force depending on the distance
    float pct = 1;
    if(distSqrd < radiusSqrd){
        pct = distSqrd / radiusSqrd; // decrease force when closer to target
    }

    // (3) update force
    dirToTarget.normalize();
    frc += dirToTarget * scale * pct;
}
```

However, we decided to make all the particles seek the corresponding closest points only for the *Boids Particles* group, for which 'getClosestPointInContour' and 'getClosestMarker' have a special case and return the closest point to the particle even when this is not inside the silhouette or the interaction area. That is the reason why the outcome of this interaction for all the other groups (i.e., Emitter, Grid, Animations) is similar to the 'Attraction Interaction', although the force in this case is softer and results in a smoother interaction. Below there is the code used to call this interaction method for both inputs (*IR Marker* and *Silhouette Contour*), where you can see the distinction done to get the closest point for the *Boids Particles*:

```
if(markersInput){
  irMarker* closestMarker;
  if(particleMode == BOIDS) // get closest marker to particle
    closestMarker = getClosestMarker(*particles[i], markers);
  else // get closest marker to particle only if particle is inside interaction area
    closestMarker = getClosestMarker(*particles[i], markers, interactionRadiusSqrd);
  if(closestMarker != NULL){
    if(seekInteraction){
      particles[i]->seek(closestMarker->smoothPos, interactionRadiusSqrd, interactionForce*4.0);
    }
    ...
  }
}
if(contourInput){
  unsigned int contourIdx = -1;
  ofPoint closestPointInContour;
  if(particleMode == BOIDS && seekInteraction) // get closest point to particle
    closestPointInContour = getClosestPointInContour(*particles[i], contour, false, &contourIdx);
  else // get closest point to particle only if particle inside contour
    closestPointInContour = getClosestPointInContour(*particles[i], contour, true, &contourIdx);
  // if particle is being touched
  if(closestPointInContour != ofPoint(-1, -1)){
    if(seekInteraction){
      particles[i]->seek(closestPointInContour, interactionRadiusSqrd, interactionForce*4.0);
    }
    ...
  }
```

```
}
```

Bounce Interaction

This interaction makes the particles bounce with the surface of the *silhouette contour*, hence it only works when the interaction input is set to *Contour* (see appendix A). In order to simulate this bounce we created a new method in the particle class that given a set of points conforming a contour it computes the reflection vector of the particle velocity where it hits this contour and assigns it to the velocity of the particle for the next frame.

The formula to get the reflection of a vector across a plane is the following:

$$\vec{v'} = \vec{v} - 2(\vec{n} \cdot \vec{v})\vec{n} \tag{4.7}$$

Where $\vec{n}$ is the normal of the plane, $\vec{v}$ the incoming direction vector and $\vec{v'}$ the reflected vector.

In order to implement this we only need to get the normal vector in the point where the particle hits the contour and apply the previous formula:

```
void Particle::contourBounce(ofPolyline contour){
  unsigned int index;
  ofPoint contactPoint = contour.getClosestPoint(pos, &index);
  ofVec2f normal = contour.getNormalAtIndex(index);
  vel = vel - 2*vel.dot(normal)*normal; //reflection vector
  vel *= 0.35; // damping
}
```

Whenever an object bounces a surface some of its speed is lost (how much depends on the object itself and the surface it hits). That is why we multiply by 0.35 the magnitude of the velocity after reflecting it, simulating this loss of speed.

In terms of the particle system, we use the contour index pointer from 'getClosestPointInContour' method to pass the contour set of points to 'contourBounce', which will only be called whenever the particle is inside the silhouette ('getClosestPointInContour' only returns a point if the given particle is inside the contour):

```
unsigned int contourIdx = -1;
ofPoint closestPointInContour;
if(particleMode == BOIDS && seekInteraction) // get closest point to particle
  closestPointInContour = getClosestPointInContour(*particles[i], contour,
    false, &contourIdx);
else // get closest point to particle only if particle is inside contour
  closestPointInContour = getClosestPointInContour(*particles[i], contour, true,
    &contourIdx);
```

```
7  if(closestPointInContour != ofPoint(-1, -1)){
8    if(bounceInteraction){
9      if(contourIdx != -1)
       particles[i]->contourBounce(contour.contours[contourIdx]);
10   }
11 }
```

4.4.4 Emitter Particles

Creates particles from a source with certain initial attributes (e.g., velocity, radius, color, lifespan). When using the *IR Markers* as the input source the particles emitted inherit the velocity of the marker and in *Silhouette Contour* the velocity from the optical flow.

We created three different kinds of emission, both for *IR Markers* and *Silhouette Contour*, that differentiate in how the particles are emitted from the two input sources (see Image 4.6):

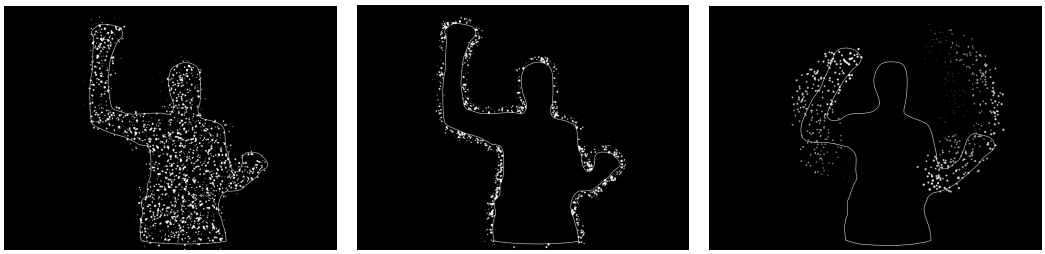
Emit all time A constant number of particles is emitted from the input source at every frame. In the case of *IR Markers* in any part within the circular area defined by the marker and for the *Silhouette Contour* anywhere inside the silhouette.

Emit all time in contour A number of particles are emitted every frame but only on the contour of the input source. In the case of *IR Markers* around the circular area defined by the marker and for the *Silhouette Contour* around the silhouette.

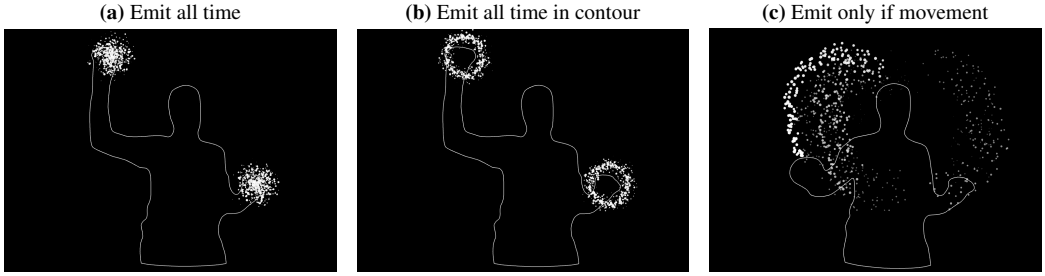
Emit only if movement A different number of particles is emitted depending on the velocity of the input source. In the case of *IR Markers* it maps the velocity of the marker to emit different amounts of particles, creating more particles with higher velocity and no particles if there is no movement. For *Silhouette Contour* we use the velocity mask (see section 3.7) to make particles born only on the areas where there has been some movement.

Below there is the code to create 'n' particles for the *Silhouette Contour* input case, which is a bit more complex than the *IR Markers* but the approach is the same in both. As arguments to the method we pass the amount of particles to create, the optical flow data and the set of points conforming the contour area where the particles will born (either the contour of the silhouette or the contour of the velocity mask so it only emits particles in the motion areas):

```
1 void ParticleSystem::addParticles(int n, const ofPolyline& contour, Contour&
    flow){
2   for(int i = 0; i < n; i++){
```



(a) Emit all time **(b)** Emit all time in contour **(c)** Emit only if movement



(d) Emit all time **(e)** Emit all time in contour **(f)** Emit only if movement

Figure 4.6: Different ways of emitting particles from the two input sources, first row from *IR Markers* and second row from *Silhouette Contour*. Also the silhouette contour is drawn to make it easier to understand.

```
3     ofPoint pos, vel;
4
5     // Create random particles inside contour polyline
6     if(emitAllTimeInside || emitInMovement){
7       // use bounding box to limit space of randomly generated positions
8       // and is faster to find a position which is inside contour
9       ofRectangle box = contour.getBoundingBox();
10      ofPoint center = box.getCenter();
11      pos.x = center.x + (ofRandom(1.0f) - 0.5f) * box.getWidth();
12      pos.y = center.y + (ofRandom(1.0f) - 0.5f) * box.getHeight();
13
14      while(!contour.inside(pos)){
15        pos.x = center.x + (ofRandom(1.0f) - 0.5f) * box.getWidth();
16        pos.y = center.y + (ofRandom(1.0f) - 0.5f) * box.getHeight();
17      }
18
19      // set velocity to random vector direction with 'velocity' as magnitude
20      vel = randomVector()*(velocity+randomRange(velocityRnd, velocity));
21    }
22
23    // Create particles only around the contour polyline
24    else if(emitAllTimeContour){
25      float indexInterpolated = ofRandom(0, contour.size());
26      pos = contour.getPointAtIndexInterpolated(indexInterpolated);
27
28      // Use normal vector in surface as vel. direction so particle moves out of
      the contour
29      vel =
      -contour.getNormalAtIndexInterpolated(indexInterpolated)*(velocity+randomRange(velocityRnd,
      velocity));
30    }
31
```

```
32      ofPoint motionVel = flow.getFlowOffset(pos); // get velocity vector in
        particle pos
33      vel += motionVel*(velocityMotion/100);
34
35      pos += randomVector()*emitterSize; // randomize position within a range of
        emitter size
36
37      float initialRadius = radius + randomRange(radiusRnd, radius);
38      float lifetime = this->lifetime + randomRange(lifetimeRnd, this->lifetime);
39
40      addParticle(pos, vel, ofColor(red, green, blue), initialRadius, lifetime);
41    }
42  }
```

In this code there are a couple of functions that were created that are really useful to randomize the attributes of the particles and since they are used a lot it is worth mentioning them: (1) randomRange and (2) randomVector.

The first one returns a random unit vector around the unit circle and it is useful to get a random direction. This is the code:

```
1  ofPoint ParticleSystem::randomVector(){
2    float angle = ofRandom((float)M_PI * 2.0f);
3    return ofPoint(cos(angle), sin(angle));
4  }
```

The second method, given a scalar value and a percentage of randomness, it returns a random value between the range in between the percentage of the value above and below. For instance if the percentage is 50 and the value is 4.0, it will return a random value between 2.0 and 6.0.

```
1  float ParticleSystem::randomRange(float percentage, float value){
2    return ofRandom(value-(percentage/100)*value, value+(percentage/100)*value);
3  }
```

4.4.5 Grid Particles

Grid Particles creates a grid of a fixed number of particles with infinite lifetime filling the output window. Basically it creates a particle with zero initial velocity every certain distance when we initialize the particle system. It does so by calling the following method:

```
1  void ParticleSystem::createParticleGrid(int width, int height){
2    for(int y = 0; y < height/gridRes; y++){
3      for(int x = 0; x < width/gridRes; x++){
4        int xi = (x + 0.5f) * gridRes;
5        int yi = (y + 0.5f) * gridRes;
6        ofPoint vel = ofPoint(0, 0);
7        addParticle(ofPoint(xi, yi), vel, ofColor(red, green, blue), radius,
       lifetime);
8      }
9    }
10 }
```

One special feature of this particle system is the use of the stored initial position of the particle to make the particle return to its initial state after applying an interaction force. This creates a very natural effect and makes the grid remain intact even after applying different forces to the particles. The method that does this is very similar to the one explained in 'Seek Interaction' but it uses the initial position as a fixed target:

```
void Particle::returnToOrigin(float radiusSqrd, float scale){
  // (1) calculate the direction to origin position and distance
  ofPoint dirToOrigin = iniPos - pos;
  float distSqrd = pos.squareDistance(iniPos);

  // (2) set force depending on the distance
  float pct = 1;
  if(distSqrd < radiusSqrd){
    pct = distSqrd / radiusSqrd; // decrease force when closer to origin
  }

  // (3) update force
  dirToOrigin.normalize();
  frc += dirToOrigin * scale * pct;
}
```

4.4.6 Boids Particles

Creates a fixed number of particles with infinite lifetime that simulate a flocking behavior.

[Reynolds, 1987] is the first article describing a computer model, that the author named Boids, to emulate the flocking behavior; boid is each of the free agents that form the flock which in our implementation is the same as a particle. The basic model consists of three simple rules that describe how each individual boid behaves based on the positions and velocities of the boids nearby: (1) Separation, (2) Alignment and (3) Cohesion. However, we can add more rules to extend the model in several different ways, like make the flocks avoid obstacles or seek a specific target.

There are many source codes to implement this algorithm but our code is based on a Cinder tutorial[9]. Chapter 6: Autonomous Agents in [Shiffman, 2012] describes the model step by step and it is a great source to understand the basics and get some ideas on how to extend its functionality.

The three basic rules of the flocking algorithm (see Figure 4.7) that are applied to each of the boids of the system are the following:

[9] http://libcinder.org/docs/dev/flocking_chapter1.html

Separation The separation rule makes the particle move away from its neighbors, avoiding collisions between them. To do that it checks all the boids within the flocking radius (distance at which the other boids are considered flockmates) to see if the distance between them is too small, and if so, adds a repulsion force to move away from them.

Cohesion The cohesion rule makes the particle stay close together with their flockmates, moving towards the center of its neighbors. In order to do that it averages the location of all the boids within the flocking radius and moves towards this average location.

Alignment The alignment rule makes the particle move in the same direction as its neighbors if it is not too far nor too close from them. In order to do that it averages the velocity vectors of all the boids within the flocking radius (average direction of the flock) and it steers to move in the same direction.

By combining these three rules we simulate the flocking behavior, using the same technique as in 'Particle-particle repulsion' to speed up the comparisons between the different particles. We compare each pair of particles only once and break the second loop when the distance between particles is too far:

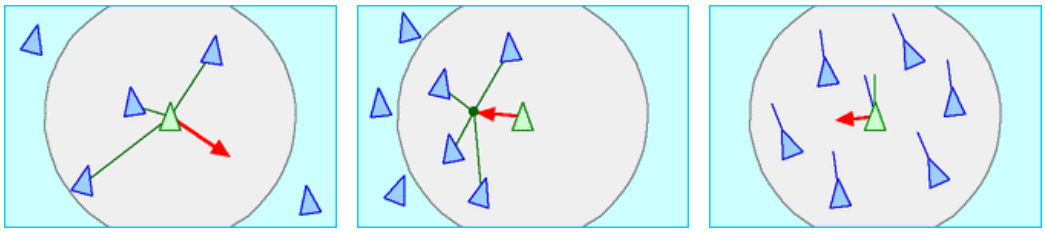
```
void ParticleSystem::flockParticles(){
  for(int i = 0; i < particles.size(); i++){
    for(int j = i-1; j >= 0; j--){
      if (fabs(particles[i]->pos.x - particles[j]->pos.x) > flockingRadius)
    break;
      particles[i]->addFlockingForces(*particles[j]);
    }
  }
}
```

And the code inside the particle class to apply the flocking forces is the following:

```
void Particle::addFlockingForces(Particle &p){
  ofPoint dir = pos - p.pos;
  float distSqrd = pos.squareDistance(p.pos);

  if(0.01f < distSqrd && distSqrd < flockingRadiusSqrd){ // if neighbor particle
    within zone radius...
    float percent = distSqrd/flockingRadiusSqrd;

    // Separate
    if(percent < lowThresh){            // ... and is within the lower threshold
    limits, separate
      float F = (lowThresh/percent - 1.0f) * separationStrength;
      dir = dir.getNormalized() * F;
      frc += dir;
      p.frc -= dir;
    }
    // Align
```



(a) Separation **(b)** Cohesion **(c)** Alignment

Figure 4.7: Flocking behavior basic rules[10].

```
16      else if(percent < highThresh){       // ... else if it is within the higher
        threshold limits, align
17        float threshDelta = highThresh - lowThresh;
18        float adjustedPercent = (percent - lowThresh) / threshDelta;
19        float F = (0.5f - cos(adjustedPercent * M_PI * 2.0f) * 0.5f + 0.5f) *
        alignmentStrength;
20        frc += p.vel.getNormalized() * F;
21        p.frc += vel.getNormalized() * F;
22      }
23      // Attract
24      else{                                // ... else, attract
25        float threshDelta = 1.0f - highThresh;
26        float adjustedPercent = (percent - highThresh) / threshDelta;
27        float F = (0.5f - cos(adjustedPercent * M_PI * 2.0f) * 0.5f + 0.5f) *
        attractionStrength;
28        dir = dir.getNormalized() * F;
29        frc -= dir;
30        p.frc += dir;
31      }
32    }
33  }
```

If the particle we are comparing with is within the flocking radius we add it as a neighbor and we compute the flocking forces that apply to both particles. Depending on how far are the particles we apply one or another rule. If the particles are too close we apply a separation force, if they are too far an attraction force and otherwise we just align their velocities. The values used to compute the force 'F' that we can see in the previous code are just to make the transitions between one rule to the other smoother and more natural.

4.4.7 Animations Particles

Creates particles with different lifetimes simulating three predefined effect animations:

[10]http://www.red3d.com/cwr/boids/

Rain Simulates rain by creating small particles on the upper part of the output window with an initial velocity and a gravity force downwards.

Snow Simulates snow by creating medium-sized particles on the upper part of the output window with an initial velocity and gravity downwards and some Perlin noise to simulate the randomness of the snow flakes falling down.

Explosion Simulates an explosion by creating various particles at once on the bottom part of the output window with a big initial velocity upwards. It also has some gravity force downwards and high friction that simulates the particles slowing down by its own weight. Finally we add some noise using the method 'addNoise' described in section 4.4.2 to give a more organic look.

4.4.8 Fluid particles

Creates particles based on the fluid flow from the fluid solver, which can either be interacted by the *IR Markers* or the *Silhouette Contour*. This is a feature that comes with the *ofxFlowTools* addon and it uses the OpenGL Shading Language to create and update the particles behavior; it takes advantage of the parallelism from the GPU to create a great number of particles without slowing down the system.

Particles change their acceleration based on the fluid velocity and the optical flow textures at each time step, using a very similar approach to the one described in section 4.4.1 to update the motion of the particles. However, here it makes use of textures to take advantage of the GPU instead of doing all the computations in the CPU as in the previous particle system groups.

The implementation consists of a pair of four-channel textures of size (i.e., $width \times height$) equal to the maximum number of particles that we will want to draw in one frame, saving in each pixel the state of a single particle. The first texture saves four basic attributes: age, lifespan, mass and size. The second texture includes the position of the particle, 'x' and 'y' coordinate, using only the first two channels of the texture (see Figure 4.8). In fact, these textures are duplicated due to the fact that a texture cannot be used as input and output at the same time. Because of that we have to use a pair of textures and a double buffering technique (often named Ping-pong technique[11]) to be able to compute the new data from the previous values; we save the current data to one texture while we read from the other and then

[11] http://www.pixelnerve.com/v/2010/07/20/pingpong-technique/

we swap their roles.

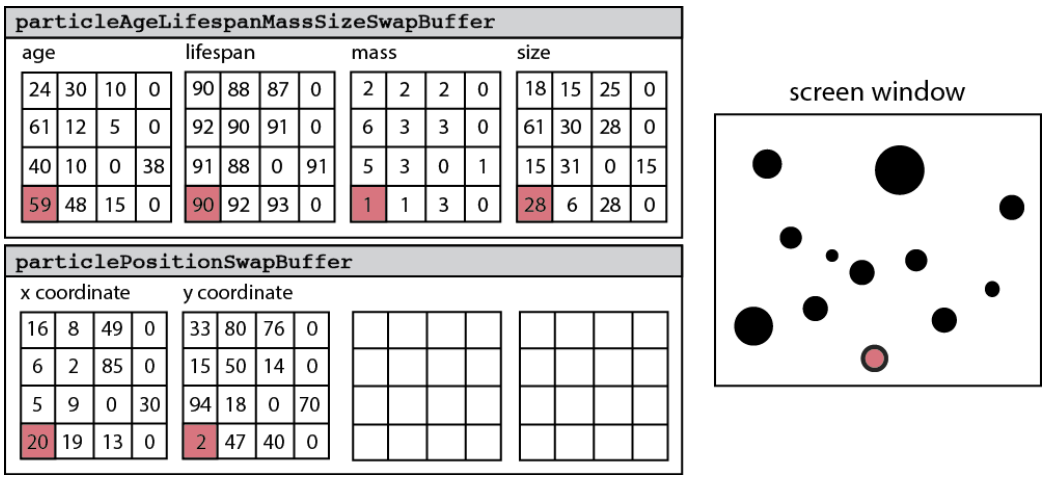


Figure 4.8: Diagram showing the two textures containing the state attributes of the particles we transfer to the GPU to update its behavior over time. In this example we could only draw 16 particles (16 pixels in the texture, although 4 particles are dead) and the red square represents the red particle in the screen (the numbers in the matrices are not exact and it is only to help understand the implementation).

With the use of fragment shaders we update each pixel (i.e., particle attributes) of the textures at every frame; the fluid velocity and optical flow textures are used to update the particle position and to create new particles and the elapsed time since the application started is used to update their age. As a last step it takes the information stored in these pair of textures to draw the particles in the screen; given a mesh of vertices (with as many vertices as particles) it uses a vertex shader to draw each vertex as a circle, of size equal to the particle radius attribute, in the corresponding position.

Very briefly explained, a fragment shader assigns a color value to each pixel and a vertex shader takes a vertex array and transforms these vertices through a series of operations to show them in the screen space.

The way it decides how many particles to create at every iteration is based on a couple of variables: (1) particlesBirthChance and (2) particlesBirthVelocity-Chance, and the amount of motion in each pixel of the screen. These are the steps:

1. The fragment shader checks for every dead particle (pixel with age attribute equal to 0), if there is velocity data in that position. If so, it computes a random value.

2. This random value is compared with ‘particlesBirthChance’ and if it is

lower, it creates a new particle in this pixel position.

3. This particle is initialized with the attributes previously defined for the particle system (lifespan, mass and size) and it starts counting its age at every frame.

Having said that, we can see that the larger the value of 'particlesBirthChance' the most likely will be that new particles are created. On the other hand, the random computed value in 1 is divided by 'particlesBirthVelocityChance' before comparing it with 'particlesBirthChance' in 2, meaning that if the value is high, the random value will be smaller and hence will be also more likely that this is lower than the threshold to create a new particle. Moreover, the value of 'particlesBirthVelocityChance' is directly proportional to the velocity in that pixel, meaning that if the velocity is high it will be more likely that new particles are created.

4.5 Silhouette Representation

These are the graphics that represent the silhouette of the dancer in a few different ways (see Figure 4.10). At first the idea was to create some more options but we ended up deciding to focus on other parts of the system and just create a few examples. The output possibilities are (1) silhouette, (2) convex hull and (3) bounding rectangle. These were inspired by the performance Disorder[12] by CENC (Centre d'Expression Numérique et Corporelle).

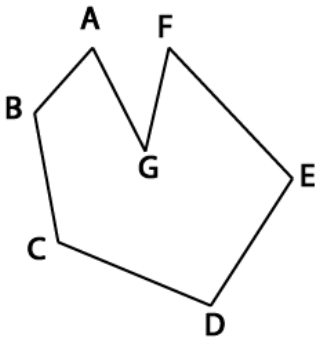


Figure 4.9: Polygon with vertex G non-convex. A vertex is called convex if the angle formed by its two edges, with the polygon inside the angle, is less than 180°.

Silhouette takes the set of points from the *silhouette contour* and considers them as vertices of a closed polygon. Applying a weighted arithmetic mean between the vertices we make the silhouette look less noisy and smoother, being able to set the amount of smoothness by taking more or less vertices in computing the average. If the smoothing size is two, for every vertex in the polygon we take the two vertices on its left, the two on its right and the current vertex and apply a weighted average with all of them, giving more priority (more weight) to the closer vertices to the current.

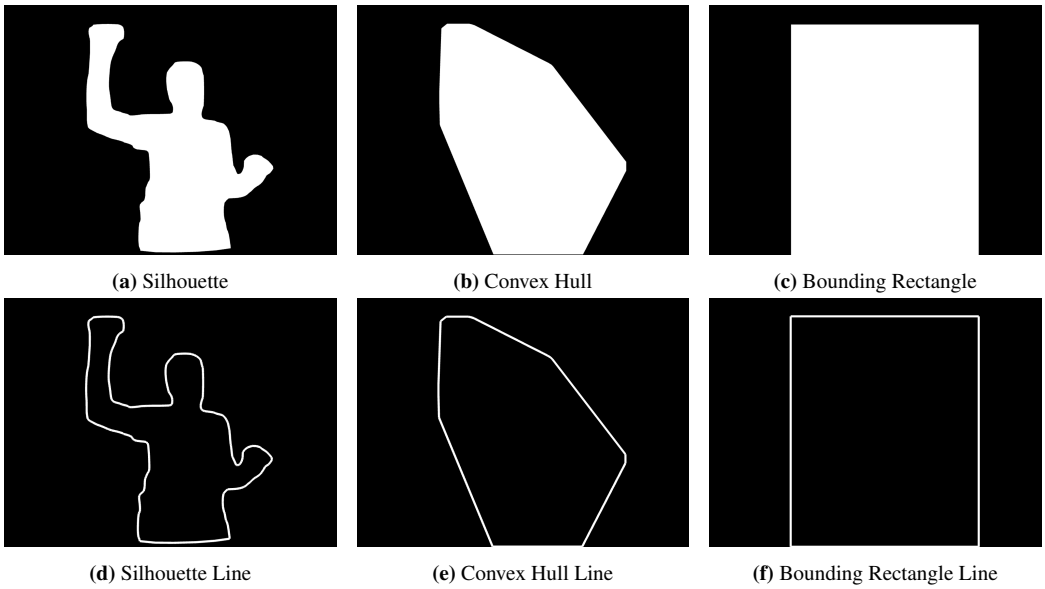
[12]https://vimeo.com/32111537

Convex hull is the smallest convex region enclosing the set of contour points, using the algorithm described in [Sklansky, 1982]. The idea of this algorithm is quite simple. Assuming we have the set of vertices of the polygon ordered, we start the algorithm from one of its four extremal vertices (leftmost, rightmost, top or bottom). At each step we look at the two consecutive vertices (left and right) and compute an equation to determine whether the middle vertex is convex or not (see Figure 4.9). If the vertex is not convex this is deleted and we regress one vertex to check if the vertex is still convex. The algorithm ends when we check the vertex we started the algorithm with.

The steps we just described only work for certain simple polygons and the algorithm OpenCV uses does a few more operations but the principle is the same.

Bounding rectangle is the smallest rectangle enclosing the set of contour points subject to the constraint that the edges of the rectangle are parallel to the coordinate axes (window margins). The resulting bounding rectangle is the same as the minimum rectangle enclosing the convex hull, which is faster to compute. The rectangle is defined by the minimal and maximal value of the corresponding coordinate (x, y) for the set of points of the polygon:

$$\Big\{\big(min(x), min(y)\big), \big(max(x), min(y)\big), \big(min(x), max(y)\big), \big(max(x), max(y)\big)\Big\}$$



(a) Silhouette **(b)** Convex Hull **(c)** Bounding Rectangle

(d) Silhouette Line **(e)** Convex Hull Line **(f)** Bounding Rectangle Line

Figure 4.10: Silhouette representation graphic possibilities.

4.6 Conclusions

In this chapter we covered the algorithms that have been used in the *CREA* system.

The results obtained after implementing the graphic algorithms are satisfactory. The graphics respond very naturally to the motion capture data and after having adapted some of the algorithms to make them more effective, the system works smoothly in real-time even when generating the most computationally expensive graphics. Having said that, we consider the requirements specified for this part have been fulfilled satisfactorily.

However, if I had to start this part of the project again I would probably change a few things. My idea from the start was to create a tool with as many graphical variety choices as possible within the area of particle systems, abstract silhouette representation and fluid simulation. Now I would focus on fewer graphical possibilities in order to have more time to explore deeply all of them. For each of the particle system types created there is a whole world to explore, as well as for the fluids simulation and silhouette representation; slightly changing a few attributes gives a complete new way of interactional behavior.

One thing I wanted to explore more but I haven't had time enough is the use of the GPU to create more complex particle systems. This is a thing for which you first need to understand good enough how particle systems work and the algorithms involved in creating the specific behavior we want to simulate. By using the CPU we can prototype the particle system behavior with a few hundred particles and then port it to the GPU. When I started this project I didn't have experience in simulating particle systems behaviors nor with the workflow to use the GPU functionalities but now I consider I have the knowledge enough to explore these.

Furthermore, I also wanted to explore the techniques that optimize the comparisons between particles in the CPU by spatially organizing them in data structures, such as geometric hashing[13] and quad-trees[14,15].

The graphics component is closely tied to the rest of the *CREA* application so in the final chapter I include the overall conclusions.

[13] http://en.wikipedia.org/wiki/Geometric_hashing
[14] http://en.wikipedia.org/wiki/Quadtree
[15] http://forum.openframeworks.cc/t/quadtree-particle-system

Chapter 5

CREA: SYSTEM PROTOTYPE

The two previous chapters have explained the motion capture and the graphics generation parts of the system. This chapter includes the requirements used to develop the overall system, then it goes over the system architecture, the implementation of all the other relevant components not yet explained in the previous chapters and it finishes with a short conclusion.

5.1 Introduction

The motion capture and generation graphics algorithms used in *CREA* have been already discussed in the previous chapters but there were many other things that had to be done to develop the overall system and to make it a useful tool in a live performance setting.

Figure 5.1 summarizes the functional blocks of the overall system. Apart from the motion capture and graphics generation, we created a graphical user interface to configure the parameters to be used in a particular performance and to control the behavior of the system during the actual performance. In order to set up the set of graphics that will be used in the performance we created the Cue List module. This allows the user of the system to save a sequence of graphic settings and switch between them with a smooth transition. The system also integrates preliminary research results on gesture following developed by Cheng-i Wang and Shlomo Dubnov ([Wang and Dubnov, 2014]). This tool allows us to recognize patterns from a prerecorded sequence and use the output to change the graphic settings automatically.

The project has been developed under the openFrameworks toolkit[1]. Its addons and its community have been an essential part to fulfill the requirements of this project. Also thanks to other open source projects I have been able to learn best programming practices and use existing code to be able to make the tool more interesting. Because of that I decided to use an open source license for my project.

In order to manage the software development process of the project and have control over the different versions of the software I used the GitHub platform. This platform allowed me to share the code with other people and at the same time work collaboratively to implement the Gesture Follower feature (see section 5.6). Moreover, since I wanted *CREA* to be an open source project, GitHub allowed me to publicly publish the code under an open source license. I chose to license the code under the GNU Affero License Version 3, which is a copyleft license[2].

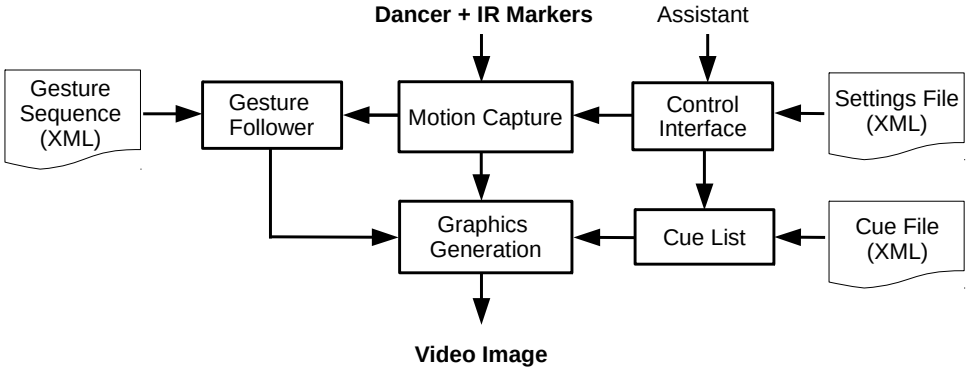


Figure 5.1: Block diagram of the *CREA* system

5.2 Requirements

The requirements of the overall system are a consequence of the user scenario that we defined and were elaborated at the beginning of the project while talking with possible users and my advisors. The specific requirements for the motion capture module and the graphics generation have been already explained in the previous chapters. Below we list the general requirements that we defined. First we men-

[1] http://openframeworks.cc/

[2] http://www.gnu.org/licenses/agpl-3.0.en.html

tion the functional requirements and then we continue with the non-functional ones.

5.2.1 Functional Requirements

The system must fulfill the following functional requirements:

1. **Motion Capture**
   CREA must have a motion capture module from which get the motion data from the users movement using the Microsoft Kinect depth sensor.

2. **Graphics Generation**
   CREA must have a graphics generation module that creates different graphics based on the motion data.

3. **Control Interface**
   CREA must have a control interface to have control over the motion capture and graphics generation modules. This control interface must allow to configure the parameters to be used in a particular performance and to control the behavior of the system during the actual performance.

4. **Gesture Follower**
   CREA must have the elements necessary to integrate a gesture following algorithm developed by Cheng-i Wang and Shlomo Dubnov.

5.2.2 Non-functional Requirements

The non-functional requirements that the graphics system should fulfill:

1. **Open source**
   CREA source code shall be publicly available online under an open source license.

2. **Real-time**
   CREA shall be computationally efficient and fast so it can run in real-time.

3. **Platform compatibility**
   CREA shall work in Windows, Linux and OS X systems.

4. **Implementation**
   CREA has to be written under the openFrameworks toolkit using the C++ programming language.

5. **Extensibility**
   CREA shall be easy to extend in order to add new functionalities. In order to do that it should also include documentation so other users can understand and use all the current features.

6. **User friendly**
   CREA shall be easy to learn even to novice technology users.

7. **Usability**
   CREA shall allow a choreographer to prepare a dance performance offline and have control over the system during the live performance.

8. **Crash recovery**
   CREA shall be capable to restart fast in case of unexpected crashes and errors.

9. **Portability**
   CREA shall be able to adapt to different physical environments, with different lighting conditions and setups.

10. **Price**
    The hardware and system on top of which *CREA* is used shall be affordable.

5.3 Overall System Architecture

Having the list of functionalities that we wanted the system to have we conducted an object-oriented design of the software to decide the different objects and components required to meet all the requirements. Figure 5.4 shows the result of this process at a high-level of abstraction without much details. The figure shows how the different classes/objects interrelate and interact.

We describe below the main classes that conform *CREA*:

ofApp Contains instances to all the other classes and manages their behavior by means of the control interface.

irMarker Defines the *IR Markers* objects that we track from the infrared data stream, containing the position and velocity information.

Contour Contains all the functionalities that come from the depth map stream. This includes the computation of the optical flow (see section 3.6), the motion contours using velocity mask (see see section 3.7) and the different ways of representing the *silhouette contour* (see section 4.5).

Fluid Describes all the functionalities related to the fluid solver and fluid simulation (see 4.3). Solves the equations of fluid dynamics to simulate a two-dimensional fluid behavior and contains the methods to use the state of the fluid to interact with other objects.

ParticleSystem Describes a particle system. Contains a dynamic number of particles and defines the set of rules they have to follow to simulate various behaviors.

Particle Describes a single particle. Contains the basic functionality to make this particle move around the screen and update its motion by means of applying external forces.

Sequence Describes the sequence of the positions of the *IR Markers* in a period of time (gesture).

vmo Contains the functionality to follow gestures from a previously recorded sequence of *IR Markers* positions.

Following the structure of every openFrameworks project, there is always three basic files (main.cpp, ofApp.cpp and ofApp.h) that contain the core structure to run the project and create a window for visual output. In the 'ofApp' class there is a 'setup', 'update' and 'draw' methods. Very briefly explained, 'setup' is where we initialize all the variables and parameters of the application, since it is called just once at the start of the project. Right after 'setup' there is 'update', which is where we perform all the computations to update the state of our application in an infinite cycle that is called at every frame until we finish the application. Finally there is 'draw', which is called right after 'update' and it performs all the drawing functions to visualize the application in the screen. Because of this, it is in this class where we have instances to all the other classes and manage the project behavior by means of the control interface.

5.4 Control Interface

The control interface had to be easy to understand and user-friendly, with a balance between simplicity and usability; a novice user would have to be able to use the basic features but at the same time a curious and experienced user should have the options enough to customize better the graphics and control more advanced settings.

After looking over all the different GUI addons created for openFrameworks we

decided to use ofxUI[3]. This addon is easy to use, very fast, with minimalist design, beautiful aesthetics and very presentable to novice technology users. Moreover, this includes a saving/loading functionality of the widget values, which we required in order to be able to save and load the settings of the control interface.

Having decided all the requirements of the system we designed the control interface in a way that all the functionalities would be separated in different panels, each with a very clear purpose. We identified 14 different elements for which we would require independent interface controls:

Basics General functions of the control interface and *CREA* (i.e., save/load control interface settings, change output window background color, number of frames to transition between graphic scenes...)

Kinect All parameters that control how we process the data of the Kinect input streams. These should allow us to calibrate the Kinect to a particular physical space (see section 3.3).

Gesture Follower Settings to control the gesture follower (i.e., record sequence of *IR Markers*, start/stop gesture tracking, mapping of the recognized gestures to graphic outputs... see section 5.6).

Cue List Sequence of graphic settings that allows us to switch from one to the other in a smooth transition (see section 5.5).

Optical Flow Settings to control the optical flow output (i.e., strength of the output velocities, blur velocities over time, threshold to consider motion... see section 3.6).

Velocity Mask Settings to control the velocity mask output (see section 3.7).

Fluid Solver Settings to control the fluid properties (i.e., viscosity, vorticity, dissipation... see section 4.3).

Fluid Simulation Settings to control the color of the fluid simulation and how we inject new values to the fluid solver (see section 4.3.2).

Fluid Particles Properties of the particles based on the fluid flow from the fluid solver (mass, size, number of particles... see section 4.4.8).

Silhouette Contour Different ways of representing the *silhouette contour* plus a few settings to modify this representation (see section 4.5).

[3]https://github.com/rezaali/ofxUI

Emitter Particles Settings to control Emitter Particles behavior (see section 4.4.4).

Grid Particles Settings to control Grid Particles behavior (see section 4.4.5).

Boids Particles Settings to control Boids Particles behavior (see section 4.4.6).

Animations Particles Settings to control Animations Particles behavior (see section 4.4.7).

Having all these elements, we defined the most relevant controls for each of them, thinking on which values made more sense to be able to manipulate. Next we had to define which type of GUI widgets use for each of the controls (i.e., toggle, button, slider, text box...) and plan how they would react to the system. The next step was to structure all the controls around the window in a way that was as clear and simple as possible, adding all the elements that we thought were useful to understand the current status of the system. At the end we decided to group the 14 panels we have mentioned in 8 different menus (see Appendix A).

In order to save/load the settings of the control interface we defined our own format using XML, taking advantage of the XML functionalities that come with the *ofxUI* addon. In this way we were able to decide which widget values we wanted to save in the file and which not. Furthermore, we defined two different ways of loading the settings files: (1) change all the current values to the ones in the file immediately or (2) gradually interpolate between the current values and the ones in the file within a determined number of frames (see 'saveGUISettings' and 'loadGUISettings' methods inside ofApp.cpp file for more details).

Implementing (2) was a bit more complicated. The approach we use is to save the original value and the goal value for each of the widgets and use the number of frames elapsed since we loaded the settings file to compute the current value. In order to do that we created a map container with a pointer to each widget as key and a vector of floats as the mapped value for each key. Then we just store the initial value in the first position of the vector and the goal value to the second, interpolating these two values at every frame in order to get the current value (see 'interpolateWidgetValues' method inside ofApp.cpp file for more details).

5.5 Cue List

Cue List is a simple feature we created to save different graphic settings (cues) in a list and be able to switch from one to the next in a smooth transition (see Figure

5.2). This includes some basic functionalities to manage the list, such as creating new cues, delete them, put a name to the cue and save/load cues as XML files. In this way we can design the sequence of graphics we want to have for a particular performance.

The save/load approach is very similar to the one used for the control interface settings (in fact we use the same methods); the only difference is that for the cues we only save and load the settings values belonging to the graphics.

We created a trigger button ('GO') that when is pressed loads the next cue in the list by interpolating the settings values, as explained before. Moreover, we created some methods inside each of the classes containing output graphics (ParticleSystem, Contour and Fluid) to do a fade in/out of the drawing opacity when they get activated/deactivated.

Figure 5.2: Cue List panel in the control interface.

5.6 Gesture Follower

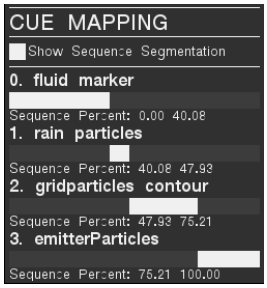
As said in Chapter 1, one of the the goals of the project was to combine *CREA* with a prototype of a gesture tracking technology implemented by Cheng-i Wang and Shlomo Dubnov. The idea of this tool is to give the performer the freedom to improvise from a set of recorded gestures and have the system dynamically change based on his movements.

[Wang and Dubnov, 2014] introduced a data structure (VMO) capable of identifying repetitive fragments and finding sequential similarities between observations. In [Dubnov et al., 2014] they used these properties for tracking and recognizing gestures in real-time, using the positions of the *IR Markers* as a representation of a gesture (sequence). First they record a long sequence with the intended gestures to be recognized and then using the 'VMO' they are able to recognize sub-portions (gestures) of the piece. In [Wang and Dubnov, 2015] this gesture tracking was improved by automatically capturing repeated sub-segments from the long sequence with no manual interaction.

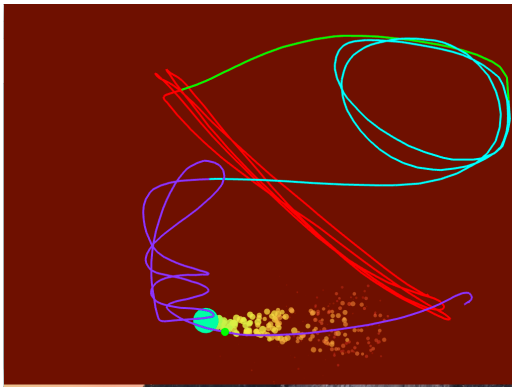
This technology had to be implemented using C++ within the openFrameworks

environment in order to be used from *CREA*. Moreover, in *CREA* we had to implement several features to be able to use the gesture follower. We explain very briefly some of these features.

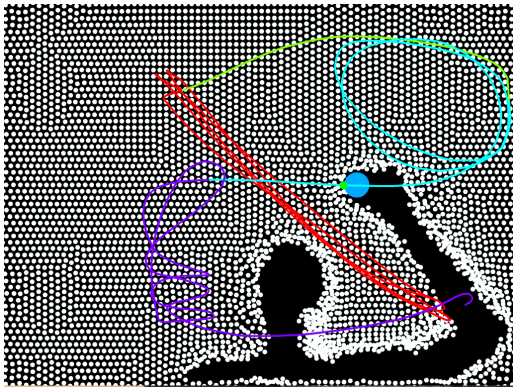
One of the most important things to do was save the gestures done with the markers (sequence of its position over time) in a way that could be used to feed the gesture follower. We created our file format for saving this sequence of positions in an XML file. Since there can be more than one marker in one same frame and we can loose track of them, we had to take that into consideration when writing the marker information over time. Besides this, we had to implement the method that could read these files in a robust way and store the data in memory so the gesture follower could later use it.



(a) Cue Mapping



(b) Cue with Emitter particles



(c) Cue with Grid particles

Figure 5.3: Snapshots of using the Gesture Follower in *CREA*. The colored line represents the stored sequence, with each color belonging to different graphics (cues). Since we have four cues in the Cue List we can see four different colors.

Besides this, we had to implement several functionalities to show the output of the tracking in real-time and devise a way to map this output to the graphics during a performance. In Figure 5.3 the colored line represents the stored gesture (sequence of 2D points from one *IR Marker*) with different colors indicating mappings to different graphics (cues). The bigger circle is the input marker and the

green small circle along the trajectory is the current gesture-following position on the stored gesture corresponding to the input marker. As we can see, the system generates different graphics when the gesture following position (green circle) traverses onto different segments of the stored gesture represented by different colors. These segments can be defined through the control interface by saying which percentage from the long sequence assign to each of the cues in the Cue List (see Figure 5.3a).

5.7 Conclusions

In this chapter we have presented all the components of *CREA* that were not yet explained in the previous chapters. The Gesture Follower feature is still under a research phase and that is why we didn't enter in much detail. In terms of the use of the control interface you can find more details in the appendix A.

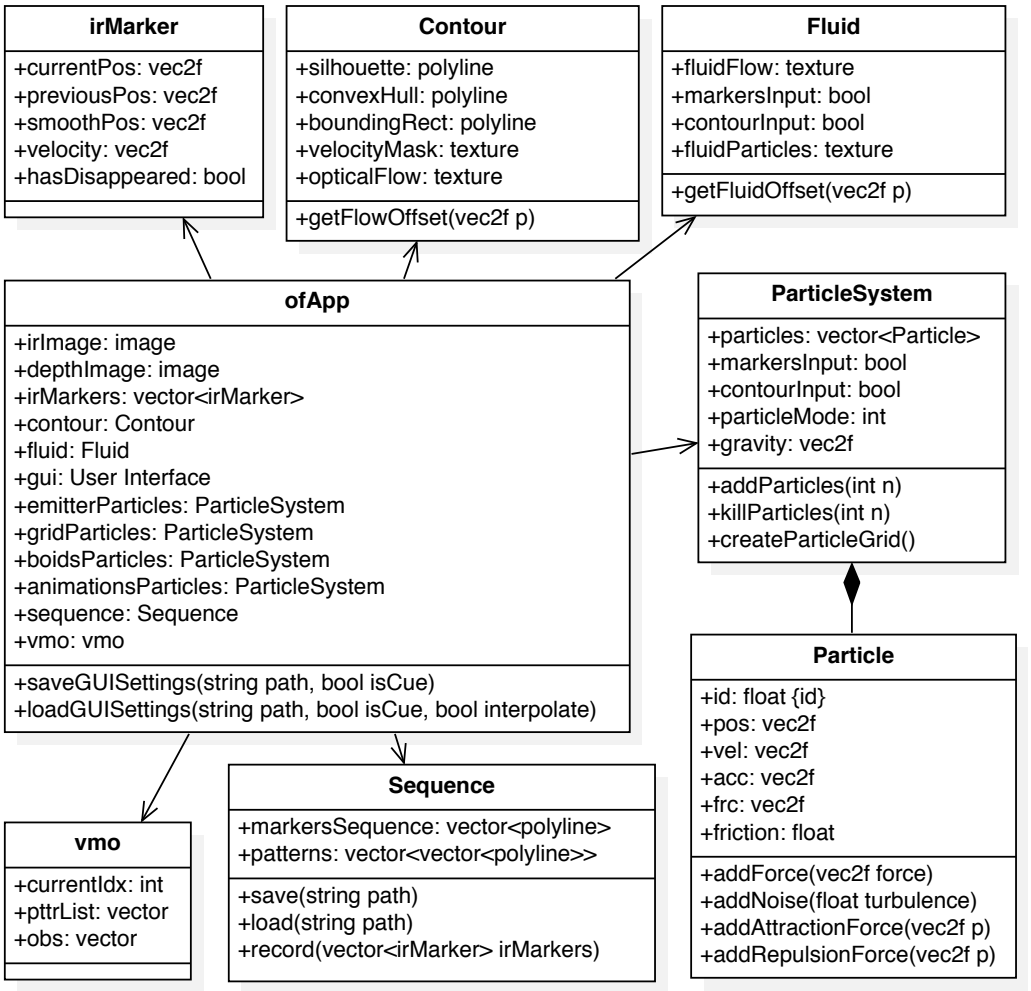


Figure 5.4: *CREA* UML Class Diagram with the basic class functionalities.

Chapter 6

EVALUATION AND TESTING

This chapter explains how *CREA*, the system developed as part of this project, was tested on a real context and reports on the feedback obtained from these testings.

6.1 Introduction

As it has been explained before, *CREA* is an application designed to be used in an artistic context. Therefore, *CREA* had to be tested emulating a real performance with people experienced in the field in order to evaluate wether it met all the requirements defined in section 5.2.

The development of *CREA* has been done through a sequence of constant and iterative evaluations. As soon as we had a working prototype, we tested its functionalities emulating as close as possible a real performance context. Thanks to a dancer close to the project we were able to emulate better real case scenarios and adapt the system to be useful within that context.

Throughout the last two months of the development of *CREA* this has been tested a total of four times in a performance context, not only with the mentioned dancer but also with other dancers and regular users. We have set up the system in three different spaces and this has been really helpful in evaluating the portability of the system.

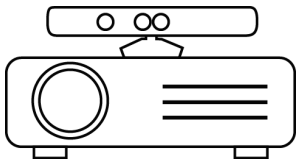
In terms of hardware, *CREA* has been tested on several computers running different operating systems. This is the list of the different system configurations where it has been tested on:

1. MacBook Pro (2015) with Intel Core i5 2.7 GHz processor and Intel Iris Graphics 6100 graphic card, with 8 GB RAM. Running OS X Yosemite.
2. MacBook Pro (2013) with Intel Core i5 2.4 GHz processor and Intel Iris 5100 graphic card, with 4 GB RAM. Running OS X Yosemite.
3. MacBook Pro (2012) with Intel Core i5 1.8 GHz processor and HD Graphics 4000 graphic card, with 4 GB RAM. Running OS X Mavericks.
4. MacBook (2009) with Intel Core 2 Duo 2.26 GHz processor and NVIDIA GeForce 9400M graphic card, with 4 GB RAM. Running OS X Mountain Lion.
5. Antec with Intel Core 2 Duo 3 GHz with NVIDIA GeForce 8800 GT with 4 GB Ram. Running Ubuntu 14.04 and Windows 8.1.

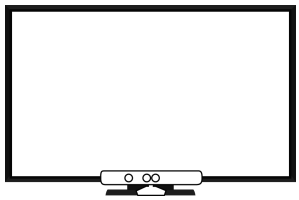
The performance of *CREA* was good on all these computers, obtaining frame rates between 20 and 30 fps in all of them. Moreover, the feeling of real-time responsiveness was reported adequate by both the dancers and the other users of the system.

6.2 User Testing of the Prototype

CREA has been tested a total of four times in front of different audiences, where we have extracted various feedback that has been really helpful in the fulfillment of the requirements. We used three different spaces for these evaluations. In two of these spaces we used a front-projection approach and in the third a 60-inch plasma display. The latter case was not meant for evaluating the system within a dance performance context but as a showcase to the project; the users could try the system by looking their movements reproduced in the display screen and see the resulting graphics.



(a) Kinect-Projector setup



(b) Display screen setup.

Figure 6.1: Illustration of the setups used in the evaluations.

In the two physical spaces where we used a projector, since we haven't developed any Projector-Camera Calibration to align the projected image to the physical space, we placed the Kinect on top of the projector and adjusted both manually in order to match the users with its projected image (see Figure 6.1a). In the case where we

used the plasma display, we placed the Kinect below the screen facing the users (see Figure 6.1b).

The first two times we tested the system, both in the same performance space, we used a front-projection approach and the computer 2. By that time there were still many things that were not yet implemented in *CREA*, making it a bit more difficult to evaluate the results. However, there were already many things that we could tell that needed to improve or change.

One example is on the motion capture procedure. The space used in these first tests was a bit small and there were always some objects in the interaction area that couldn't be removed just by filtering and thresholding the input image (either we would get the ceiling, the walls or some fixed speakers in front of the scenario). For this reason the system generated graphics on these areas. The solution was to implement some simple methods to crop the interactive area. With these methods, even though the interaction space becomes smaller, we are able to eliminate the undesired surfaces from the field of view (see section 3.3). Moreover, thanks to this first test we realized of the importance of using two thresholds when extracting the silhouette data from the depth map (see section 3.3), being able to delete all the surfaces closer to a certain distance (before this first test we were only using one threshold).

Thanks to the second test in the same space we could try these functionalities and evaluate their performance. Two videos in YouTube[1,2] show a little bit the results of these two early tests. In these two tests I wasn't present and hence I was not able to explain to the people controlling *CREA* how it worked or which graphics behaviors were intended to be used (some of the graphics shown in the video were only for debugging purposes). However, they were able to understand the basic functionality and use it. This was also really helpful in getting feedback about the control interface and which things were more confusing.

The third time we tested the system we used the plasma display setup and the computer 3. This test was a showcase for the people to get to know about the project. By this time the system was more advanced and the Optical Flow (see section 3.6) was already implemented, having two very distinct ways of interacting with the graphics (*IR Markers* and *silhouette contour*). One very common problem among the people who had their first impression with the software (without seeing anyone use it before or explaining them the basic functionality) was that they tried to use the *IR Markers* to interact with the graphics all the time, even when the graphics were meant to be interacted by moving the whole body (*silhouette contour*). That

[1] https://www.youtube.com/watch?v=4Uvh9kaXI78
[2] https://www.youtube.com/watch?v=AG7Yxm1B0Mo

was a totally normal behavior since at first glance, if you get an object (marker) before using an interactive application, the first think you imagine is that you have to use this object to interact with the system. However, this is something that only happened when the people used the software without any prior information, which is something that in our context shouldn't happen very often. During the test we also encouraged the audience to try to use the control interface and change the graphics behavior themselves. Thanks to this, we were able to realize further improvements in the interface.

The fourth time we evaluated *CREA* was during a party on which people would use it as an entertainment attraction[3]. By this time the system already implemented most of the features we initially planned. The space used in this case was also rather small and we had to use a front-projection approach so that people could dance. We used the computer 1. The party lasted for four hours and the system was running non-stop, being able to test how the system reacted when being used during a long period of time. Thanks to this, we were able to recognize a few unexpected behaviors from the graphics and a couple of bugs that were making the fluid solver stop after a long time running. Again, we invited the people to control the system and change the settings to alter the output graphics and behaviors.

6.3 Discussion

All the users that tried the software, specially in the third and fourth evaluations, were really excited with the system and they liked it very much. There was a difference between the feedback obtained from the dancers (expert users) and the one obtained from regular users. The dancers saw the potential of the system as a way to enhance their performances and create new ways of expression. Moreover, they said the graphics helped them in exploring new dance movements and make them feel more emotive. On the other hand, regular users emphasized the fact that the system was very engaging and fun, that promoted dancing and the interaction with other users.

A negative feedback from both users was that they didn't like the casted shadows from the performers on the display surface and the fact that the graphics were projected onto them; they said it was sometimes distracting and reduced the sensation of immersion. This is the most common problem with interactive displays based on front projection. In order to solve this issue the easiest solution is to

[3] https://www.youtube.com/watch?v=FvD1yGWRt6w&feature=youtu.be

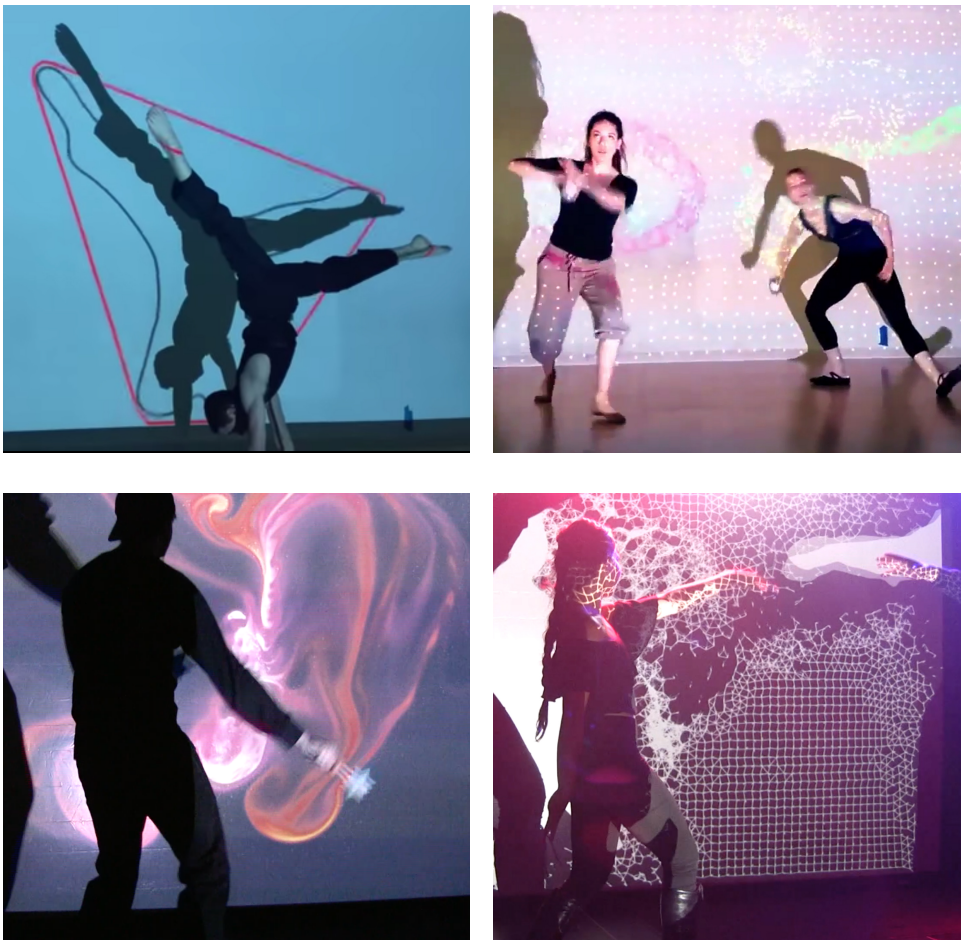


Figure 6.2: Pictures from the evaluations of *CREA*.

use a rear-projection system or non projected display technologies such as LCD and plasma display panels (as we did in the third evaluation). The problem with these methods is that they can be really expensive and hard to set up. In order to use rear-projection we need a large space because the projector requires to be placed some distance behind the screen. Moreover, the display has to be made from a special transparent material that allows the image to pass through without loss of quality. Because of this, front projection is always cheaper and increases portability. [Summet et al., 2007] describes various workarounds on mitigating the problems of the shadows when using front-projection, ranging from simple solutions to fully reactive systems. The simplest solutions are based on shifting the shadows away from the important areas of the display surface. This is done by moving the projector to a side and warping the image. The most complex alter-

natives modify the projected image by observing the output screen with an extra camera, detecting where the users are occluding the light from the projector and dynamically adjusting the amount of illumination to eliminate the shadows.

One thing that surprised me greatly in all the experiments was how the people (even though they were not aware of what was the behavior of the graphics being generated) quickly understood how to interact with the graphics and they were able to create very interesting effects and situations I couldn't have ever imagined before.

The original idea of the project was to design a choreography that would exploit the potential of the graphics and put together a performance using that. This would have been the best way to evaluate *CREA* but it has not been possible and we hope to be able to do it in the near future. However, the evaluation with users has shown that the interaction is very smooth and that it creates a very natural dialogue between the performing users and the graphics, proving that it can be used in a real performance context. Moreover, the tests have shown that the control interface works well and that is user-friendly (i.e., even non-tech users were able to use it). Because of these reasons, I consider the system fulfills satisfactorily all the requirements that we defined.

Chapter 7

CONCLUSIONS

The goal of this project has been to create a system that generates graphics in real-time using motion capture as input and that can be used in an artistic context.

As part of the project we have studied various algorithms and practices to best extract motion data from the Microsoft Kinect input streams. We have implemented these algorithms and we are able to extract information on how the performers move around the interaction space in real-time.

We have also developed several graphic generation algorithms that respond to the motion capture data and are efficient enough to work in real-time. These graphics include particle systems with different behaviors, abstract representation of the silhouette of the dancers and two dimensional fluids simulation.

Finally we have implemented a control interface that makes it easy to adapt the motion capture to different physical spaces, configure the settings of the graphics and manage the overall behavior of the system in a live performance.

The results obtained show that all the proposed objectives have been accomplished. The evaluation with users has shown that the interaction is very smooth and that it creates a very natural dialogue between the performing users and the graphics, proving that it can be used in a real performance context. The resulting graphics work as a partner of the dancers in making his movements feel more emotive and not just a mere decoration. The graphics respond to the movements of the users in an intuitive way and even when the dancers are not aware of the outcome of the graphics they are able to interact with them naturally. The graphics help the dancers also in exploring new movements and create new ways of expression.

The current state of the art in this type of tools is very advanced and there are many professional systems with a lot of functionalities. Many interactive artworks tend to be designed by a group of people that work in close collaboration, most of the times with a specific goal to create a unique piece. In the case of dance projects there is normally a choreographer and a digital artist who devise the performance and the visual graphics interactions in a single direction, with a specific visual and structural approach and a defined system/dancer dialogue.

This project cannot compete with the professional systems that exist. It has been mainly a personal project in which I had to take most of the roles myself and although I have tried my best, the results would have definitely been better if there had been a very well defined approach by someone experienced in the field. However, I believe my project is a good starting point from which I have learned a lot throughout the whole process and after which I feel much more confident to be involved in more ambitious and professional projects in the future.

7.1 Future work

From the beginning the idea of this project was to build a software framework that would include some specific functionalities but at the same time that would be easy to extend and to include new features.

Some relevant improvements that could be done would be the following:

1. Add audio analysis from the music used during the performance to extract features that can be used as another input to alter the graphics behavior.
2. Use the capabilities of the GPU to accelerate the computations of the particle systems and create more complex behaviors.
3. Add more graphic generation algorithms.
4. Extend the mappings between extracted motion and the graphics generation engine. Develop automatic behaviors without the need to set them manually with the control interface.
5. Improve the Gesture Follower feature.
6. Add camera-projector calibration functionality.

Bibliography

[Abbott, 1997] Abbott, M. B. (1997). *Computational Fluid Dynamics: An Introduction for Engineers*. Longman Pub Group.

[Andersen et al., 2012] Andersen, M., Jensen, T., Lisouski, P., Mortensen, A., Hansen, M., Gregersen, T., and Ahrendt, P. (2012). Kinect depth sensor evaluation for computer vision applications. Technical report, Aarhus University, Department of Engineering.

[Borenstein, 2012] Borenstein, G. (2012). *Making Things See: 3D vision with Kinect, Processing, Arduino, and MakerBot*. Maker Media, Inc.

[Boucher, 2011] Boucher, M. (2011). Virtual dance and motion-capture. *Contemporary Aesthetics*, 9.

[Camplani et al., 2013] Camplani, M., Mantecon, T., and Salgado, L. (2013). Depth-color fusion strategy for 3-d scene modeling with kinect. *Cybernetics, IEEE Transactions on*, 43(6):1560–1571.

[Chorin and Marsden, 2000] Chorin, A. J. and Marsden, J. E. (2000). *A Mathematical Introduction to Fluid Mechanics*. Springer, 3rd edition.

[Dubnov et al., 2014] Dubnov, T., Seldess, Z., and Dubnov, S. (2014). Interactive projection for aerial dance using depth sensing camera. In *The Engineering Reality of Virtual Reality*.

[Farnebäck, 2003] Farnebäck, G. (2003). Two-frame motion estimation based on polynomial expansion. In *Image Analysis, 13th Scandinavian Conference*, 2749, pages 363–370.

[Harris, 2004] Harris, M. J. (2004). Fast fluid dynamics simulation on the gpu. In Fernando, R., editor, *GPU Gems*, chapter 38. Pearson Education, Inc.

[Karluk et al., 2013] Karluk, L., Noble, J., and Puig, J. (2013). Introducing shaders. openFrameworks Tutorials.

[Noble, 2012] Noble, J. (2012). *Programming Interactivity*. O'Reilly Media, 2 edition.

[openFrameworks community, 2015] openFrameworks community (2015). *ofBook*. openFrameworks.

[Perevalov, 2013] Perevalov, D. (2013). *Mastering openFrameworks: Creative Coding Demystified*. Packt Publishing.

[Piana et al., 2014] Piana, S., Staglianò, A., Odone, F., Verri, A., and Camurri, A. (2014). Real-time automatic emotion recognition from body gestures. *CoRR*, abs/1402.5047.

[Reynolds, 1987] Reynolds, C. W. (1987). Flocks, herds, and schools: A distributed behavioral model. In *SIGGRAPH '87 Proceedings of the 14th annual conference on Computer graphics and interactive techniques*.

[Rideout, 2010] Rideout, P. (2010). Simple fluid simulation. The Little Grasshopper, Graphics Programming Tips.

[Rost and Licea-Kane, 2009] Rost, R. J. and Licea-Kane, B. M. (2009). *OpenGL Shading Language (3rd Edition)*. Addison-Wesley Professiona.

[Shiffman, 2012] Shiffman, D. (2012). *The Nature of Code: Simulating Natural Systems with Processing*. The Nature of Code.

[Sklansky, 1982] Sklansky, J. (1982). Finding the convex hull of a simple polygon. *Pattern Recognition Letters*, 1(2):79–83.

[Stam, 1999] Stam, J. (1999). Stable fluids. In *SIGGRAPH 99 Conference Proceedings*.

[Stam, 2003] Stam, J. (2003). Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*.

[Summet et al., 2007] Summet, J., Flagg, M., jen CHam, T., Rehg, J. M., and Sukthankar, R. (2007). Shadow elimination and blinding light supression for interactive projected displays. *IEEE Transactions on Visualization and Computer Graphics, Accepted*, 13(3).

[Suzuki and Abe, 1985] Suzuki, S. and Abe, K. (1985). Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1):32–46.

[Vivo, 2015] Vivo, P. G. (2015). *The Book of Shaders*. Patricio Gonzalez Vivo.

[Wang and Dubnov, 2014] Wang, C.-i. and Dubnov, S. (2014). Variable markov oracle: A novel sequential data points clustering algorithm with application

to 3d gesture query-matching. *Multimedia (ISM), 2014 IEEE International Symposium on*, pages 215 – 222.

[Wang and Dubnov, 2015] Wang, C.-i. and Dubnov, T. (2015). Free-body gesture tracking and augmented reality improvisation for floor and aerial dance. *SPIE Electronic Imaging 2015*.

[West, 2008] West, M. (2008). Practical fluid dynamics. *Game Developer magazine*.

Appendix A

CREA USER'S GUIDE

This appendix is the user guide of *CREA*, explains how to install *CREA* and gives an overview of the control interface.

A.1 Requirements

To use *CREA* you will need:

1. A Windows, Mac OS X or Linux computer with modern graphics.
2. Xbox Kinect Sensor, preferably model 1414, with USB adapter and Power Supply.
3. Projector[1] or LCD screen.
4. *CREA* software.
5. openFrameworks 0.8.4.

A.2 Installation

1. Download openFrameworks in your computer (http://openframeworks.cc/download/). Extract the folder you downloaded and put it somewhere convenient.

[1] http://www.creativeapplications.net/tutorials/guide-to-projectors-for-interactive-installations/

2. Download or clone the repository of *CREA* (https://github.com/fabiaserra/crea) and place it inside the openFrameworks folder, in **apps/myApps**. You should have a folder named **crea** with two folders inside, **src** and **bin**, and a few more files. The **src** folder contains C++ source codes for your project. The **bin** folder contains the **data** subfolder where all the content files needed for *CREA* are located:

   (a) cues: Cue XML files (see section 5.5).

   (b) fonts: Font files.

   (c) kinect: Sequence of images to use *CREA* without plugging a Kinect.

   (d) sequences: *IR Markers* sequence XML files.

   (e) settings: Settings XML files of the control interface.

   (f) songs: Audio files to play with the basic audio player.

   The **bin** folder contains also an executable file of *CREA* after you compile it. If this file has the '_debug' suffix means that the project was compiled in the Debug mode of compilation, which lets you debug the project by using breakpoints and other debugging tools. Projects compiled in the Debug mode can work very slowly so always compile it in the Release mode if you use *CREA* in a live performance.

3. Download the list of required addons and place them inside the **addons** folder:

   (a) ofxKinect: https://github.com/ofTheo/ofxKinect.

   (b) ofxCv: https://github.com/kylemcdonald/ofxCv

   (c) ofxFlowTools: https://github.com/moostrik/ofxFlowTools

   (d) ofxUI: https://github.com/rezaali/ofxUI

   (e) ofxSecondWindow: https://github.com/genekogan/ofxSecondWindow

4. After downloading ofxFlowTools, open the **ftVelocityMask.h** file (inside the **src/mask** folder) and add the following lines of code:

```
float getStrength() {return strength.get();} // added
int   getBlurPasses() {return blurPasses.get();}
float getBlurRadius() {return blurRadius.get();}
void  setStrength(float value) {strength.set(value);} // added
void  setBlurPasses(int value) {blurPasses.set(value);}
void  setBlurRadius(float value) {blurRadius.set(value);}
```

5. Create a new openFrameworks project using the project generator tool that is inside the openFrameworks root folder[2]. Name it 'crea' (same name as repository folder you downloaded before) and make sure all these addons are enabled in the list of addons shown: **ofxKinect**, **ofxCv**, **ofxOpenCV**, **ofxFlowTools**, **ofxUI**, **ofxXmlSettings** and **ofxSecondWindow**.

6. Go to **apps/myApps** and inside the folder **crea** you should see a new file. This file is the project file and depending on your development environment it has extension .sln (Visual Studio), .xcodeproj (Xcode), or .workspace (Code::Blocks).

7. Open the file with your development environment.

8. Inside ofApp.h there are a set of macros to change some basic features that are set by default.

   If we want to use a separate window for the control interface this line of code has to be uncommented:

```
// Use a separate window for control interface
#define SECOND_WINDOW
```

   With the following lines of code we set the resolution of the projector:

```
#define PROJECTOR_RESOLUTION_X 1680
#define PROJECTOR_RESOLUTION_Y 1050
```

   In order to use the Kinect live input the following line has to be uncommented. Otherwise we will use the sequence of images saved in the folder 'bin/data/kinect' as input:

```
// Use the Kinect live input stream
#define KINECT_CONNECTED
```

   In order to be able to use the Gesture Follower feature this line has to be uncommented:

```
// include gesture follower files (you need to have vmo.cpp, vmo.h,
    helper.cpp and helper.h in src)
#define GESTURE_FOLLOWER
```

   In order to use an offline sequence of *IR Markers* as input to the Gesture Follower this line has to be uncommented:

```
// Use an xml IR Markers sequence file as input to Gesture Follower
#define KINECT_SEQUENCE
```

9. Compile the project and enjoy *CREA*.

[2] http://www.openframeworks.cc/tutorials/introduction/002_projectGenerator.html

A.3 MAIN MENU

The interface is divided in eight different panels (see Figure A.1a), which are shown all the time in the left hand side of the window. We can open a panel and access to its functionalities by pressing its corresponding number (from 1 to 8) with the keyboard.

A.4 BASICS + KINECT

This panel (see A.1b) contains the general settings and the parameters to adjust the Kinect camera input.

A.4.1 BASICS

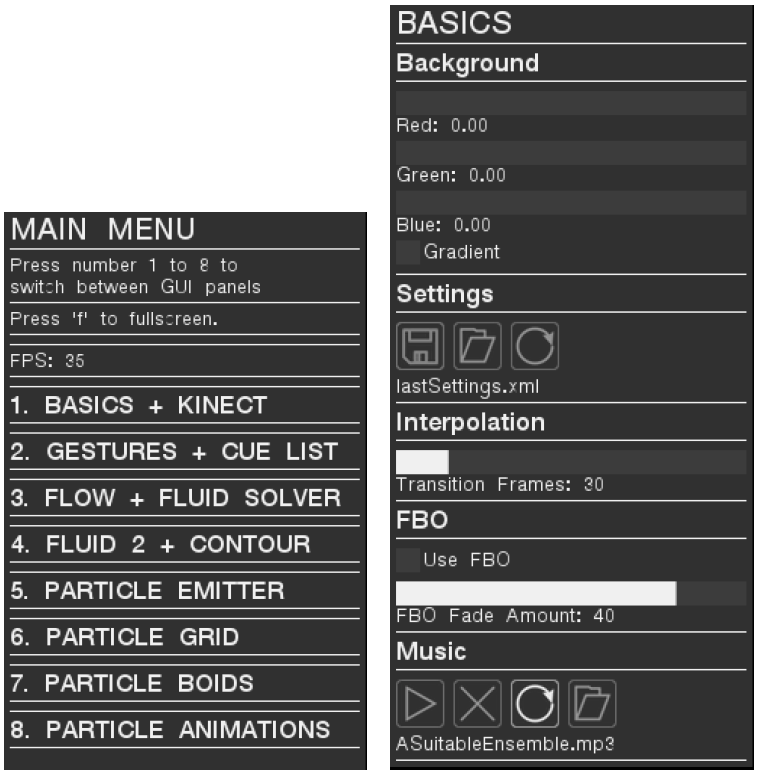
Background
Changes the background color of the output window.
With the 'Gradient' toggle we select if we want a gradient fading from center to the sides.

Settings
Save, Load and Reset the control interface settings using XML files. By default, whenever we close the application the last settings are saved in a file inside bin/data/settings (lastSettings.xml). This file is loaded when we open the application again.

Interpolation
Number of frames of interpolation between cues (see section 5.5).

FBO
Activate Frame Buffer Object to save previous rendered frames and fade them with the actual.

Music
Simple audio player. We can load a sound file from the system, play it, pause it and choose if we want to loop it.



(a) Main Menu **(b)** Basics

Figure A.1: Main menu and basics panel

A.4.2 KINECT

Up and Down Keys: Sets the tilt angle of the Kinect to define the interaction space.

Reset Kinect: Reinitializes the Kinect.

Flip Kinect: Flip the image of the Kinect.

Depth Image

Clipping Range: Sets the distance of the near and far clipping planes of the depth sensor, ignoring whatever is outside this range and having the Kinect use all the 255 levels of depth (8 bits) within only this range.

Threshold Range: Sets the highest and lowest value to threshold the depth image. Anything lower or higher than the threshold range will be black.

Contour Size: Sets the minimum and maximum size of the blobs to be

considered. The values define the radius of a circle in pixels. The area of the blob has to be in between the area of the smallest circle (lowest value) and the area of the biggest circle (highest value).

Left/Right Crop: Crop the depth image from left to right.

Top/Bottom Crop: Crop the depth image from top to bottom.

Depth Number of Erosions: Number of times we apply erosion operator to depth image.

Depth Number of Dilations: Number of times we apply dilation operator to depth image.

Depth Blur Size: Size of the blur filter we apply to the depth image.

Infrared Image

IR Threshold: Sets the threshold value of the IR image. Anything lower than the threshold value will be black.

Markers Size: Sets the minimum and maximum size of the blobs to be considered. The values define the radius of a circle in pixels. The area of the blob has to be in between the area of the smallest circle (lowest value) and the area of the biggest circle (highest value).

Tracker persistence: Determines how many frames an object can last without being seen until the tracker forgets about it

Tracker max distance: determines how far an object can move until the tracker considers it a new object.

IR Left/Right Crop: Crop the IR image from left to right.

IR Top/Bottom Crop: Crop the IR image from top to bottom.

IR Number of Erosions: Number of times we apply erosion operator to IR image.

IR Number of Dilations: Number of times we apply dilation operator to IR image.

IR Blur Size: Size of the blur filter we apply to the IR image.

Debugging

Show Markers: Show a color circle with a label identifier for each marker being tracked.

Show Markers Path: Show the path of the marker being tracked. If the loose track of the marker during some frames ('persistence' frames) the path will disappear.

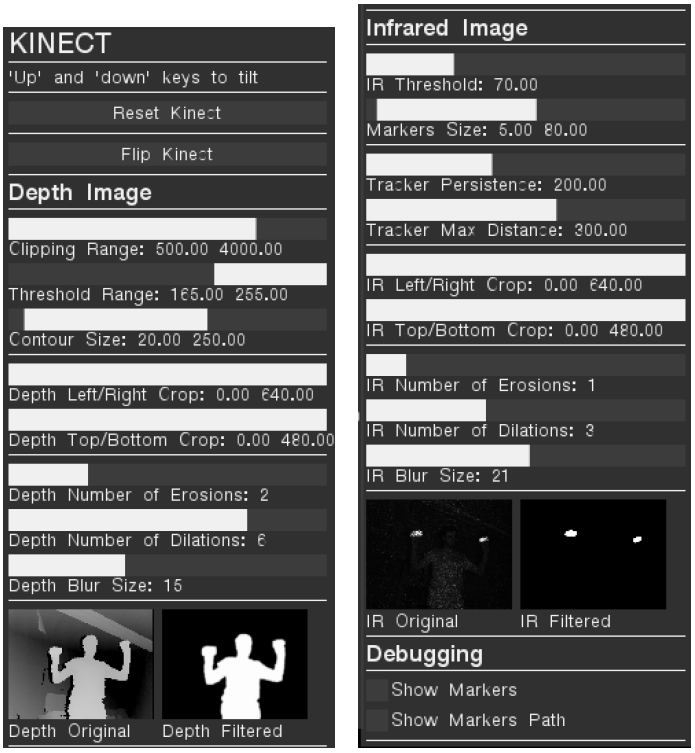


Figure A.2: Microsoft Kinect panels

A.5 GESTURES + CUE LIST

This panel contains the functionalities to record sequences of the *IR Markers*, the real-time Gesture Follower and the Cue List (see section 5.5).

A.5.1 GESTURE SEQUENCE

Record icon: Records *IR Markers* positions sequence.

Disk icon: Save sequence as an xml file.

Folder icon: Load a sequence from xml sequence file.

Arrow icon: Play loaded sequence.

A.5.2 CUE MAPPING

Show Sequence Segmentation: Draw sequence segments in different colors.

List of Cues: Assign Cue to a range of the sequence in percentage (0 to 100).

A.5.3 GESTURE FOLLOWER

Arrow icon: Start Gesture Follower.

Cross icon: Stop Gesture Follower.

Decay: Sets decay of VMO.

Slide: Sets slide of lowpass filter we apply to sequence positions.

Show Patterns inside sequence: Draw extracted patterns from gesture follower inside the input sequence.

Show Patterns: Draw extracted patterns from gesture follower in the control interface inside independent boxes.

A.5.4 CUE LIST

Plus sign icon: Add a new Cue.

File icon: Save Cue as an xml file.

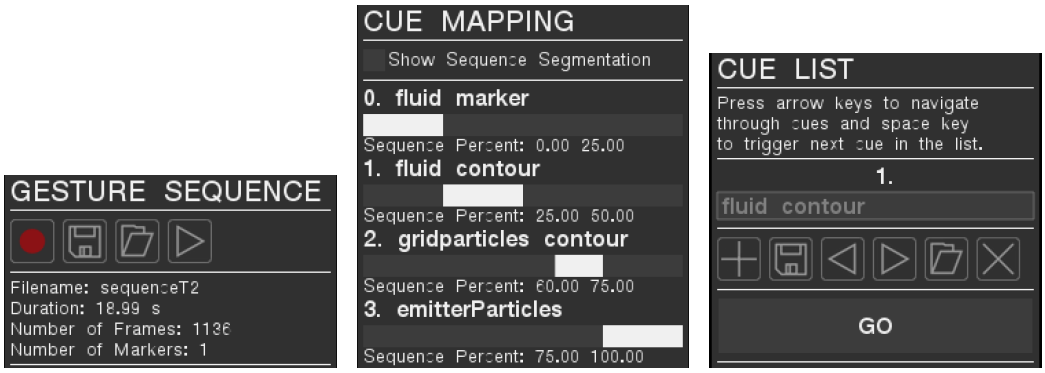
Left arrow icon: Go to previous Cue in the list.

Right arrow icon: Go to next Cue in the list.

Folder icon: Load Cue file.

Cross icon: Delete current Cue.

GO: Go to next Cue in the list interpolating the graphics settings.



(a) Gesture Sequence **(b)** Cue Mapping **(c)** Cue List

Figure A.3: Gestures and Cue List panels

A.6 FLOW + FLUID SOLVER

This panel contains the control of the Optical Flow (see section 3.6), the Fluid Solver (see section 4.3) and the control of the fluid simulation from the *silhouette contour* (see section 4.3.2).

A.6.1 OPTICAL FLOW

Eye icon: Activate/Deactivate Optical Flow.

Flow Strength: Strength scale factor of the optical flow.

Threshold: Threshold to consider if there is motion or not.

Inverse X: Inverses the direction of the horizontal component of the vector field.

Inverse Y: Inverses the direction of the vertical component of the vector field.

Time Blur Active: Activates blurring of the vector field over time.

Time Blur Decay: Time decay of the blur.

Time Blur Radius: Radius size of the blur.

Debug

Show Optical Flow: Draw optical flow vector field.

Show Scalar: Draw optical flow vector field as scalar. The red channel contains the magnitude in x and the green channel the magnitude in y.

A.6.2 VELOCITY MASK

Mask Strength: Strength of the velocity mask (alpha).

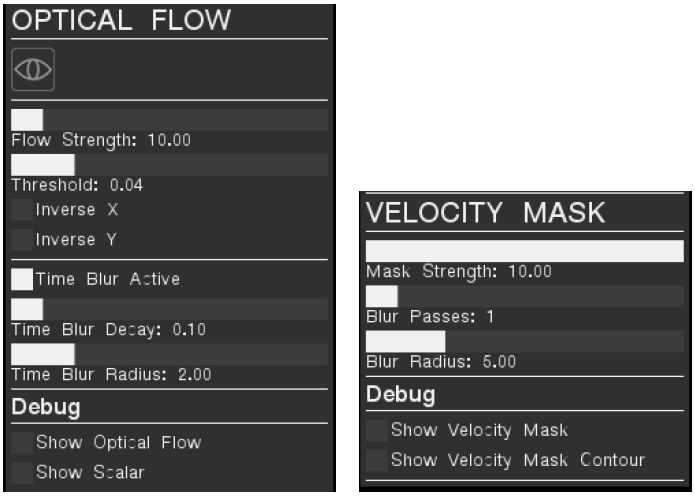
Blur Passes: Number of times we apply Gaussian Blur .

Blur Radius: Radius size of the Gaussian Blur.

Debug

Show Velocity Mask: Draw velocity mask.

Show Velocity Mask Contour: Draw contours detected from velocity mask.



(a) Optical Flow **(b)** VelocityMask

Figure A.4: Optical Flow and Velocity Mask panels

A.6.3 FLUID SOLVER

Eye icon: Activate/Deactivate Fluid Solver.

Reset Icon: Reset Fluid Solver.

General Red: Sets the red color component of all the fluids.

General Green: Sets the green color component of all the fluids.

General Blue: Sets the blue color component of all the fluids.

General Opacity: Sets the opacity of all the fluids.

IR Marker: Sets the *IR Markers* as input to the fluid.

Contour: Sets the *silhouette contour* as input to the fluid.

Solver

Speed: Increases delta time between frames so fluid advances faster.

Cell Size: Size of the two-dimensional grid cells that define the fluid space.

Num Jacobi Iterations: Number of Jacobi iterations to converge to the solution of the Poisson equations that describe the pressure physics of the fluid.

Viscosity: Resistance of the fluid, known also as "thickness".

Vorticity: Magnitude of the rotational motions of the fluid.

Dissipation: Decay of the fluid density.

Advanced Dissipation

Velocity Offset: Offset of the 'Dissipation' in the advection step of velocity field.

Density Offset: Offset of the 'Dissipation' in the advection step of density field.

Temperature Offset: Offset of the 'Dissipation' in the advection step of the temperature field.

Smoke Buoyancy

Sigma: Scalar multiplying the temperature difference. Higher it is the more buoyancy effect.

Weight: Weight of the smoke.

Ambient Temperature: Ambient temperature of the fluid space.

Gravity X: Horizontal component of the gravity force applied evenly to the entire fluid.

Gravity Y: Vertical component of the gravity force applied evenly to the entire fluid.

Maximum

Clamp Force: Maximum force magnitude limit.

Density: Maximum density limit.

Velocity: Maximum velocity magnitude limit.

Temperature: Maximum temperature limit.

Density From Pressure: Multiply fluid density by pressure. This sets the force of the pressure field.

Density From Vorticity: Multiply fluid density by vorticity. This sets the force of the vorticity field

Debug

Show Fluid Velocities: Draw fluid vector field.

Show Fluid Velocities Scalar: Draw fluid vector field as scalar. The red channel contains the magnitude in x and the green channel the magnitude in y.

Show Temperature: Draw temperature scalar.

A.6.4 FLUID CONTOUR

Eye icon: Activate/Deactivate Fluid coming from the *silhouette contour*.

Red: Sets the red channel of the injected fluid.

Green: Sets the green channel of the injected fluid.

Blue: Sets the blue channel of the injected fluid.

Opacity: Sets the opacity of the injected fluid.

Random colors: Randomizes fluid color over time.

A.7 FLUID 2 + CONTOUR

This panel contains the control of the fluid simulation from the *IR Markers* (see section 4.3.2), Fluid Particles (see section 4.4.8) and the different Silhouette Representation possibilities (see section 4.5).

A.7.1 FLUID MARKER

Eye icon: Activate/Deactivate Fluid coming from the *IR Markers*.

Reset Icon: Reset Marker forces.

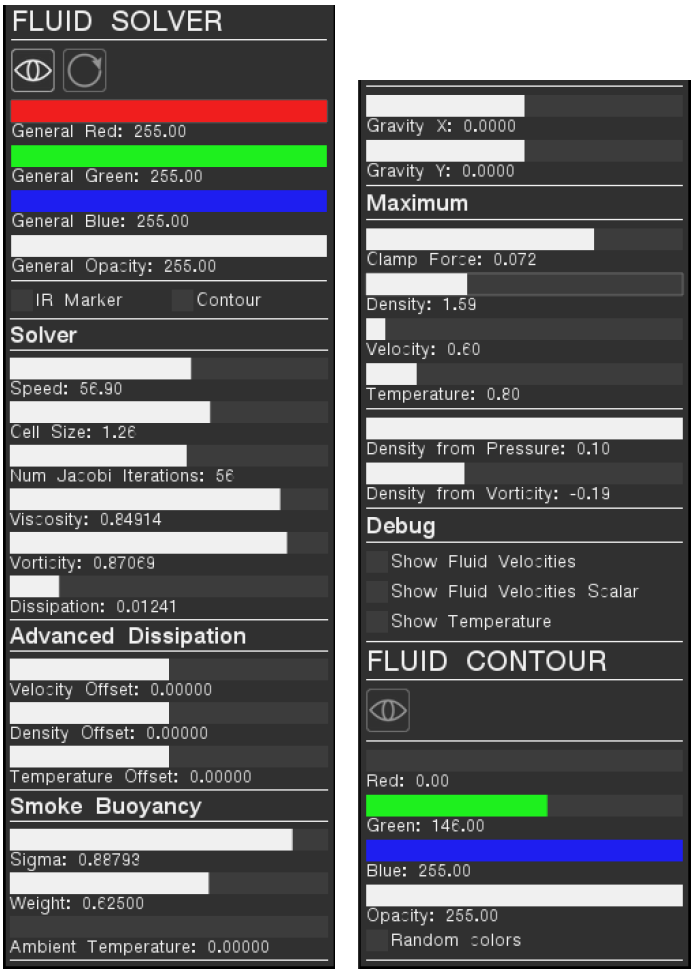


Figure A.5: Fluid Solver and Fluid Contour panels

Force 1: Density

Red: Sets the red channel of the injected fluid.

Green: Sets the green channel of the injected fluid.

Blue: Sets the blue channel of the injected fluid.

Opacity: Sets the opacity of the injected fluid.

Random colors: Randomizes fluid color over time.

Strength: Sets strength of the injected density/color from the *IR Markers*.

Radius: Sets radius of the injected density/color from the *IR Markers*.

Edge: Blurs the contours of the injected color area (less opacity in the contours).

Force 2: Velocity

Velocity Strength: Sets strength of the velocity from the *IR Markers*.

Velocity Radius: Sets radius of the injected velocity from the *IR Markers* (area of the fluid affected by its velocity).

Velocity Edge: Blurs the contours of the area that adds the *IR Markers* velocity to the fluid (less velocity in the contours).

Force 3: Temperature

Temperature: Sets the temperature injected from the *IR Markers*.

Temperature Radius: Sets radius of the injected temperature from the *IR Markers*.

Temperature Edge: Blurs the contours of the injected temperature.

A.7.2 FLUID PARTICLES

Eye Icon: Activate/Deactivate Fluid Particles.

Contour: Interact with the Contour.

Marker: Interact with the IR Markers.

Particle

Birth Chance: Sets how likely it is that new particles are born.

Birth Velocity Chance: Sets how likely it is that new particles are born in velocity areas.

Lifetime: Lifetime of the particles.

Lifetime Random: Randomness of the lifetime of the particles.

Mass: Mass of the particles.

Mass Random: Randomness of the mass of the particles.

Size: Radius of the particles.

Size Random: Randomness of the radius of the particles.

A.7.3 SILHOUETTE CONTOUR

Eye Icon: Activate/Deactivate Depth Contour.

Red: Sets the red color component of the contour.

Green: Sets the green color component of the contour.

Blue: Sets the blue color component of the contour.

Opacity: Sets the opacity of the contour.

Contours

Bounding Rectangle: Draw bounding rectangle.

Bounding Rectangle Line: Draw bounding rectangle line contour.

Convex Hull: Draw Convex Hull.

Convex Hull Line: Draw Convex Hull line contour.

Silhouette: Draw Silhouette.

Silhouette Line: Draw *silhouette contour* line.

Smoothing Size: Smoothing size of the *silhouette contour*.

Line Width: Width of the contour line.

Scale: Scaling factor of the overall contour.

Debug

Show Contour Velocities: Draw velocities detected from computing difference in contour between two consecutive frames (not being used).

A.8 PARTICLE EMITTER

This panel controls the Emitter Particles (see section 4.4.4).

A.8.1 PARTICLE EMITTER

Eye Icon: Activate/Deactivate particle emitter.

Red: Sets the red color component of the newborn particles.

(a) Fluid Marker **(b)** Fluid Particles **(c)** Silhouette Contour

Figure A.6: Fluid Marker, Fluid Particles and Silhouette Contour panels

Green: Sets the green color component of the newborn particles.

Blue: Sets the blue color component of the newborn particles.

Opacity: Sets the opacity of all the particles.

IR Marker: Sets the *IR Markers* as emitting and interacting source.

Contour: Sets the *silhouette contours* as emitting and interacting source.

Emitter

Particles/sec: Controls how many particles are born every second.

Velocity: Sets the initial velocity of newborn particles.

Velocity Random[% :] Adds randomness to the initial velocity of newborn particles.

Velocity from Motion[% :] Lets the particles take the velocity from the marker.

Emitter Size: Increases the area from where particles are emitted.

Emit all time: Born particles all the time.

Emit all time on contour: Born particles all the time but only on the contour shape.

Emit only if movement: Born particles only if there is enough movement.

Physics

Friction: Makes particles velocity decrease over time.

Gravity X: Adds gravity in the horizontal direction.

Gravity Y: Adds gravity in the vertical direction.

Turbulence: Adds noise affecting the motion of the particles.

Bounces: Makes the particles bounce with the boundaries of the window.

Steers: Makes the particles steer away before hitting the boundaries of the window.

Infinite: Makes infinite boundaries.

Repulse: Makes particles repulse from each other.

Repulse Dist: Sets the repulsion distance between particles.

Particle

Empty: Draws empty particles, just the contour line.

Line: Draws a line instead of a circle.

Stroke: Draws a black contour stroke around the particles.

Stroke Line Width: Sets the line width of the stroke.

Connected: Draws a line connecting particles that are closer than Connected Dist.

Connected Dist: Sets the maximum distance between particles to be connected.

Connected Line Width: Sets the line width of the connecting line.

Lifetime: Controls how long the particles live.

Life Random[% :] Adds randomness to the lifetime of particles.

Radius: Sets radius size of the particles.

Radius Random[% :] Adds randomness to the radius size of the particles.

Time Behavior

Size: Decrease particle size over its lifespan.

Alpha: Decrease particle opacity over its lifespan.

Flickers: Make particle flicker before dying.

Color: Change particle color over its lifespan.

Interaction

Interact: Makes the particles interact with the input source.

Interaction Radius: Sets the radius of the interaction area.

Interaction Force: Sets the force of interaction.

Flow Interaction: Make optical flow and *IR Markers* velocities affect particles motion.

Fluid Interaction: Make fluid velocities affect particles motion.

Seek Interaction: Make particles seek markers position (only works for *IR Marker* input).

Gravity Interaction: Make particles that are "touched" fall in a natural way.

Attract Interaction: Make particles attract to input source.

Repulse Interaction: Make particles repulse from input source.

Bounce Interaction: Make particles bounce from *silhouette contour* (only works for Contour input).

A.9 PARTICLE GRID

This panel controls the Grid Particles (see section 4.4.5).

Eye Icon: Activate/Deactivate particle emitter.

Red: Sets the red color component of the newborn particles.

Figure A.7: Emitter Particles panel

Green: Sets the green color component of the newborn particles.

Blue: Sets the blue color component of the newborn particles.

Opacity: Sets the opacity of all the particles.

IR Marker: Sets the *IR Markers* as interacting source.

Contour: Sets the *silhouette contours* as interacting source.

Grid

Resolution: Resolution of the grid (width/resolution x height/resolution).

Particle

Empty: Draws empty particles, just the contour line.

Line: Draws a line instead of a circle.

Stroke: Draws a black contour stroke around the particles.

Stroke Line Width: Sets the line width of the stroke.

Connected: Draws a line connecting particles that are closer than Connected Dist.

Connected Dist: Sets the maximum distance between particles to be connected.

Connected Line Width: Sets the line width of the connecting line.

Radius: Sets radius size of the particles.

Physics

Friction: Makes particles velocity decrease over time.

Return to Origin: Make particles return to its original position.

Return to Origin Force: Force of making particles return to its original position.

Repulse: Makes particles repulse from each other.

Repulse Dist: Sets the repulsion distance between particles.

Interaction

Interact: Makes the particles interact with the input source.

Interaction Radius: Sets the radius of the interaction area.

Interaction Force: Sets the force of interaction.

Flow Interaction: Make optical flow and *IR Markers* velocities affect particles motion.

Fluid Interaction: Make fluid velocities affect particles motion.

Seek Interaction: Make particles seek markers position (only works for *IR Marker* input).

Gravity Interaction: Make particles that are "touched" fall in a natural way.

Attract Interaction: Make particles attract to input source.

Repulse Interaction: Make particles repulse from input source.

A.10 PARTICLE BOIDS

This panel controls the Boids Particles (see section 4.4.6).

Eye Icon: Activate/Deactivate particle emitter.

Red: Sets the red color component of the newborn particles.

Green: Sets the green color component of the newborn particles.

Blue: Sets the blue color component of the newborn particles.

Opacity: Sets the opacity of all the particles.

IR Marker: Sets the *IR Markers* as interacting source.

Contour: Sets the *silhouette contours* as interacting source.

Flocking

Number of particles: Number of particles born at once.

Flocking Radius: Distance at which the boids are considered flockmates.

Lower Threshold: Threshold that determines if boids get separated or aligned.

Higher Threshold: Threshold that determines if boids get aligned or attracted.

Max Speed: Maximum speed of the boids.

Separation Strength: Strength of the separation force.

Attraction Strength: Strength of the attraction force.

Alignment Strength: Strength of the alignment force.

Physics

Friction: Makes particles velocity decrease over time.

Gravity X: Adds gravity in the horizontal direction.

Gravity Y: Adds gravity in the vertical direction.

Turbulence: Adds noise affecting the motion of the particles.

Bounces: Makes the particles bounce with the boundaries of the window.

Steers: Makes the particles steer away before hitting the boundaries of the window.

Infinite: Makes infinite boundaries.

Particle

Empty: Draws empty particles, just the contour line.

Line: Draws a line instead of a circle.

Stroke: Draws a black contour stroke around the particles.

Stroke Line Width: Sets the line width of the stroke.

Connected: Draws a line connecting particles that are closer than Connected Dist.

Connected Dist: Sets the maximum distance between particles to be connected.

Connected Line Width: Sets the line width of the connecting line.

Radius: Sets radius size of the particles.

Radius Random[% :] Adds randomness to the radius size of the particles when born.

Interaction

Interact: Makes the particles interact with the input source.

Interaction Radius: Sets the radius of the interaction area.

Interaction Force: Sets the force of interaction.

Flow Interaction: Make optical flow and *IR Markers* velocities affect particles motion.

Fluid Interaction: Make fluid velocities affect particles motion.

Seek Interaction: Make particles seek markers position (only works for *IR Marker* input).

Gravity Interaction: Make particles that are "touched" fall in a natural way.

Attract Interaction: Make particles attract to input source.

Repulse Interaction: Make particles repulse from input source.

Bounce Interaction: Make particles bounce from *silhouette contour* (only works for Contour input).

A.11 PARTICLE ANIMATIONS

This panel controls the Animations Particles (see section 4.4.7).

Eye Icon: Activate/Deactivate particle emitter.

Red: Sets the red color component of the newborn particles.

Green: Sets the green color component of the newborn particles.

Blue: Sets the blue color component of the newborn particles.

Opacity: Sets the opacity of all the particles.

IR Marker: Sets the *IR Markers* as interacting source.

Contour: Sets the *silhouette contours* as interacting source.

Animation

Rain: Set Rain animation.

Snow: Set Snow animation.

Explosion: Set Explosion animation.

Rain and Snow

Particles/sec: Controls how many particles are born every second.

Explosion

Number of particles: Number of particles born at once.

Interaction

Interact: Makes the particles interact with the input source.

Interaction Radius: Sets the radius of the interaction area.

Interaction Force: Sets the force of interaction.

Flow Interaction: Make optical flow and *IR Markers* velocities affect particles motion.

Fluid Interaction: Make fluid velocities affect particles motion.

Seek Interaction: Make particles seek markers position (only works for *IR Marker* input).

Gravity Interaction: Make particles that are "touched" fall in a natural way.

Attract Interaction: Make particles attract to input source.

Repulse Interaction: Make particles repulse from input source.

Bounce Interaction: Make particles bounce from *silhouette contour* (only works for Contour input).

Particle

Empty: Draws empty particles, just the contour line.

Line: Draws a line instead of a circle.

Stroke: Draws a black contour stroke around the particles.

Stroke Line Width: Sets the line width of the stroke.

Connected: Draws a line connecting particles that are closer than Connected Dist.

Connected Dist: Sets the maximum distance between particles to be connected.

Connected Line Width: Sets the line width of the connecting line.

Lifetime: Controls how long the particles live.

Life Random[% :] Adds randomness to the lifetime of particles.

Radius: Sets radius size of the particles.

Radius Random[% :] Adds randomness to the radius size of the particles.

Physics

Friction: Makes particles velocity decrease over time.

Gravity X: Adds gravity in the horizontal direction.

Gravity Y: Adds gravity in the vertical direction.

Turbulence: Adds noise affecting the motion of the particles.

Bounces: Makes the particles bounce with the boundaries of the window.

Steers: Makes the particles steer away before hitting the boundaries of the window.

Infinite: Makes infinite boundaries.

Repulse: Makes particles repulse from each other.

Repulse Dist: Sets the repulsion distance between particles.