Development of a full-body interaction videogame for children with Autism based on generative graphics and a Kinect-based tracking system

Domènech López, Marc

Curs 2016-2017

Director: Narcís Parés Burgues

GRAU EN ENGINYERIA EN SISTEMES AUDIOVISUALS

Treball de Fi de Grau
A en Martí, la Mercé
i la Paula.
Amics i familiars.
APPRECIATION

I would like to thank Narcís Parés and Ciera Crowell for giving me such a great opportunity to participate in this project and in which I’ve learnt that much, professionally and personally.

Also thank all the people who have helped me in some point of the project, like Guillem and Joan.

And finally thank my family and friends for having supported me during the entire project and for always helping me in everything they could.
ABSTRACT

The main objective of this research project is to develop the tracking and virtual system framework to support a multiplayer game specially destined for children with Autism. With this system, the players are expected to interact and communicate with each other in order to make them develop new social and communication behaviors. To do so, this project is based on a full body interaction system. In order to create such a system, the game uses a Kinect-based tracking system, allowing us to detect gestures as well. Concretely, two Kinects are being used and connected via an OSC connection. The game that will be developed for this platform will be implemented using Unity, a game engine program. Currently, a prototype application has been developed also in Unity to be able to test the technology developed in this TFG. With the objective of bringing this system to special schools and hospitals, the set-up will also be portable. This way it will be easily transportable, and adaptable to different classrooms sizes and configurations.

Keywords: Multi-Kinect, Full-body interaction, Unity

ABSTRACT (CATALAN)

El principal objectiu d’aquest projecte és desenvolupar un sistema virtual i de seguiment que suporti un joc multi jugador, especialment dissenyat per a nens amb Autisme. Amb aquest sistema, els jugadors podran interactuar i comunicar-se entre ells per tal que puguin desenvolupar nous comportaments tant socials com comunicatius. Per això, aquest projecte es basa en un sistema d’interacció de cos sencer. Per tal de crear el sistema el joc utilitza el sistema de seguiment de la Kinect, permetent-nos la detecció de gestos de l’usuari. Concretament, s’utilitzen dues Kinects i estan connectades via OSC. El joc ha estat implementat amb Unity. Actualment, s’ha desenvolupat, també en Unity, un prototip per tal de testejar la tecnologia utilitzada en aquest TFG. Amb l’objectiu de portar aquest sistema a les aules i hospitals, aquest serà portable per tal que sigui fàcil de transportar, i s’espera fer-ne diferents versions pel que fa a les seves dimensions per tal d’adaptar-se a diferents classes i configuracions.
This project has been developed in order to help children with Autism improve social and communicative behaviors.

In order to achieve this objective, a full body system based on a multi-Kinect-tracking system has been developed and, while the children are playing, they will be developing these new skills.

In the project it will be seen the development of this system from the very beginning; from the selection of the Kinect version, to the implementation of the code used to distinguish both users. It will also be seen the compatibility and connection of both Kinects.

At the end, a first version is released and future work is exposed, since this is a research project and other researchers will go on to complete this project using this system as a basis for the future versions.
## INDEX

ABSTRACT ............................................................................................................ vii

1. INTRODUCTION ................................................................................................. 1
   1.1. Calendar ........................................................................................................ 2

2. STATE OF THE ART ............................................................................................. 5
   2.1. Autism ......................................................................................................... 5
   2.2. Similar Projects .......................................................................................... 6

3. PHASE 1: REQUIREMENTS GATHERING .......................................................... 9
   3.1. Materials Specifications ............................................................................. 9
       3.1.1. Projector EPSON EB-430 ................................................................. 9
       3.1.2. Kinect for Windows v2 ..................................................................... 11
   3.2. Comparing the Kinects ............................................................................. 14
   3.3. Location of the Kinects ............................................................................. 15
   3.4. Location of the Projectors ......................................................................... 16

4. PHASE 2: UNITY ENVIRONMENT ..................................................................... 19
   4.1. Particle System .......................................................................................... 19
   4.2. Kinect Avatars .......................................................................................... 23
   4.3. Calibration between physical space and virtual space ............................. 24
   4.4. Camera Calibration .................................................................................... 26
   4.5. Projector Calibration ................................................................................ 27

5. PHASE 3: COMMUNICATION OF MULTIPLE KINECTS .................................. 29
   5.1. Connecting the Kinects ............................................................................. 29
   5.2. Compatibility between Kinects ................................................................. 30
   5.3. Relationship between classes .................................................................. 31
   5.4. OSC Connection between Kinects ............................................................ 33
   5.5. Sending and receiving OSC messages ...................................................... 35
   5.6. Creating a solid group of data from both Kinects .................................... 39
   5.7. Distinction of both users ........................................................................... 40

6. PHASE 4: INTEGRATION OF THE GRAPHIC CONTENT AND PROTOTYPE. 51

7. ADAPTIVE VERSIONS ......................................................................................... 53

8. CONCLUSIONS .................................................................................................. 57
   8.1. Results ....................................................................................................... 57
   8.2. Future work ............................................................................................... 58
   8.3. Personal evaluation .................................................................................... 58

9. BIBLIOGRAPHY ................................................................................................. 59

10. APPENDIX ......................................................................................................... 61
    10.1. Code ........................................................................................................ 61
    10.2. Particle system code ............................................................................... 74
    10.3. Installation guide ..................................................................................... 77
    10.4. Manual of OSC Connection .................................................................. 85
    10.5. Projector EPSON EB-430 data sheet ...................................................... 87
    10.6. Kinect detection ranks plans .................................................................. 89
    10.7. Defining the number of particles in the particle system ....................... 90
1. INTRODUCTION

The very first section of the project is meant to give a slightly touch of what will be seen in the next sections and also to introduce the main topic of the project. It will also be seen the opportunity to have formed part in such a project and the main objectives.

It can be said that this project is part of the continuation of previous projects, mainly the project called *Lands of Fog*. This was based on the installation and development of a full body interactive system providing children who suffer from Autism the possibility to develop social integrating skills while they are playing.

The opportunity of forming part of this project is thanks, in part, of the good results obtained by this previous project, obtaining good results within the field of technology and Autism.

As regards to the motivation, it comes up when the final objectives are mentioned. The fact of working in such a project that tries to develop a game with the aim of helping a certain children that suffers from Autism is one of the most motivational points. The fact of, at one point, being able to see that you can help someone with your own work is very retrieving.

Apart from that, getting deep inside in this programming world, concretely in the video games world, for the first time has been appealing. Apart from that, being also for the first time part of a research project was motivating.

Basically, all the facts mentioned before put all together have created a very motivational and ambitious project that was difficult to refuse.

Additionally, this project was also thought to be a base for upcoming projects that will use full body interaction systems. So, one of the main objectives is to achieve a system where, in a future time, the researchers will be able to adapt this system to their project without the need of having to develop it again.

In terms of objectives, with the aim of developing a multi Kinect full body interaction system, the project was divided in three main parts:

- Requirements gathering
- Multiple Kinect communication
- Integration of the graphic content and prototype

Regarding the first objective, it was mainly related to doing research on how the project would be faced and what path had to be followed. A part from that, in this section there was also a part of becoming familiar and thinking about the disposition of all the devices used in the project; concretely the location and technical specifications of the projector EPSON EB-430 and the Kinect for Windows version 2. All this research was meant to be done between the months of January and February. However, a last
verification of this location of the material with all the system done is also necessary to finally adjust all the equipment in the best possible way.

Secondly, there is the multiple Kinect communication. In this part of the project, and having the entire equipment configuration, at least, suggested, the most important part was to make both Kinects compatible between each other with the aim of forming a solid group of data. In this stage, it was very important to have implemented a system able to detect and distinguish between the users, even if they crossed between each other. This part of the project was thought to be done between the months of March and April. However, because of some problems regarding to the multi Kinect compatibility the plan has been extended until May. It could be said that this is the main and most important part of the system as all the implementations are done in here, from the Kinect connections to each computer until the user distinction.

Last but not least, and once the graphic content of the game is available and all the above parts are also working properly, a good prototype of the game can now be developed. In this part, the testing of the system is also included, to see if everything is working fine and in case there were bugs, to have time to fix them. In a brief explanation, the final system has to support:

- Being functional with two users
- The graphic content has to respond to the user movements and gestures.

Since the graphic content was not available by this time, this part of the project has now been extended for a future work, which will be handled by designers.

### 1.1. Calendar

Initially, the project has been divided in three great part:

- Requirements recollection
- Communication between multiple Kinects
- Integration of the graphic content and prototype

Once divided these parts, the first part, which was based on knowing the hardware capabilities that will be used. So, during all this time, the work has been based in doing research in aspect such as Unity, the Kinect that will be used, the projector that will be used and their corresponding technical specifications in order to be able to configure the system in the best way respecting the hardware available.

The second, and once done all the configuration system, consisted of consisted of making both Kinects compatible between each other and that their detections form a solid data base. This part of the project should be done by March and April. In addition to that, in this part of the project the entire system that will be used has to be created. To do so, Unity will be used.

Once the Kinects are compatible between each other and data can be obtained from both, the next and last phase of the project starts, concretely in the months of May and July. This is the last part of the project and is where the graphic content should be added. Additionally, in this part the testing of the system should also start.
In the following figure, a Gantt diagram shows when all of these parts have been done and how each part of the project has been divided.

**Illustration 1** Gantt diagram of the development of the project
2. STATE OF THE ART

2.1. Autism

According to the Diagnosis of Autism spectrum disorders (ASD) in the first three years of life, Autism is a neurodevelopmental disorder characterized by impaired social interaction, verbal and non-verbal communication, and restricted and repetitive behaviour. Autism usually appears at an early age, concretely when the child is two or three years old. [1]

Currently Autism is known to have not only one specific type but many different types. This is why children with Autism are, once diagnosed with ASD, divided in subgroups. Thanks to this subdivision research groups and therapists can focus themselves in a specific type and learn more about it.

The diagnosis of Autism is based on behaviour. Autism is characterized by persistent deficits in social communication and interaction across multiple contexts, as well as restricted, repetitive patterns of behaviour or activities. [2, 3]

Several tools are used to diagnose Autism. The three most common ones are:

- **Autism Diagnostic Interview Revised (ADI-R):** based on a semi-structured parent interview.
- **Autism Diagnostic Observation Schedule (ADOS):** uses observation and interaction with the child.
- **Childhood Autism Rating Scale (CARS):** widely used in clinical environments to assess severity of autism based on observation of the children.

Since there are several different groups of Autism, the project has to focus itself in one of them. To do so, the ADOS classification has been used. The classification of this diagnosis is based on the age of the children. There are four different groups, which are the following: the first group is formed by children who have a low or null verbal capacity; the second one is formed by children who are not fluent when talking; the third one is formed by children who speak fluently; and the last one is formed by teenagers and adults with a fluent verbal capacity. [4]

Once all the classifications are defined, our game is destined for the children of group three. A part from that, we have divided the children from this group depending their intellectual coefficient, selecting at the end the children who had an intellectual coefficient higher than 70, also known as high functioning autism.

In relation to that group of children, they are the ones that have more possibilities to have an independent life as long as they are properly looked after during their childhood.
2.2. Similar Projects

One of the main important features of our project was that it had to be a full body interactive game. It can be said that this is the tool used for intervention on motivation and social initiation for the children of our group.

Thus, some previous projects have been used as a reference in order to succeed in getting the best results in our project.

The main projects used as reference have been:

- MEDIATE

MEDIATE (Multisensory Environment Design for an Interface between Autistic and Typical Expressiveness) [5] is a project that its main objective was to design and develop a physical environment that allowed the children to interact with multimodal stimuli in order to give them a sense of control.

In order to achieve the main goals, the communication between the user and the system had to be very simple as well as a dialogue based on the user actions had to be created too. Hence, this is why full body interaction is used; the simple actions that the player has to do in a full body interacting system, such as moving or gesticulating, would provide the user this sense of control.

As regards to the conclusions of the project, the psychologists confirmed that the developing of full body interacting systems for children with Autism has some advantages. For instance, one of this advantage is to make the children who participate in this project understand the cause-effect relation in their environment.
• Lands of Fog

Another reference project has been Lands of Fog [7]. It is quite similar to the project commented before.

This full body interaction installation in which the two players were also grabbing kind of a lantern had the main objective to help children with Autism in social interaction.

In this circular projection, the space was divided in four different zones: water, grass, ice and lava. This way, as the user was walking around, the projection from where he was passing through was also changing showing one of these zones. Each of these zones had different behaviours.

A part from the different zones, the player could discover insects that had particular actions and behaviours attached. With the aim of making both of the players interact between each other, to proceed and get better characters, they had to place their lanterns together in order to evolve to new characters.

Illustration 3 Two users playing Lands of Fog
Development of a full-body interaction video game for children with Autism
3. PHASE 1: REQUIREMENTS GATHERING

In this part of the project it will be explained what materials are being used as well as their specifications. Since this project is meant to be displayed, projectors will be needed. Additionally, as it is a full body interaction game a device to capture the player data and transfer it to the computer will also be needed. To do so, Kinects are being used. It will also be seen why this specific version of the Kinect has been chosen and all along with the explanation of the materials the location of each of them will also be explained and showed.

3.1. Materials Specifications

In this project two projectors and two Kinects will be used in order to have a large playground area.

The fact of having a big playground area will allow the children to have their own personal space to play by themselves and they will also not feel that anyone is occupying their own space. Furthermore, the game will give the children instructions and notifications so they will notice that they have to interact with the other player in order to achieve new objectives in the game and in order to start developing new skills such as communication.

Another purpose of the game is that children do not isolate themselves during the game. This is why the playground will be round. If it was a square, there would be the possibility that the children would go to the vertexes in order to isolate themselves; this way, there is no other way than starting communicating with the other player.

3.1.1. Projector EPSON EB-430

The projector used in the game will be the EPSON EB-430 [9].

Some of the main features of this projector are that it is one of the most used in the classrooms, which is a good point as the game is destined to be placed there.
Another important aspect is its price-quality relation. The projector presents high definition content and the low price make it affordable for most of the schools.

Last but not least is the 0.55:1 throw ratio, which allows the projector to be quite close from the screen in order to avoid possible shadows. In our case, as it will be pointing the floor, it is a good feature as we will be able to place close from the floor and there will be no shadows so the projection will be seen at its fullness.

Entering to more technic aspects, the projector has a 1024 x 768 resolution. Regarding to the projection ranks, we are willing to achieve a 5 x 5 meter playground. The good thing about these projectors is that their projection rank is variable and depending on the projector’s configuration a bigger or smaller projection can be achieved.

As the game is displayed by two different projectors, the configuration has to match in order to achieve a smooth change between the two projections so there is no noticeable change. In order to achieve this, the configuration has been the following:

- Angle: 10°
- Keystone:
  - Horizontal: 0
  - Vertical: 20

With this configuration set, the projection achieved has dimensions of 320 centimeters horizontally and 250 centimeters vertically. Taking into account that we dispose of two projectors, this dimensions can be doubled, achieving this way the 5 x 5 meter dimensions that it was firstly thought.
3.1.2. Kinect for Windows v2

The Kinect is a sensor that is able to detect objects that are positioned inside its detection ranks. The device has a RGB camera, a depth sensor and multi array sensors and thanks to all this software together the Kinect is able to reach a 3D full-body capture, facial recognition and voice recognition capabilities.

The most important feature of the project is the Kinect since without it we would not have been able to detect the players in the playground area.

Before start talking about how the Kinect will be used in our project, it is necessary to explain its functioning. First of all, the sensor is formed by the following components [10]:

- RGB camera
- Depth sensor
- IR emitters
- Microphone array

In the following image it can be seen how all these components are distributed in the sensor:
The sensor functioning uses a very innovative technology, which has turned out to give impressive results in a good way. This new technology is called Time-of-Flight [11], also called ToF, and it basically emits light signals and the measures how long it takes them to return. With such technology, the camera is able to differentiate light reflecting from objects in a room and the surrounding environment providing the users with an accurate depth estimation that enables the shape of these objects to be computed.

Another new and improved feature of the newest release of the device is the infrared sensor; that enables their users to capture a wider field of view with greater accuracy and higher resolution. This increase of accuracy and resolution allows the device to, a part from being able to identify an object with little to no light, an improved hand-pose recognition. Additionally, this improved infrared sensor is now performing awesomely. For instance, it can now recognize people and track bodies without the necessity of light. This, also can be related to the fact that now the device is able to detect smaller objects, such as hands; being able to detect the player’s hand from up to four meters. The hand-pose recognition allow the users to interact with the game only using their hands, and giving the users a better experience.

Another important feature of the device is to know how it measures the coordinates of the user being tracked, as they will be afterwards. In the following figure it can be clearly seen how the coordinates are computed:
As it can be previously seen, if the player gets closer or further, then the z-coordinates are changing; if the player is moving horizontally, the x-coordinates are changing; and, finally if the player, for instance, moves his leg up and down or jumps, the y-coordinates will be changing.

In our project, we will be using the Kinect for Windows version 2. The decision to choose either of the two Kinects that are on the market now was meditated. We had to think and compare the characteristics of both of the Kinects and then select the one that suited us the most. This is why we chose Kinect for Windows Version 2. In the next section a comparison between the two versions will be explained.

In the project two Kinects are being used. This way, we achieve a bigger playground and a better player detection among the entire playground.

Technically speaking, our Kinect can reach up to 4.5 meters, giving us the possibility to have a 8 x 8 meter playground if we had the space, however, as this game will be placed in a classroom, it would be quite big and the playground dimensions will vary between 4 x 4 meters to 5 x 5 meters. Regarding the detection ranks, we also have to think about how wide the Kinect can detect. In this case, this Kinect can detect from 70 degrees in the horizontal view to 60 degrees in the vertical view. So, this is important because we will have to place the playground inside all the detection ranks otherwise the player will not be detected.
3.2. Comparing the Kinects

As we mentioned before, we could choose between the two versions of Kinect that are available so far and we decided to go for the last version of the device as it suited us way better thanks to all its new and improved characteristics.

In the next figure we can see the comparison of the specifications of both Kinects:

![Illustration 8 Table comparing the specifications of the two Kinects available](image)

As we can see in the previous figure, there are all the specifications of both Kinects compared.

Briefly commented, we can clearly see that the new version of the Kinect is way better in terms of specifications. Some of the most remarkable ones are [12]:

- The improvement of the depth camera: while in the first version the depth camera resolution is 320 x 240, the newest release of the device has a resolution of 512 x 424. This upgrade in terms of the depth resolution allows the device to build a better 3-D map of the room, objects and people within it.
- The wider detection rank horizontally and vertically achieved by the last version of the device; allowing the device to have a wider rank detection, which means that the user has a bigger space to move around while still being detected.
- The capability to track up to 6 skeletons.
- The substitution of the way to compute the depth has increased the performance of the newest device. While the first version started using IR light pattern projection in order to compute the depth, the second version changed and started using time-of-flight (TOF) which has turned out to be way better. This way of
computing the depth of objects consists of throwing some infrared light rays and looking how much time these rays need to bounce on surfaces and come back. This new method provides more stability, precision and less interferences.

- The skeletal tracker also outperforms the older version, providing the users of the new version of the Kinect more tracked users at the same time (up to six as we have seen before), more velocity and more precision.

New additions have also been done in the new version that the older one did not count on. For instance, hands thumbs tracking is one of the new features that allows the user to know how the hand is being rotated.

Besides all the advantages that the Kinect version 2 presents, there have also been disadvantages that supposed important problems in our project.

One of our main concerns was the fact that due to the Microsoft Kinect SDK that needs to be installed in the computer in order to make the Kinect work is limited to only one Kinect per computer. In our case, we were planning to use two Kinects in the same computer and we had to change a bit our plans in order to adapt to this limitation. Also related to this aspect, the Kinect that we finally chose only supports USB 3.0. However, still in the multi Kinect topic, the generation of interferences between two Kinects when working at the same time with them has decreased considerably. So, despite having to use two computers in case two Kinects are needed, it still provides a better performance comparing it if we used two Kinects version 1.

### 3.3. Location of the Kinects

Having two Kinects provides us a better player detection and a bigger playground area for the game. So one of the things to think about was where we should put the two Kinects in order to detect all the users in a way that was not so confusing at the time of transferring the data from one Kinect to the other one.

There were different options, and we finally decided to put the two Kinects facing each other. This way, the playground will be divided in two different zones; one for each Kinect and depending where the players are a different Kinect will be getting the data.

Furthermore, the fact of having the Kinects facing each other is the combination that allows us to reach the biggest playground as each Kinect detects up to 4.5 meter. Apart from that, this is the easiest case as regards to the Kinect calibration as we will only have to take into account that the Kinects will be acting like a mirror as they are facing each other.

In the next figure the location of the Kinects can be seen, also with their respective detection areas.
As it can be seen in the figure below, the Kinects will be placing each one in a part of the playground and facing each other. As well, the Kinects will not be placed straight after the limits of the playground, because this way the Kinects will not detect the children if they start getting closer to them. If the Kinects are placed a bit further respecting the limits of the playground area, then the children will be detected in the entire playground area no matter where they are walking around.

Additionally, the detection distance is now doubled up with the addition of a second Kinect. Even though a simple Kinect could detect up to four meters and a half, it is not strictly necessary to force a Kinect to detect a player placed at four meters from the Kinect because the detection will not be as accurate as if there is the second Kinect placed in the other part of the court and it can perfectly detect the player as it is way closer from the second one rather than the first one.

3.4. Location of the Projectors

Regarding to the projectors’ location, they will also be facing each other. As the projectors are meant to be placed close to the screen in order to avoid possible shadows, we think that the best place for them is on top of the Kinects’ location.
According to the exact position of both projectors, they have been placed facing each other. In addition, the projectors have been placed a distance of a meter away from the projection in order to achieve the expected dimensions.

As regards to the exact specifications, the following enumeration specifies them:

- Height of the projector in respect with the floor: 157 centimetres
- Angle of the lent in respect with the floor: 10 °

Apart from all the specifications above, we also had to think about the structure that will be holding the projectors. Since we want a portable kit so the teachers can be moving the projectors and Kinects around the class so they don’t disturb them, we thought about building a structure that would allow the teachers to change the angles and the projection ranks. This way, the game can be adapted, in a certain way, to different classrooms sizes.

To do so, a prototype has been proposed and can be seen in the next figures:

As we can see in the figure above, these structures allow us to adjust the height and angles to adapt to the playground sizes. The structure can hold the projector from one to two meters.

With this structure, we achieve a very portable and easy to carry system in which we will only need a free space to project all the game. It will also be a way to keep all the materials needed for the game together and without being so spacious.

In the next figure it can be seen the disposition of the two projectors all together with the Kinects:
Illustration 12 Location of the projectors
4. PHASE 2: UNITY ENVIRONMENT

First of all, the software used to create all the system is Unity. Unity is a cross-platform game engine developed by Unity Technologies, which is primarily used to develop video game and simulations for PC, consoles, mobile devices and websites [16].

The programming language in this game engine can be variable; in our case C# will be used. However, Unity allows the developer to create and make actions directly from its platform without the necessity to implement code.

In this section, we can divide all the process in two different parts:

- The particle system
- The Kinect avatars or players.

Regarding to the particle system, it is the place where all the action will take place, where the players will be walking around and where the players will be interacting with the other objects of the system.

On the other hand there are the players and their recognition. In Unity, they are represented as avatars and the avatar will be doing exactly the same movements that the player does. In our project, it will be useful, as we will know where the player is and all his gestures.

Before starting this section, the components used in Unity should have to be explained in order to understand the next sub sections.

4.1. Particle System

The particle system is the part where all the players of our system will be interacting. Although Unity has its own particle system object implemented, a customized particle system has been created in order to satisfy all the project needs [17].

By definition, a particle is a small, simple image or mesh that is displayed and moved in great numbers by a particle system. Each particle represents a small portion of a fluid or amorphous entity and the effect of all particles together creates the impression of the complete entity. Despite the fact that there was this implemented, our players had to interact and collide with the particle system and this is why a customized particle system had to be implemented.

In order to be able to interact with our particle system and also to interact and collide with the same particle system itself, game objects needed to be used.
With all these decisions taken, we finally decided to use, as a test, the following game object:

The above object has been the one used in our system; and next to it, we can see its corresponding collider. Although the particle system provided by Unity allows to collide with planes, colliding with the same particles of the system is also needed. This way, the particles of the entire system can collide with everything.

Additionally, as we are implementing our own particle system, a conjunct of this exact game object has to be created in order to form the whole particle system. To do so, a script done to create clones of a specific game object has been attached to this game object. This way, a number, specified by the user, of game objects will be cloned as the one attached.

What this script basically does is to, from the game object where the script is attached, it gets its mesh and renderer, also with its own material, and from there it clones the empty game object creating a number of game object that are exactly the same.

The final result after implementing the script and attaching it to the game object desired to be cloned is the following:
As we can see, there are all the particles that are also colliding between each other and in the case we added another game object that was not part of the particle system, e.g., an avatar getting the Kinect information, it would also interact with the whole system as they are game objects with their own colliders.

Furthermore, as it can also be seen in the figure below, the game objects are kind of floating in the air. This is because there are 6 planes restricting the area and that are also colliding with the same particle system. The fact that they are not displayed is because they have the mesh disabled, but they are still in the scene. If there were not these planes in the scene, the game objects would all fall down since they have also gravity. In the following figure the composition of these planes is showed:

As it can be seen, the planes form, all together, a perfect cube where all the particle system will be taking place.

If we finally compare our final system, formed by game objects, and the one implemented by Unity, big differences can be seen, like the two following figures:
As it can be seen, there are clearly big differences between our implemented system and the Unity one.

To start, the main difference is that, as stated before, our particle system allows collision between the whole system and also between other game objects that are not part of the system. This main difference can also be clearly seen in the next two figures. As it will be seen, our implemented system is perfectly colliding with the whole system; even if we moved it all around the system it would still collide with all the particles. On the other hand, the particle system does not seem to collide with any game object; starting with the fact that it does not collide with its own particles, if we add another object, nothing different happens.

Another difference, also related with the collision, is that it is quite difficult for Unity particle system the collision between the planes that are restricting the areas. All the particles go out of the restricted boundaries, and once they are outside of the boundaries, they disappear and the particle system emits all the system again. So, it can be said that another fact that is behaving differently in both systems is that while in our system all the game objects are cloned at the very beginning of the game, the Unity system emits constantly a new system of particles. This way of continuously emitting particles is not suitable for our project as we would like to have the system controlled all the time and not constantly disappearing and being emitted every time. Regarding our system, we can select how many game object we want to clone, giving us the possibility to know exactly the amount of particles of our system.
4.2. Kinect Avatars

The other part of the system, and the most important one, are the players. To display the players in our system avatars have been used. With them, the players will be able to interact with the whole particle system.

As it has been seen in previous sections, the avatars of the game are formed six game objects: five spheres and a cylinder. These six game objects represent the six body positions sent from the second computer to the main one.

Two important features of Unity form these game objects that, all together, form the avatar of the game:
- Mesh renderers
- Colliders.

The mesh renderers [18] are the ones in charge of assigning the geometry of the mesh filter and render it in the position defined by the object’s transform. In a more general way, it is also the one in charge of enabling of disabling the display of the game object in the screen, i. e., if the mesh renderer is enabled, the game object will be displayed, and otherwise it will not be displayed in the screen.

On the other hand, there are the colliders [19]. These are the ones in charge of defining the shapes of the game objects in order to allow physical collisions to happen. Colliders are transparent and there are different shapes available; in our case, two different shapes are used: capsule colliders and sphere colliders.

In the next figure it can be seen the colliders that form the avatar:
As it can be seen in the above figure, the colliders also form a human shape. Additionally, it can be seen that there are more colliders than game objects attached. This is because although only having six game objects attached, there are missing parts of the body and they also have to collide with other game objects if it is the case. The avatar is formed by capsule colliders and sphere colliders.

### 4.3. Calibration between physical space and virtual space

Before start talking about the camera calibration and the projector calibration the relation between the physical space and the virtual space needs to be very clear, otherwise the system will not be taking the correct coordinates from the Kinect and, consequently it will not be projected properly.

In this section, two different spaces are defined: the physical and the virtual. The physical space corresponds to the space where the players will be walking around, i. e., the real world. Regarding the virtual space, it corresponds to the world created in Unity; this is the place where the position of the materials used is defined in order to establish a good relation between the real world coordinates and the virtual coordinates.

To start, the physical will be defined, and it can be seen in the following figure:

![Illustration 23 Physical space](image)

As it can be seen in the above figure, this is the disposition of the playground area and the Kinects with their respecting coordinates.
Once explained how the physical space is structured, the relation between this space and the virtual space has to be done. To do so, the Kinects will be replaced as cameras in the Unity space, and thanks to that they will be attached to the skeletons that are being tracked by the Kinect, otherwise the Kinect would take as origin where the skeleton has been placed in the system.

In order to achieve the desired relation between the two spaces, the plane will have to be placed in the same coordinates as in the physical space, i.e., a 5 x 5 plane will be created with its vertexes in the same coordinates as in the physical space. Once replicated the plane, it is time to link the Kinects with a camera in order to let the system know that it should now take that camera as a reference system. This way, the Kinect will correctly convert the user’s position from the virtual space into the physical space.

In a more graphical way, the following figure shows how this link should be done.

![Conversion from physical space to virtual space](image)

In the figure above it can be seen how the conversion from the physical space in the real world to the virtual space in Unity has been done. At this point, if a player was tracked, it would not be placed in his real world position since Unity is still taking as a reference system the position of the game object. For instance, if there was a skeleton of a player located in the coordinates (1,0,1) and then a player gets in the scene in the coordinates (2,0,3) and the Kinect tracks him, the skeleton of the player tracked in the system would not go to the player’s position, it would stay at its position and start imitating the player, which would not be correct. This is because the reference system of the Kinect is not well defined. Now the Kinect is taking as a reference system the game object’s position whereas the correct reference system in our project should be the Kinect.

To do so, the camera that appears in the virtual space figure, that corresponds to the Kinect, should be attached to the player’s inspector so that it changes its reference system.
In the player’s inspector, concretely in the script that has attached called ‘Avatar Controller’ provided by the package used, there is a variable that allows to attach a camera in there in order to set it as a reference system. This way, the player’s position will now be relative to the camera attached. Note that since there are two players, this configuration will have to be applied in the player two inspector as well.

### 4.4. Camera Calibration

As stated before, two Kinects will be used in the project; concretely they will be placed facing each other. This way, the detection rank is higher and more accurate, what it actually means that a bigger playground area can be achieved.

In terms of Unity, when a player is detected and an avatar is assigned to him, then the game object starts moving right where it was placed in the scene. For instance, if an avatar was firstly placed in the coordinates (1,2,2) and the Kinect detected a player, then the avatar would take that starting position as the coordinates origin. In our case, this is not what it was wanted. The system had to know, in some way, where the player was in the world coordinates system in order to allow the player to start anywhere he wanted.

To do so, the unity package used in our system to track the users has an option enabled to attach a camera to each avatar of the system that will represent the Kinect’s position in our game. This way, when the Kinect detects a player, this one will be correctly placed in the game as the system is now taking his position respecting the camera attached to the avatar. By this time, what we will only have to do is place a camera in the position where one of our Kinects is placed in the real life and then attach it the two avatars and the real world coordinates of the players will be correctly obtained.
Since two Kinects are being used in two different computers, this process had to be done twice. However, the process changed slightly depending on the Kinect as they were placed in different positions. Regarding the first Kinect, the one attached to the main computer, it had to be placed in the following coordinates:

- **x-coordinate**: 2.5 meters, as it is needed to be placed in the middle of the playground.
- **y-coordinate**: firstly, it has been placed at 1.13 meters, which means that the Kinect has been placed 1.13 meters up from the floor, but it can be easily changed if we changed the Kinect’s position.
- **z-coordinate**: it is also similar to the y-coordinate as it can vary depending on where we place the Kinect. For this coordinate it is important to think that it is better to place the Kinect a bit separated from the playground area otherwise if the player is close from the limit and in front of the Kinect he will not be detected. Firstly, it has been placed at -0.7, which means that it is 70 cm far from the starting playground area.

In the case of the second Kinect, the fact of being placed right in front of the other one meant that the x-coordinate would be different. Concretely, at 5.7 meters approximately. This way, when the user gets closer from the Kinect, the coordinates will be close to 5, and then the computer will send these coordinates and they will be properly displayed in the main program.

In the next figure the disposition of both Kinects in the Unity scene can be appreciated.

![Illustration 26 Kinect disposition in the Unity scene view](image)

### 4.5. Projector Calibration
Regarding the projector calibration in the system, the case is quite similar to the camera calibration one. Thanks to the new Unity updates this calibration can also be done in the same program without the need of having to download a different program or implementing some code. However, a few time ago, there was not the possibility to do it this new way and a program called Middle VR had to be used in the case that two projectors wanted to be connected with the game.

Nowadays, if the game has two projectors connected in order to display it in the floor, nothing has to be done but placing the cameras in the position where the projectors will be placed in the real life. A part from that, another important aspect is the configuration of the field of view that corresponds to the detection angle that the camera will have.

In regards to our project, two projectors are also being used as the playground area is meant to be 5 x 5 meter and a single projector would not be able to display the whole area. In order to connect the two projectors to our game we will only have to place the cameras to the same world position where the projectors will be places.

What it will be displayed by the two projectors can be seen in the following figures:

As it can be seen in the two previous figures, this is what will be displayed by both of the projectors connected in our game. This way, the entire playground will be displayed. However, it can be clearly seen that there is a part of the playground repeated in the two projections. In order to avoid seeing a clear difference in that repeated zone of the projection, a quad has been added to decrease the brightness in that particular zone of one of the projectors.

This quad will have a fade in order to reduce this amount of brightness in one of the projectors due to the fact that the same part of the playground area is displayed twice, one for each projector. Regarding to this fade attached to the quad, it will go from black to transparent applying the black part at the repeated part.
5. PHASE 3: COMMUNICATION OF MULTIPLE KINECTS

In this part of the report we will take a step forward and we will talk about more programming aspects, concretely, how the Kinects have been connected and how the system has been built and its components. In addition, it will also be explained how the user detection and distinction has been done, which has also been an important part of the project among with the multi-Kinect connection.

5.1. Connecting the Kinects

In our project we will be using two Kinects and, consequently, as we are using Kinect version 2 we will finally have to use two different computers and an OSC connection between the two of them in order to establish a communication to transfer all the data from one computer to the other one.

Apart from that, which will be explained in detail in further sections, we do have to connect the Kinect with its respecting computer in order to have the users tracked in our system. To do so, we have had to follow some instructions that included installations of specific programs and drivers.

Firstly, we will get into the computer’s specifications so we can get the data information from the Kinect. There are two important requirements among all the others, which are the operating system and the USB port. Regarding the operating system, we do need to have at least a Windows 8. Concurrently, the USB port has to be 3.0 otherwise the computer will not be able to detect the Kinect [13].

Secondly, once we accomplish all the computer requirements, it is time to install the Microsoft SDK; concretely we have to install Kinect for Windows SDK 2.0. This SDK will enable us to create applications supported by the Kinect.

After all that, now our computer can get data information from the Kinect and use it in our program, in this case Unity. Now we have to get the necessary Unity packages in order to be able to track the players and display them in our Unity system. To do so, we use the Unity package called Kinect v2 Examples with MS-SDK.

With this package, we can see an avatar in our scene that simulates the movements of our players, as we can see in the following figure.

An avatar is formed by game objects that, all together, have a human shape. This way, when the avatar is interacting with our system, thanks to the colliders attached to all its extremities, it will interact with the system.
As the above figure shows, this is the shape of an avatar provided by a Unity package. It is really similar to a human shape and it also imitates all the movements that a user does when the Kinect is detecting it. In addition to this, Unity provides a huge variety of packages, that can be uploaded by the same Unity users, that contain lots of avatars. However, as in our project no avatars will be displayed in the screen, the avatar was not that important and the testing has been done with the default one, this one.

**5.2. Compatibility between Kinects**

The fact of having two Kinects connected and getting user information instead of only one allows us to have a bigger playground and also to get a better user detection in the entire playground. However, adding another Kinect to our project was not as easy as it seems because, as mentioned before, the Microsoft SDK does not allow to have multiple Kinects connected to the same computer, and this was out first intention.

So, in a very first moment, and still wanting to have both of the Kinects connected in the same computer, we decided to use another kind of drivers instead of the Microsoft SDK called libfreenect2 [14].

Libfreenect2 is an open source driver for Kinect version 2 that supports multiple Kinects in the same computer. Libfreenect2 does work like the Microsoft SDK though. If we would had used these drivers, we would have only got the next information from our devices:

- RGB image transfer
- IR and depth image transfer
- Registration of RGB and depth images
Although two Kinects could be connected at the same computer, this was quite limited since we needed a skeleton tracking for our players. In addition to these drivers, we found out that installing complementary drivers apart from libfreenect2 we could get that information. These complementary drivers are OPENNI and NiTE.

OpenNI, also called *Open Natural Interaction*, is a set of open source drivers released by the company called PrimeSense that, all along with the motion tracking middleware called NiTE allow their users to access natural interaction devices. The OpenNI SDK provides support for:

- Voice and voice command recognition
- Hand gestures
- Body motion tracking

Once we had all these drivers installed, we found out that we would not be able to keep going this way due to the incompatibility of these drivers with Unity, which is restricted to the SDK if you want to work with Kinects.

So, after facing all these problems, we decided to change our first plans, as we needed two different computers if we still wanted to work with two Kinects and we decided to establish an OSC connection between the two computers in order to be able to transfer data. So, in this new scenario, we have two computers connected to each other via an Ethernet cable. This way, we can work with two Kinects in the same project and reach our goals.

### 5.3. Relationship between classes

As in the following sections pieces of code will be showed to explain part of the project, firstly a definition of the classes and their relationship is needed.

Before starting with the definition of these classes, to find the codes in both computers it is necessary to go to a folder called ‘Source’ located in the Assets folder of the Unity project.

Starting with the main computer, the one that will be running the whole project, in this folder called ‘Source’, four c# scripts and another folder will be there. From all of these c# files, the most relevant ones are the ones called ‘OSCHandler’ and ‘UserPosition’; the one called ‘CubeClone’ refers to the particle system and the resting one was made to test the OSCConnection connection between the computers.

Regarding the most important ones, the following tables show how each class is formed by their methods and also the functioning of each of them.
The two tables below describe the scripts ‘UserPosition’ and ‘OSCHandler’. As it can be seen, the class UserPosition needs methods from the OSCHandler class in order to work at its fullness. Concretely, the first class uses the following methods:

- Init(), in order to initialize the establishment of the future connection.
- CreateServer(), to create a new server
- UpdateLogs(), to constantly update all the clients and servers at every frame.

On the other hand, the second computer, which is in charge of sending messages when the player is in its zone. In this case, to find the scripts used there is a folder in Assets
called ‘Source’. Once in this folder, there will be the two scripts that this computer uses called ‘OSCConnection’ and ‘OSCHandler’. The ‘OSCHandler’ script is exactly the same as the one used in the main computer.

As regards to the script called ‘OSCConnection’, the following table shows how it is formed.

<table>
<thead>
<tr>
<th>OSCConnection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variable definition</strong></td>
</tr>
<tr>
<td><strong>Methods:</strong></td>
</tr>
<tr>
<td>• Start():</td>
</tr>
<tr>
<td>o Establish connection with the other computer</td>
</tr>
<tr>
<td>o Creates a new client</td>
</tr>
<tr>
<td>• Update():</td>
</tr>
<tr>
<td>o Update client</td>
</tr>
<tr>
<td>o Send a message</td>
</tr>
<tr>
<td>• sendMessage():</td>
</tr>
<tr>
<td>o send a message to the other computer</td>
</tr>
</tbody>
</table>

In this case, this class uses the methods from ‘OSCHandler’ to create a new client, to update the clients at every frame and to send a message to the other computer. Additionally, in order to have a more structured code, the method sendMessage() is called at the Update() method.

### 5.4. OSC Connection between Kinects

OSC, which means Open Sound Control, is a UDP protocol for communication among computers and other multimedia devices that is optimized for modern networking technology. OSC’s advantages include interoperability, accuracy, flexibility and enhanced organization and documentation.

In our case, as we will finally have two different computers, we can divide this OSC connection in what the two computers will do: the main computer and the computer used for transferring the data of the second Kinect.

Firstly, the main computer will have the entire project running. This computer, which has also a Kinect connected, is the one that we will use to interact with the system and the one that will also be connected to the two projectors in order to display it on the floor. We can say that this computer, in terms of the OSC connection, will be acting as a server since it will constantly be listening what the other computer is sending when the second Kinect has detected either one or two players.

On the other hand we will have the second computer that will also be connected to the resting Kinect and that will only be used to transfer data. In this case, this computer will only be used for the task mentioned before every time a user is detected in its zone. We can affirm that, in the OSC connection, this computer will be the client as it will be

Development of a full-body interaction video game for children with Autism

33
sending the data extracted from the Kinect to the other computer, and once sent, we will be able to see the user’s movements in the other computer and start treating this data.

Once we have done all this differentiation, we can now step forward to establish the connection of the two computers. To do so, we first have to connect both computers via an Ethernet cable and as we are not connected to Internet anymore, we can set each computer an IP address and a port. In our case, we decided to set the following IP addresses to the computers:

- Server: 192.168.1.5 port: 6666
- Client: 192.168.1.6 port: 6666
  - As regards to the mask of both, the client and the server, it was 255.255.255.0

To sum up all this idea of the OSC, it can be represented in a figure, where it will be seen how two computers, thanks to an Ethernet cable are connected and can transfer data between each other.

![Illustration 30 OSC Connection between the two computers](image)

It is very important that both computers are on the same network; otherwise the connection will not be possible. Also, we have to check if the ports are correct because if they are different, the connection might be correctly done but the server will be expecting to receive a message from a port and the client will be sending a message to a different port.

To check if the connection was correctly established, there is a Windows program called *Hercules* that lets you be a server or a client depending on your needs and see if you receive messages from the client.
5.5. Sending and receiving OSC messages

Once the connection is correctly established we can start coding in Unity to start sending and receiving messages. To do so, we have used a set of scripts from GitHub [15] called UnityOSC. This library comprises the following classes:

- OSCBundle
- OSCMessage
- OSCPacket
- OSCClient
- OSCServer
- OSHandler

This last class, OSHandler, it could be said that it is the main class, and the one that we will use in order to create the client and server and also to send and receive messages.

In order to start sending and receiving messages we have to create the client and server and specify the IP address and port that we want the client to send the message and the port from where we want the server to listen.

First of all, we have to call the OSHandler initialization like this:

```csharp
OSCHandler.Instance.Init();
```

An important thing to take into account is to know and differentiate that we currently have two different computers and, consequently, we will have to write two different codes, as the computers will be behaving differently.

After the OSHandler initialization and firstly focusing on the server we can now start with its creation, in which we will have to type the next code line:

```csharp
OSCHandler.Instance.createServer("server1", 5555);
```

Note that, as mentioned before, we need to specify the server name and the port from where the server will be listening.

After creating the server, we now have to call the UpdateLogs() method to read the incoming messages, which will be done thanks to the next code line:

```csharp
OSCHandler.Instance.UpdateLogs();
```

Before starting to listen to the messages sent by the client, we also have to create a dictionary of our server in order to be able to process our received data. This will be useful in our case to filter our certain messages and to trigger or control events in our application. The creation of the dictionary is done like the following:

```csharp
Dictionary<string, ServerLog> servers;
```

// The following line has to be placed in the very beginning of the class creation
// The following line has to be placed in the Init method
servers = new Dictionary<string, ServerLog>();
// The following line has to be placed in the Update method
servers = OSCHandler.Instance.Servers;

After typing all these code lines we are done with the creation of our server and now, before explaining how to treat the received messages, we will explain how to create the client and how to send the messages.

To start, after also initializing the OSCHandler by calling the Init() method, we now can create the client by doing the following:

```csharp
OSCHandler.Instance.CreateClient("client1", IPAddress.Parse("192.168.1.5"), 6666);
```

As we can see, we have to specify the client name, the IP address where we will be sending the message, also known as the server IP address, and the port to where we are sending the message.

In this case, we can now proceed to send the message. In our project, we send the following message:

```
OSCHandler.Instance.SendMessageToClient("client1", "message", "P1/" +
"LF" + footLPosition.transform.position.ToString () + "/" +
"RF" + footRPosition.transform.position.ToString () + "/" +
"RH" + handRPosition.transform.position.ToString () + "/" +
"LH" + handLPosition.transform.position.ToString () + "/" +
"HH" + headPosition.transform.position.ToString () + "/" +
"TT" + torsePosition.transform.position.ToString () + ");
```

So, this is the message sent by the client. As we can see, we can divide the message in two different parts:

- The part where we differentiate what player’s data we are sending, which corresponds to the very first part of the message, and it can be either “P1” or “P2”.

---

Table 1 Table showing how messages are sent and received.

<table>
<thead>
<tr>
<th>Message sent</th>
<th>Message received</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1/LF(2.4, 0.2, 3.6)/RF(2.6, 0.2, 3.6)/RH(2.7, 0.9, 3.7)/LH(2.3, 0.9, 3.7)/HH(2.5, 1.6, 3.4)/TT(2.5, 1.1, 3.6)</td>
<td>P1/LF(2.4, 0.2, 3.6)/RF(2.6, 0.2, 3.6)/RH(2.7, 0.9, 3.7)/LH(2.3, 0.9, 3.7)/HH(2.5, 1.6, 3.4)/TT(2.5, 1.1, 3.6)</td>
</tr>
</tbody>
</table>
• The part where we send the position of the selected body parts so the server can know where the player is.

Regarding to the second part of the message, we do not send all the body parts of the avatars as that would be too much information and it would take a lot of time to treat it all. This is why selected six different parts of the body with which we still can create a very simple human shape avatar.

To replace the avatar to this simple human shape we have attached six different game objects to the following body parts:
  • The two feet
    o LF
    o RF
  • The two hands
    o LH
    o LH
  • The torso
    o TT
  • The head
    o HH

This way, while the user is being tracked, we extract the position of each game object and put it all together in the same message with a corresponding code (as we can see in the above numeration) and separated by a slash. Thanks to the codes assigned to the game objects and the separation slash, the server can differentiate what part of the body corresponds to each coordinates sent by the client and display the player’s position all along the playground.

In the next figure we can see how the server receives the messages:
The figure above shows how the messages are received from the main computer. So, as soon as a player is detected by the second Kinect, which is the one that does not have the whole project running, it starts sending messages to the main computer in order to be able to know all the time the position of the user.

In the following figures, we can see the difference from the avatar given by the Kinect package that we are using and our final six game objects avatar.
As we can see, the avatar that we use to send the body parts (the right one) is a very simple copy of the avatar that the package provides. However, as we did not need to display any avatar in our videogame, we just need to know the current position of the players, we do not need to use all the parts of the avatar and this way we only transfer the necessary information so the data treatment is not affected by the amount of messages sent.

If we now switch to the main computer and analyse the way that it receives the data, it will start identifying what player is being tracked. After this, we can now start to assign each body part to its respecting game object and this will be displayed in the screen and, as we are now in the main computer, it will be able to start interacting with the whole system.

Thanks to the messages sent, the main computer can display each of the six positions of the body parts at the exact time and make look like that the two Kinects are connected to the same computer. Furthermore, we can see how the second computer captures the user, and how the same user, now in the main computer, is interacting with the whole system.

5.6. Creating a solid group of data from both Kinects

Once the OSC connection is done and we have the two computers sending and receiving messages respectively, it is now time to create a solid group of data from both Kinects in order to be able to manage the whole system from the main computer.

This part is more important than it seems because if we do not create a solid group of data and that is easy for us to treat, then we will have difficulties in order to manage the received data.
The first thing in this part of the project is to have everything clear; we do have to know that we will have a computer running all the project and receiving messages from the other computer and then we will have the second computer that will be sending the information from the Kinect. So, the more separated and specified information that we send, the easier will be for the main computer to work with that data. This is why we are sending messages specifying what player is being captured and his six main body positions.

5.7. Distinction of both users

Now we will see how the user distinction has been done. Before that, though, we will explain how a single user is detected.

So, to start, we had to download all the software explained in the section 2.1 Connecting the Kinects; and once everything is installed, we will only have to place the player inside the Kinect detection rank and he will be displayed in the screen.

After both computers are able to detect the players, we can now step forward to the user distinction. In this case, this part is also important because as we have two Kinects and two computers connected, we have to create a solid way to distinct the players in order not to generate confusion or not to change the users’ avatar in our system. This way, we have thought the user distinction in the following way:

As we can see in the figure below, we have generated four different skeletons that will be the ones to imitate the players’ movements. Furthermore, we can see that the playground has been divided in two different parts corresponding to each Kinect’s detection areas. Additionally, each player will have assigned two different skeletons,
one for each zone and when one of the users is being detected in one zone, then his assigned skeleton of that zone will be displayed in the main computer while the other skeleton will be hidden. Focusing on the zone distinction, it can be seen that there are two lines in two different colours; these lines represent the time when one skeleton from one zone will switch to the corresponding skeleton of the other zone. This way, as we can see the skeleton one and two (the blue ones) will be displayed in the main screen until they blue line but, as soon as they cross the blue line, then its corresponding red skeleton will be enabled. The same will happen with the red skeletons, as long as they do not cross the red line, they will be displayed.

It looks like there could only be one line to determine what skeleton to display, but then if a user would be walking through that crossing point for a long time, the system would not know what skeleton to display; it could be say that we would be generating a critical point. This way, when a skeleton crosses its line, then it will have to go to that skeleton crossing line to change again and the system is not constantly changing the skeleton if the user stays in that critical point for a long time since it is not a critical point anymore.

Now that these two lines are being part of the system, a more graphical way to see the different situations that can occur in the game can be shown in the following figures:
As it can be seen previously, the possible situations during the game are:

- Both players in the same zone
- One player to each zone

Now, talking about the programming side of this part, we firstly have to take into account some aspects in order to have our system totally working fine and avoid possible bugs:

- The maximum amount of skeletons that can be displayed during the whole game is two.
- If one player is being detected in a zone and the other one is being detected in the other one, the two different skeletons have to be displayed as well.
- If both players are being detected in the same zone and one of them goes to the other one, the correct skeleton has to change.
• If there is only one player in the playground area and the second one enters from the other zone, the right skeleton has to be chosen.

In order to accomplish the above specifications, we have done a series of verifications and we will now explain them by parts.

Before starting with all the verifications, and as stated before, we have divided the playground in two different zones in order to select the correct skeleton all the time. In addition, all the following verifications have been done regarding the user’s position. To make things more clear for the following explanations the following terms will be defined:

• Zone 1: between 0 and 3 meters of the playground area. Note that these are the z-coordinates as it is referring to the depth position.
• Zone 2: between 3 and 5 meters.
• Kinect1: Kinect that detects the players of the first zone.
• Kinect2: Kinect that detects the players of the second zone.
• Skeleton1: corresponds to the skeleton attached to the first player when this one is in the first zone.
• Skeleton2: corresponds to the skeleton attached to the second player when this one is in the first zone.
• Skeleton3: corresponds to the skeleton attached to the first player when this one is in the second zone.
• Skeleton4: corresponds to the skeleton attached to the second player when this one is in the second zone.

To start, the first possible case is to have only one player playing. To check his position we are constantly checking the position of his attached game object, corresponding to the avatar. Also, the position of the other skeleton attached to the same player but corresponding to the other zone is checked. To do so, we have used the following code:

```csharp
if (skeleton1 is detected in the first zone AND skeleton3 is detected in the first zone)
{
    Enable skeleton1's mesh renderers;
    Enable skeleton1's colliders;
    Disable skeleton3's mesh renderers;
    Disable skeleton3's colliders;
}
```

As we can see, if the player one is in the first zone, then we enable all the skeleton1’s mesh renderers and colliders because this skeleton is the one detected by the first Kinect. Furthermore, we also disable all the skeleton3’s mesh renderers and colliders because there can be some areas of the playground that both skeletons can be detected by their respecting Kinects so we have to make sure that the proper one is being displayed.
In the next figure we can see the first player being detected in the first zone:

Contrary to the first case, there is also the other case if there is still only one player playing: the case that the player crosses the 3 depth meters and the second Kinect starts sending messages. To do so, the code used is the following:
if (skeleton1 crosses to the second zone) {
    Disable skeleton1’s mesh renderers;
    Disable skeleton1’s colliders;
    Enable skeleton3’s mesh renderers;
    Enable skeleton3’s colliders;

    // start treating the received messages
    foreach (item in the servers Dictionary) {
        if (we have received one message) {
            Assign the last received message to a variable
            Assign the received message to a variable
            if (this player is the first one to cross to the second zone) {
                Assign each body position to the corresponding game object
                Set that this player has been the first one to cross
            }
        }
    }
}

So, in this case, two different parts can be separated.

The first one, corresponds to the part of enabling and disabling the correct skeletons; in this case it is the other way round respecting the first code described before: we disable the skeleton1 and we enable the skeleton3.

The second part is a bit more complex: treating the data received from the other computer. As now the skeleton that is being displayed in the main screen is the one detected by the second Kinect, we have to treat the messages received by the other computer. To do so, we first check that it is the first player that has crossed to the second case, which in fact, as there is only one player detected, it is. Once checked this part, it is now time to set the position of the body parts sent via OSC to their corresponding game objects. In this part, there is only one issue, the position of a game object is a Vector but what we are receiving is a message. To convert the string received to the corresponding vector we have created the following method.
Vector3 stringToVector(string sVector)
{
    if (the message starts with “LF” or “RF” or “LH” or “RH” or “HH” or “TT” AND finishes with “)”) {
        Do no count the first 3 characters and the last one of the message
    }
    Split the message by its commas
    Declare new Vector corresponding to the position
}

As we can see in this function, we first get rid of the first 3 characters and the last one because they are not necessary anymore and the final result is meant to be the position that the game object has to be placed. After this verification, we divide the message in three different parts, corresponding to the three coordinates (x, y, z) and finally create the new vector that will be the new position.

After having set all the sent position to their respecting game objects, there is only one more thing missing. We have defined a variable to check what player is the first one to cross from the first zone to the second one. This is very important because it is the way used to differentiate if the sent message is for the first player or for the second one, i.e., this variable was created to be used when both players are in the second zone. For instance, when there were both players in the second zone, how does the main computer assign to each player the correct message corresponding to their current position if there is no variable in charge of this action? And this is where this new variable turns out to be important. This variable defines what player has been the first one to cross to the second zone so when the second player gets in the second zone, the program knows for what player is each message that receives.

Thanks to this variable, we now know that, in the case that both users are in the second zone, the message containing “P1” will be assigned to the first player and the message containing “P2” will be assigned to the second player. Note that this does not happen when both players are in the first zone because the Kinect can handle this by itself but since we are only sending the body positions, a different way has to be implemented for this case.

In the following figure the first player is being detected in the second zone, and the body is moving thanks to the received messages:
Now, the second player takes place in the scene and we also have to do the same verifications as for the first player and, depending on where the second player is positioned either the skeleton2 or the skeleton4 will be enabled.

In the following figures, the two players can be seen in different positions and with different skeletons enabled.

All these series of verifications are also following a general block diagram which states all the possible situations that had to be taken into account and how the user distinction had to be done in a proper way. The block diagram can be seen in the following figure:
Development of a full-body interaction video game for children with Autism

Illustration 43 User distinction block diagram
Now that two players are in the scene, it is time to define how the received messages are differentiate in the case that two players are walking in the second zone.

As mentioned before, a variable has been defined to distinguish what player has been the first one to cross from the first zone to the second zone. However, there are more aspects to take into account as regards to this case. In the following code, it can be seen what kind of comparisons and verifications are done in order to assign the received messages to the correct skeletons every time.

```python
if ((player is the first one to cross to the second zone) AND received message address contains "P1") {
    Assign to the variable that this player has been the first one;
    Set the positions to corresponding game objects;
}
if (player is the second one to cross) AND received message address contains "P2") {
    Set the positions to corresponding game objects
```
As we can see in the code before, the series of verifications can be divided in two different sets:

- Check if the player has been the first one to cross to the second zone.
- Check what message has been received.

Starting for the first part, to check if the player that has crossed has been the first one to do it or not we check two things. The first thing to check is the position of his assigned skeleton of the first zone (for instance, in the case of the first player the position of the skeleton1 would be checked) and then we check what is the value of the variable created: true or false. If the value of the variable is true, then it means that there is already one player in the second zone, otherwise, it means that the player is the first one to cross.

Finally, this variable will be set to false again when the two players are in the first zone, so all the verifications can start again when one player crosses to the second zone.
6. PHASE 4: INTEGRATION OF THE GRAPHIC CONTENT AND PROTOTYPE

Once the system is totally configured, in order to create a functional and definitive version of the game the graphic content has to be added.

It could be said that the graphic content is the key to connect the work done by the team in order to have a functional prototype and the fact that the players will now be able to play and interact with the whole system.

So, as mentioned before, in this report the graphics and images used are only to be able that the system works, but it is not definitely the last version. In this case, in order to have a proper version of a prototype professional designers have been contacted and it is also one of the missing things to be done in a close future.

The designers task will be to design the whole system that will be seen by the players. This part of the work is also very relevant since it will be what the children with Autism will watch and it has to be designed in order that they do not get tired of the game soon or that they play the game properly.

For instance, since one of the objectives of the projects is to improve the children’s social and interactive behaviours, the graphic content will take an important part in making the children act like planned. One of the examples is that, to emphasise the fact that they have to interact and communicate together, the graphic content will be in charge of getting the children close to each other in order that they understand that they have to interact between each other and to see the new features of the game.

In a certain way, the graphic content is the final part of the process and in charge of making the children understand the game since it will be the part that they will see. This is why it is a very important part, because it has to transmit what it is really planned and not generate any confusions. For instance, if one of the players comes up with a bad character, the graphic content has to make look like that character as bad so the player knows that it is bad, otherwise the players will not be able to play the game.

Another fact that is really important is that the game is destined for children with Autism and that these children do not have the same perceptions as children who do not suffer this.

In order to succeed in this part of the project, the opinion of the children who will end up playing the game is very important and their collaboration has an infinite value. So, taking information of what they think about the game and what would they change in order to obtain better results and in order to define the narrative and interaction content.

Another thing that can be done, and which it has previously been done in Lands of Fog is to conduct sessions with high functional Autism children to have a feedback about the game. For example, in one of the sessions the children can have the opportunity to
design the environment through a college activity. Another session could also be to ask the children what creatures or characters they would like to inhabit in their game; this way, the children will want to play with the game to see the creatures that they have requested. Additionally, another session could also be to ask the children what actions they would like their creature to do; this way, the children would define the behaviour of their creatures and we would make sure that they understand and express what they really think.

Conducting all these sessions is a powerful tool to start showing the children what they will be playing with in a close future and a way to start introducing the children the game so they can adapt faster to the content and they can focus on playing with the game.

Another thing to take into account is the physical interface. In this case, the playground area has to be big enough so they have their own space and they do not feel that somebody is invading their personal zone. On the other hand, the dimensions have to make them think that they possibly have to interact between each other.

As regards to the shape of the playground area, a thing to consider is that it should be a circle or, at least it should have a rounded shape, in order to avoid the vertexes, which are critical zones in children with Autism because they can isolate themselves.
7. ADAPTIVE VERSIONS

Once all the system was implemented and working properly, it was time to project it and see how the game was being displayed.

One of the main concerns of the project at this point was the dimension of the projection. Using two projectors facing each other, provided us quite a big playground area. This is pretty good for our objectives, as the players will have a big enough area so they will have their own space and they will also suppose that they have to interact between each other.

However, since this system is meant to be placed in hospitals and classrooms, the possibility that there is not enough space to project the whole system exists. For instance, not all the classrooms of a school are that big so they can handle such a big projection (note that the space needed to use all the system, including the structures and the projection, would end up being 8 meters approximately). To solve this problem, a second version is available as well.

In this second version, a smaller projection is used; only one projector is being used and the dimensions of the playground area are reduced at its half. Since the projection is smaller, only one Kinect is also used because it can detect the users in the entire playground area.

The functionalities of the game will be exactly the same; the only difference is that now the court will be smaller. This way, classrooms that do not have space enough to place the two projectors will be available to make profit of the system.

In terms of technical changes, the code will now be much more simple, since the user distinction is not user in this case, neither the OSC connection between the two computers.

As respects to the Unity configuration, only one camera will be used, since we are using only one projector. Focusing on the Kinect calibration, it will be the same as the projector’s configuration; one camera will be attached in Unity to know where the camera is placed.

To sum up, the only change in this version is the dimension of the playground area and that only one projector and one Kinect are being used.

In the following figure it can be seen how the projection would end up being in this case.
As it can be seen, in this case the projector projects the entire playground area.

In terms of user distinction, the Kinect can handle that itself, so it is not necessary complementary code in this version.

In the following figure it can be seen the projection on the floor:
Development of a full-body interaction video game for children with Autism
8. CONCLUSIONS

8.1. Results

In terms of the results from the system, it can be observed that the main objectives of the project have been positively achieved. It is true that in some of the parts of the project some future work will have to be done, whereas in some other parts, the system is perfectly working and no extra work will have to be done.

To start, regarding to the Unity particle system, a solid particle system has been implemented and improved respecting to the one provided by Unity. Currently, with the new particle system the particles can collide between each other and with extern game objects. Also, any shape and material can be chosen for the particles.

Another important objective of the project has been the multiple Kinect compatibility. In this part of the project some problems have been faced and solved with good decisions and good results. The connection of the Kinects between two different computers via an OSC connection is established and working perfectly. It is true that it was not the first idea to do it with two computers but due to the incompatibility of the Microsoft SDK and the connection of multiple Kinects in a same computer this has been a good solution. Although this aspect has not completely been evaluated, it is true to say that a set of solutions for doing a multi-kinect system and multi-projector projection have been provided, as well as the mapping between them to provide a large interactive space.

Another new implementation has the projectors in the leading role. Thanks to new updates of Unity, two projectors are attached to the system and all together with their respecting projections they are projecting the entire playground area. This will be pretty useful for future work because everyone that need to work with two projectors attached in the same project will know how to do it, and no extra program will be needed.

Lastly, the user detection has also been implemented. This aspect of the project could be known as the one that puts all the mentioned results together. The user detection is working quite well. The system can detect up to two different users and they can walk around with their respecting avatars. Although most of the implementation is working fine, there are still some things that could be improved, such as the way that the skeletons are switched when a user crosses form one zone to the other, specially when the user goes from the main zone to the second one. For a future work, and in order to leave this aspect done as well, it is said that a solid way to select a skeleton for each player when this one crosses from one zone to the other one has to be implemented.

Another important achievement is that the system is very portable. With this, as it was thought to be placed in the classrooms, it will be very easy for the teachers to move it and place it where it suits the best.

One more important result is that we have created an independent system and it can also be used separately. For instance, the projectors configuration in Unity can be used by
anyone who needs it without the need to use the entire system. Another example is the particle system, which can also be used in a completely different project if it accomplishes its needs.

8.2. Future work

Respecting to the future work that needs to be done, it could be considered that the system is a first working version. The good thing of this system is that it has been made in a very general way. With this, it will be easy for the people who want to use this project as a base to start with this system and get their own customized system. For instance, they way that the particle system has been implemented gives the future worker of this project to be able to create a series of game objects with the necessity of only creating one.

For a future work, as stated in the previous section, there is the way to select a skeleton when a player crosses form the first zone to the second one. It is needed to implement a more solid way to do this since now it is working perfectly fine for one user but when the second player gets in the scene there are some situations where it gets a bit confused with the skeleton selection. Nonetheless, the system can still be used without problems.

As it has also been stated before, a very independent project has been achieved and with this it means that people will be able to use only the parts of the project that they need for theirs. This way, it will be available for projects that maybe do not have anything in common with ours.

8.3. Personal evaluation

In a personal valuation, forming part of this project has been a very positive personal experience as well as professionally talking.

A part from the positive that I retrieve to the overall project; the development of the entire project has opened up a new world full of opportunities that I did not think about before getting into this team. It is not only the fact of gaining a lot of experience in the programming field, concretely in Unity and C#, it is also that I will most certainly continue studying in such a field: videogames. This is why I have to thank for the opportunity and responsibility that I have been given to value all this experience as a very positive and rewarding one.

The fact of working in a project where other people were also enrolled has made me learnt lots of things from all of them. Working in such a team full of professionals that I could only learn from them, and the way they have treated me, is one of the most valuable aspects that I take from this project.

On the other hand, I am also delighted to say that I have been capable of applying most of the learnt concepts during my three years of career and also to learn a huge variety of
new knowledge. Not only that, the fact of developing this project with the aim that after my work others will still use it and make the best out of it, produces me a big and positive personal satisfaction.

9. BIBLIOGRAPHY

9. EPSON EB-430 Data Sheet
15. Garcia, J. *Unity OSC.* GitHub: https://github.com/jorgegarcia/UnityOSC
17. Particle System. Unity documentation: https://docs.unity3d.com/Manual/PartSysWhatIs.html
Development of a full-body interaction video game for children with Autism
10. **APPENDIX**

10.1. **Code**

Respecting to the code, the implemented code can be divided in two, as regards that we have two different computers.

First of all, the code implemented in the main computer is the following:

```csharp
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class UserPosition : MonoBehaviour {
    // Declare servers dictionary
    Dictionary<string, ServerLog> servers;

    // Declaration of the four different skeletons
    public GameObject skeleton1;
    public GameObject skeleton2;
    public GameObject skeleton3;
    public GameObject skeleton4;

    // Declaration of the body parts
    public GameObject leftFoot;
    public GameObject rightFoot;
    public GameObject rightHand;
    public GameObject leftHand;
    public GameObject head;
    public GameObject torse;

    public GameObject leftFoot1;
    public GameObject rightFoot1;
    public GameObject rightHand1;
    public GameObject leftHand1;
    public GameObject head1;
    public GameObject torse1;

    public GameObject leftFoot2;
    public GameObject rightFoot2;
    public GameObject rightHand2;
    public GameObject leftHand2;
    public GameObject head2;
    public GameObject torse2;

    public GameObject leftFoot3;
    public GameObject rightFoot3;
    public GameObject rightHand3;

    // Code implementation
    void Start() {
        // Initialization of servers dictionary
        servers = new Dictionary<string, ServerLog> {
            {"user1", new ServerLog() {
                Name = "User 1",
                Age = 10,
                Log = "User 1 logged in"}}
        };
    }

    void Update() {
        // Code
    }
}
```

Development of a full-body interaction video game for children with Autism

61
public GameObject leftHand3;
public GameObject head3;
public GameObject torso3;

//Declare the center zone of the playground area
private float center = 2.5f;

//Boolean to know what is the first player crossing the "switch skeleton" zone
public bool player1Detected = false;
public bool player2Detected = false;

// Use this for initialization
void Start () {
    //Initialize the OSCHandler
    OSCHandler.Instance.Init();
    servers = new Dictionary<string, ServerLog>();

    //Create new server
    OSCHandler.Instance.CreateServer("server1", 6666);

    //Start with all the renderes and colliders turned off
    MeshRenderer[] myRenderers1 = skeleton1.GetComponentsInChildren<MeshRenderer>();
    enableMeshRenderer(false, myRenderers1);

    SphereCollider[] mySphereCollider1 = skeleton1.GetComponentsInChildren<SphereCollider>();
    enableColliders(false, mySphereCollider1, torso.GetComponent<CapsuleCollider>());

    MeshRenderer[] myRenderers2 = skeleton2.GetComponentsInChildren<MeshRenderer>();
    enableMeshRenderer(false, myRenderers2);

    SphereCollider[] mySphereCollider2 = skeleton2.GetComponentsInChildren<SphereCollider>();
    enableColliders(false, mySphereCollider2, torso1.GetComponent<CapsuleCollider>());

    MeshRenderer[] myRenderers3 = skeleton3.GetComponentsInChildren<MeshRenderer>();
    enableMeshRenderer(false, myRenderers3);

    SphereCollider[] mySphereCollider3 = skeleton3.GetComponentsInChildren<SphereCollider>();
    enableColliders(false, mySphereCollider3, torso2.GetComponent<CapsuleCollider>());
}
```csharp
MeshRenderer[] myRenderers4 = skeleton4.GetComponentsInChildren<MeshRenderer>();
enableMeshRenderer(false, myRenderers4);

SphereCollider[] mySphereCollider4 = skeleton4.GetComponentsInChildren<SphereCollider>();
enableColliders(false, mySphereCollider4, torso3.GetComponent<CapsuleCollider>());
```

```csharp
// Update is called once per frame
void Update () {
    // Call the OSCHandler to start reading messages
    OSCHandler.Instance.UpdateLogs();
    servers = OSCHandler.Instance.Servers;

    // Declare the skeleton position variable
    Vector3 skeleton1Pos = skeleton1.transform.position;
    Vector3 skeleton2Pos = skeleton2.transform.position;
    Vector3 skeleton3Pos = skeleton3.transform.position;
    Vector3 skeleton4Pos = skeleton4.transform.position;

    // Declare meshRenderers and colliders
    MeshRenderer[] myRenderers1;
    MeshRenderer[] myRenderers2;
    MeshRenderer[] myRenderers3;
    MeshRenderer[] myRenderers4;
    SphereCollider[] mySphereCollider1;
    SphereCollider[] mySphereCollider2;
    SphereCollider[] mySphereCollider3;
    SphereCollider[] mySphereCollider4;

    // The first player is in the main zone, so we enable mesh renderers and colliders of the skeleton1
    // and deactivate the mesh renderers and colliders of the skeleton3
    if (skeleton1Pos.z <= 3 && skeleton3Pos.z < 3) {
        myRenderers1 = skeleton1.GetComponentsInChildren<MeshRenderer>();
        enableMeshRenderer(true, myRenderers1);
        mySphereCollider1 = skeleton1.GetComponentsInChildren<SphereCollider>();
```
enableColliders(true, mySphereCollider1, torse. GetComponent<CapsuleCollider>());

myRenderers3 = skeleton3.GetComponentsInChildren<MeshRenderer>();
enableMeshRenderer(false, myRenderers3);

mySphereCollider3 = skeleton3.GetComponentsInChildren<SphereCollider>();
enableColliders(false, mySphereCollider3, torse2.GetComponent<CapsuleCollider>());
}

// the first player is now in the secondary zone and we enable the renderers and colliders of skeleton3 and deactivate mesh renderers and colliders of skeleton1

if(/*skeleton3Pos.z >= 3 &&*/ skeleton1Pos.z > 3)
{
    myRenderers1 = skeleton1.GetComponentsInChildren<MeshRenderer>();
    enableMeshRenderer(false, myRenderers1);

    mySphereCollider1 = skeleton1.GetComponentsInChildren<SphereCollider>();
    enableColliders(false, mySphereCollider1, torse.GetComponent<CapsuleCollider>());

    // here is where now we have to compare what is the skeleton from the second kinect that is closest to skeleton1
    // and assign that skeleton to the player to see what renderers and colliders we enable: 3 or 4.
    // to compare the distance: Vector3.Distance(a,b)

    myRenderers3 = skeleton3.GetComponentsInChildren<MeshRenderer>();
    enableMeshRenderer(true, myRenderers3);

    mySphereCollider3 = skeleton3.GetComponentsInChildren<SphereCollider>();
    enableColliders(true, mySphereCollider3, torse2.GetComponent<CapsuleCollider>());

    // start treating the received messages
    foreach (KeyValuePair<string, ServerLog> item in servers)
if (item.Value.log.Count > 0)
{
    int lastPacketIndex = item.Value.packets.Count - 1; // Declare last received packet
    string message = item.Value.packets[lastPacketIndex].Data[0].ToString(); // Declare the message received
    string[] finalMessage = message.Split('/'); // Split the message received the /

    // The first player is the first one to enter to that zone
    if ((skeleton2Pos.z <= 3 || player1Detected==true) && item.Value.packets[lastPacketIndex].Address.Contains("P1") /* finalMessage[0].Contains("P1")*/) {
        // Assign each body position received to its respecting game object
        leftFoot2.transform.position = stringToVector(finalMessage[1]);
        rightFoot2.transform.position = stringToVector(finalMessage[2]);
        rightHand2.transform.position = stringToVector(finalMessage[3]);
        leftHand2.transform.position = stringToVector(finalMessage[4]);
        head2.transform.position = stringToVector(finalMessage[5]);
        torso2.transform.position = stringToVector(finalMessage[6]);

        // We might have to do the activation or disactivation of the skeletons here as we first have to compare what skeleton is closer to the actual skeleton
        player1Detected = true;
        Debug.Log("Player 1 first to arrive");
    }

    // The first player is the second one to arrive to the secondary zone
    if (!((skeleton2Pos.z >= 3 && skeleton4Pos.z > 3) && /* player2Detected==true && item.Value.packets[lastPacketIndex].Address.Contains("P2")*/ finalMessage[0].Contains("P2"))) {
        // Start assigning to the respecting game object its position received
        Debug.Log("Player 1 second");
        leftFoot2.transform.position = str
Development of a full-body interaction video game for children with Autism

```csharp
ngToVector(finalMessage[1]);
rightFoot2.transform.position = stringToVector(finalMessage[2]);
rightHand2.transform.position = stringToVector(finalMessage[3]);
leftHand2.transform.position = stringToVector(finalMessage[4]);
head2.transform.position = stringToVector(finalMessage[5]);
torse2.transform.position = stringToVector(finalMessage[6]);
}
}
}
// The second player is in the main zone
if(skeleton2Pos.z <= 3 && skeleton4Pos.z < 3)
{
  // Activate the mesh renderers and colliders of the skeleton2 and deactivate the mesh renderers and colliders of skeleton4
  myRenderers2 = skeleton2.GetComponentsInChildren<MeshRenderer>();
  enableMeshRenderer(true, myRenderers2);
  mySphereCollider2 = skeleton2.GetComponentsInChildren<SphereCollider>();
  enableColliders(true, mySphereCollider2, torso1.GetComponent<CapsuleCollider>());

  myRenderers4 = skeleton4.GetComponentsInChildren<MeshRenderer>();
  enableMeshRenderer(false, myRenderers4);
  mySphereCollider4 = skeleton4.GetComponentsInChildren<SphereCollider>();
  enableColliders(false, mySphereCollider4, torso3.GetComponent<CapsuleCollider>());
}
// The second player crosses to the secondary zone
if(/*skeleton4Pos.z >= 3 &&*/ skeleton2Pos.z > 3)
{
  // Activate mesh renderers and colliders of the skeleton4 and deactivate the ones from skeleton2
  myRenderers2 = skeleton2.GetComponentsInChildren<MeshRenderer>();
  enableMeshRenderer(false, myRenderers2);
```
mySphereCollider2 = skeleton2.GetComponentsInChildren<SphereCollider>();
enableColliders(false, mySphereCollider2, torso1.GetComponent<CapsuleCollider>());

torso1.GetComponent<CapsuleCollider>().enableColliders(false);

myRenderers4 = skeleton4.GetComponentsInChildren<MeshRenderer>();
enableMeshRenderer(true, myRenderers4);

mySphereCollider4 = skeleton4.GetComponentsInChildren<SphereCollider>();
enableColliders(true, mySphereCollider4, torso3.GetComponent<CapsuleCollider>());

foreach (KeyValuePair<string, ServerLog> item in servers)
{
    if (item.Value.log.Count > 0)
    {
        int lastPacketIndex = item.Value.packets.Count - 1;
        string message = item.Value.packets[lastPacketIndex].Data[0].ToString();
        string[] finalMessage = message.Split('/');

        if (((skeleton1Pos.z <= 3 || player2Detected==true) && item.Value.packets[lastPacketIndex].Address.Contains("P1")/*finalMessage[0].Contains("P1")*/) || (/*skeleton1Pos.z>3 && skeleton3Pos.z>3) && */player1Detected==true && item.Value.packets[lastPacketIndex].Address.Contains("P2")/*finalMessage[0].Contains("P2")*/) )
        {
            player2Detected = true;
            leftFoot3.transform.position = stringToVector(finalMessage[1]);
            rightFoot3.transform.position = stringToVector(finalMessage[2]);
            rightHand3.transform.position = stringToVector(finalMessage[3]);
            leftHand3.transform.position = stringToVector(finalMessage[4]);
            head3.transform.position = stringToVector(finalMessage[5]);
            torso3.transform.position = stringToVector(finalMessage[6]);
            Debug.Log("Player 2 first");

            if (/*(skeleton1Pos.z>3 && skeleton3Pos.z>3) && */player1Detected==true && item.Value.packets[lastPacketIndex].Address.Contains("P2")/*finalMessage[0].Contains("P2")*/) )
            {
                Debug.Log("Player 2 second");
            }
        }
    }
}
leftFoot3.transform.position = stringToVector(finalMessage[1]);
rightFoot3.transform.position = stringToVector(finalMessage[2]);
rightHand3.transform.position = stringToVector(finalMessage[3]);
leftHand3.transform.position = stringToVector(finalMessage[4]);
head3.transform.position = stringToVector(finalMessage[5]);
torse3.transform.position = stringToVector(finalMessage[6]);

if(skeleton1Pos.z <= 3 && skeleton2Pos.z <= 3)
{
    player1Detected = false;
    player2Detected = false;
}

//Function to enable or disable the mesh renderers
public void enableMeshRenderer(bool action, MeshRenderer[] myRenderer)
{
    foreach (MeshRenderer GOrenderer in myRenderer)
    {
        GOrenderer.enabled = action;
    }
}

//Function to enable or disable the colliders
public void enableColliders(bool action, SphereCollider[] mySphereCollider, CapsuleCollider myCapsuleCollider)
{
    foreach (SphereCollider GOcollider in mySphereCollider)
    {
        GOcollider.enabled = action;
    }
    myCapsuleCollider.enabled = action;
}

//Function that converts from string to Vector
Vector3 stringToVector(string sVector)
{
    {
        sVector = sVector.Substring(3, sVector.Length - 4);
    }

    string[] sArray = sVector.Split(',');

    Vector3 result = new Vector3(float.Parse(sArray[0]), float.Parse(sArray[1]), float.Parse(sArray[2]));
    //Vector3 result = new Vector3(-float.Parse(sArray[0]), 0, float.Parse(sArray[2])); //we inverse the x-coordinate because we want a mirror effect as the kinects are facing each other

    Debug.Log(result);

    return result;
}

On the other hand, the code used in the computer that has been acting as a client is the following:

using System.Collections;
using System.Collections.Generic;
using System.Net;
using UnityEngine;
using UnityOSC;

public class OSCConnection : MonoBehaviour {
    //Declare servers Dictionary
    Dictionary<string, ServerLog> servers;

    //Declare the two different players
    public GameObject player1;
    public GameObject player2;

    //Declare body parts that will be sent of both players
    public GameObject footLPosition;
    public GameObject footRPosition;
    public GameObject handLPosition;
    public GameObject handRPosition;
public GameObject headPosition;
public GameObject torsoPosition;

public GameObject footLPosition1;
public GameObject footRPosition1;
public GameObject handLPosition1;
public GameObject handRPosition1;
public GameObject headPosition1;
public GameObject torsoPosition1;

// Declare timer so we are sending 30fps
private float timer = 0f;

// Use this for initialization
void Start () {
    // Initialize OSCHandler
    OSCHandler.Instance.Init();
    servers = new Dictionary<string, ServerLog>();

    // Create new server (just in case we would need to send any messages at any time, even though we don't have to initially)
    OSCHandler.Instance.CreateServer("server2", 5555);

    // Create new client
    OSCHandler.Instance.CreateClient("client1", IPAdress.Parse("192.168.1.5"), 6666);
}

// Update is called once per frame
void Update () {
    // Call oschHandler method to allow us to receive messages
    OSCHandler.Instance.UpdateLogs();
    servers = OSCHandler.Instance.Servers;

    // Increase timer every frame (this is similar to a fixedUpdate method)
    timer += Time.deltaTime;

    if (timer > 0.030f)
    {
        // Call the sendMessage function to send the message
        sendMessage();
        timer = 0f;
    }
}
Vector3 stringToVector3 (string sVector)
{
    //remove parenthesis
    if(sVector.StartsWith("(") && sVector.EndsWith(")")
    )
    {
        sVector = sVector.Substring(1, sVector.Length - 2);
    }
    //split items
    string[] sArray = sVector.Split(',');
    //store new Vector3
    Vector3 result = new Vector3(float.Parse(sArray[0]),
    float.Parse(sArray[1]), float.Parse(sArray[2]));
    Debug.Log("New Position: " + result);
    return result;
}

void sendMessage()
{
    //Start sending messages as soon as we detect a user
    if (KinectManager.Instance.IsUserDetected())
    {
        //foreach (KeyValuePair<string, ServerLog> item in servers)
        //{
        //    //int lastPacketIndex = item.Value.packets.
        Count - 1;
        //In case there is only one player detected
        if (player1.transform.position.z >= 3)
        {
            //Send the message
            OSCHandler.Instance.SendMessageToClient
            ("client1", "P1", "P1/" +
            "LF" + footLPosition.transform.position
            .ToString() + "/" +
            "RF" + footRPosition.transform.position
            .ToString() + "/" +
            "RH" + handRPosition.transform.position
            .ToString() + "/" +
            "..." +
            "LF" + footLPosition.transform.position
            .ToString() + "/" +
            "RF" + footRPosition.transform.position
            .ToString() + "/" +
            "RH" + handRPosition.transform.position
            .ToString() + "/" +
        }
    }
}
"LH" + handLPosition.transform.position.ToString() + "/" + "HH" + headPosition.transform.position.ToString() + "/" + "TT" + torsePosition.transform.position.ToString() + ");

//In case there are two players detected
if (player2.transform.position.z >= 3)
{
    //Send the two messages regarding each player detected
    /*OSCHandler.Instance.SendMessageToClient("client1", "P1", "P1/" +
        "LF" + footLPosition.transform.position.ToString() + "/" +
        "RF" + footRPosition.transform.position.ToString() + "/" +
        "RH" + handRPosition.transform.position.ToString() + "/" +
        "LH" + handLPosition.transform.position.ToString() + "/" +
        "HH" + headPosition.transform.position.ToString() + "/" +
        "TT" + torsePosition.transform.position.ToString() + "/"); */
    OSCHandler.Instance.SendMessageToClient("client1", "P2", "P2/" +
        "LF" + footLPosition1.transform.position.ToString() + "/" +
        "RF" + footRPosition1.transform.position.ToString() + "/" +
        "RH" + handRPosition1.transform.position.ToString() + "/" +
        "LH" + handLPosition1.transform.position.ToString() + "/" +
        "HH" + headPosition1.transform.position.ToString() + "/" +
        "TT" + torsePosition1.transform.position.ToString() + ");
}

//}
Development of a full-body interaction video game for children with Autism
10.2. Particle system code

In order to implement the particle system, a piece of code has also been implemented to clone the game objects. This code has been the following:

```csharp
using UnityEngine;
using System.Collections;
using UnityEngine.Rendering;

public class CubeClone : MonoBehaviour {

    // Attach this script to the object you want to instance, such as a cube object. It should have a mesh renderer on it.
    private int cloneNumber = 20;
    private Vector3[] position = new Vector3[20];

    Mesh mesh;
    Material mat;

    // Make an empty game object and drag it into the obj va variable in the editor. This object's transform will be used as the transform for the instanced object.
    public GameObject obj;

    private GameObject[] clone = new GameObject[20];

    Matrix4x4[] matrix = new Matrix4x4[20];
    ShadowCastingMode castShadows;

    Matrix4x4[] matrix1;
    ShadowCastingMode castShadows1;

    public bool turnOnInstance = true;

    void Start () {
        for (int i = 0; i < cloneNumber; i++) {
            position[i] = new Vector3(Random.Range(-4, 4), Random.Range(-2, 7), Random.Range(-4, 4));
        }

        for (int i = 0; i < cloneNumber; i++) {
            clone[i] = Instantiate(obj, position[i], Quaternion.identity);
        }
    }
}
```
mesh = GetComponent<MeshFilter>().mesh;
mat = GetComponent<Renderer>().material;
matrix = new Matrix4x4[2] { obj.transform.localToWorldMatrix, this.transform.localToWorldMatrix };  
castShadows = ShadowCastingMode.On;
Graphics.DrawMeshInstanced (mesh, 0, mat, matrix, matrix.Length, null, castShadows, true, 0, null);

//matrix1 = new Matrix4x4[2] { clone.transform.localToWorldMatrix, this.transform.localToWorldMatrix };  
//Graphics.DrawMeshInstanced (mesh, 0, mat, matrix1, matrix1.Length, null, castShadows, true, 0, null);

for (int i = 0; i < cloneNumber; i++) {
    matrix1 = new Matrix4x4[2] { clone[i].transform.localToWorldMatrix, this.transform.localToWorldMatrix };  
    Graphics.DrawMeshInstanced (mesh, 0, mat, matrix1, matrix1.Length, null, castShadows, true, 0, null);
}

void Update () {
    if (turnOnInstance) {

        mesh = GetComponent<MeshFilter>().mesh;
        mat = GetComponent<Renderer>().material;
        matrix = new Matrix4x4[2] { obj.transform.localToWorldMatrix, this.transform.localToWorldMatrix };  
        castShadows = ShadowCastingMode.On;
        Graphics.DrawMeshInstanced (mesh, 0, mat, matrix, matrix.Length, null, castShadows, true, 0, null);

        //matrix1 = new Matrix4x4[2] { clone.transform.localToWorldMatrix, this.transform.localToWorldMatrix };  
        //Graphics.DrawMeshInstanced (mesh, 0, mat, matrix1, matrix1.Length, null, castShadows, true, 0, null);

        for (int i = 0; i < cloneNumber; i++) {
            matrix1 = new Matrix4x4[2] { clone[i].transform.localToWorldMatrix, this.transform.localToWorldMatrix };  
            Graphics.DrawMeshInstanced (mesh, 0, mat, matrix1, matrix1.Length, null, castShadows, true, 0, null);
        }
    }
}
10.3. Installation guide

To have all the devices connected to their computers, have all the computers connected between each other and, after that, have all the system working, a series of installations have been done. Since it could be quiet confusing if someone wanted to use the system for the first time, an installation guide needs to be followed.

In this guide, all the necessary steps will try to be explained and so everything will be following a concrete order in order not to generate any confusions for the future users of the system.

The guide can be divided in two great parts: one part that can be considered to be general as it will have to be done in both of the computers and then there will be another part which will explain what extra installations are needed to be done in each of the computers, such as the IP configuration.

GENERAL INSTALLATIONS

As mentioned before, this part of the installation has to be applied in both of the computers.

In this section, it will be explained how to install Unity, how to connect the Kinect in order to be able to work in the computer and what packages are needed to be installed in Unity.

Starting with the Unity installation, it is a simple installation and the steps can easily be followed. In the following enumeration the installation is explained:

- Go to: [https://unity3d.com/get-unity/download](https://unity3d.com/get-unity/download) and choose the desired Unity. To run this project it is not necessary to have a special Unity, so the personal version, which is free, can be downloaded.
- After executing the file and starting the installation, we basically have to follow the indicated steps. At some point, the Windows Firewall will ask the computer administrator to allow the Firewall go through the applications. In this case mark ‘Yes’.
- To check if the Windows Firewall is enabled in our application and working properly, the next figures can be followed:
To access to that particular window the user has to go: Control Panel → System and Security → Windows Firewall.

Once in this window, go to ‘Allow an app or feature through Windows Firewall’. It is the red mark in the figure.

Once the user is in that window, scroll down until seeing the Unity Editor program.

Once there, make sure that all the squares are marked with a tick (✓) and select OK.

With all the steps mentioned before, the entire Unity configuration should be working. Now Unity is installed in our computer and the configuration in order to be able to receive or send messages is also done.
After this, it is necessary to install the Microsoft SDK in order to be able to work with the Kinect in Unity.

To do so, the next steps clearly explain how to download the SDK and how to verify if the Kinect is working.

  - In this page, there is also a section to consult details, system requirements or install instruction if there are some doubts remaining after following the guide.
  - Once clicked to ‘Continue’ a new page will pop-up suggesting the user to register. It is not necessary to register to download the SDK, just select ‘No, I do not want to register. Take me to the download.’

Registration Suggested for This Download

Registration takes only a few moments and allows Microsoft to provide you with the latest resources relevant to

Register and receive the download:

- [ ] Yes, I want to register and receive the download.
- [x] No, I do not want to register. Take me to the download.

[Illustration 52] Registration not necessary to be done to download the SDK

- Once selected that option, the download will start.
- Execute the file and install the SDK.
- Once the SDK is installed, connect the to the power hub and plug the power hub into an outlet. Make sure the USB cable is plugged into a USB 3.0 port, otherwise the computer will not recognize the Kinect.
- After this, the Kinect drivers should be installed. In addition with the SDK, a program called ‘Kinect Configuration Verifier’ has been installed and it can be used to double-check if the system accomplishes all the requirements and if the Kinect is being detected.
- In this part the computer should be able to detect the Kinect.

Once Unity and the Microsoft SDK are installed it is now time to launch Unity and select our project.

However, before starting with the specific computer installation, in the project a unity package has to be imported. Although the package is included in the project and nothing has to be done, the following figures and explanations show how to import a custom package in a Unity project and it can be useful for further purposes.

- To import a project so, a set of actions have to be done. The first one is to select that a custom package is desired to be imported. To do so, the following figure explains how to do it:
Illustration 53 Import a custom package to Unity

- As it can be seen, go to: assets → import package → custom package…
- After selecting the above, a new window will pop up to select the package. Go to the location where the package is saved and select it.
After having the customized package selected, click ‘Open’ and the importation will start.

Before finishing the import, Unity will ask for a final time to import the package. In this window, it can be seen all the files that the package contains. In this new window, select ‘Import’ to finish the import of the package.

**SPECIFIC INSTALLATIONS**

In this part it will be seen how to configure the computer in order to set the IP that we want, also what project to execute in each computer, and how to place the Kinects for each computer.

As two computers are being used, the first one to configure will be the main one, i.e., the one that will have all the project running.

- **IP configuration:**
  - Unplug the Ethernet cable that connects the computer to the internet
  - Plug an Ethernet cable between the two computers
  - Go to Network settings
    - Go to Ethernet
    - Go to change adapter options
    - Select the Ethernet icon
    - Select the IPv4 check
Select the option in order to write the IP manually.
As we are configuring the main computer, this one corresponds to the server.
- IP: 192.168.1.5
- Mask: 255.255.255.0
• Unity configuration:
  o As this is the computer corresponding to the server, the project that has to be running in this computer is called: osc connection server player difference
  o Once the project is opened in Unity, the computer is now ready to run.

Regarding to the second computer, in this case we are talking about the client, the one that will be sending the data from the Kinect that it has attached to the other computer.

The process will be the same until the point of going to Network settings. So now we will start from that point:
  o Go to Network settings
    ▪ Go to Ethernet
    ▪ Go to change adapter options
    ▪ Select the Ethernet icon
    ▪ Select the IPv4 check
    ▪ Select the option in order to write the IP manually.
    ▪ As we are configuring the main computer, this one corresponds to the server.
      ▪ IP: 192.168.1.6
      ▪ Mask: 255.255.255.0
    ▪ Click apply
  • Unity configuration
    o As this is the computer corresponding to the client, the project to be selected will have to be: Connection OSC client newww.
    o Once the project is opened, the computer is now configured.
Open a project:

- When launching Unity, it will make the user select a specific project. At the top-left of the window, there is an option to choose either to start a new project or to open an existing one. Select to open an existing project.
10.4. Manual of OSC Connection

To get started, you have to define first the servers/clients that are going to be used in your project, at the OSCHandler.init() method. Any number of servers/clients can be created, but take into account that the performance would drastically decrease if many clients and servers are running at the same time, for instance, if you use multiple mobile phones or input devices. It is also possible to configure the maximum number of messages that are tracked in the logs (by default 25), by modifying the OSCHandler class attribute:

```
private const int _loglength = 25;
```

Examples of initialization usage at OSCHandler.init()

```csharp
... public void Init()
{
    //Initialize OSC clients (transmitters)
    CreateClient("SuperCollider", IPAddress.Parse("127.0.0.1"), 5555);
    //Initialize OSC servers (listeners)
    CreateServer("AndroidPhone", 6666);
}
...```

Remember that the OSCHandler initialization method has to be called once at any point in your project scripts. A good place can be your Network manager or your Sound manager constructor or initialization method. Do it by just calling:

```
OSCHandler.Instance.Init();
```

You can now send messages to a defined client in two different ways. Take into account that if the client isn't defined, OSCHandler will output a Debug error at the Unity Log console.

```
//Send a single float value to a server
float orientationAngle = 30.0f;
OSCHandler.Instance.SendMessageToClient("AServerName", "address/folder", orientationAngle);
//Send a list of object values.
//Note: you can send whatever values as a combination of floats, ints
//strings, bytes and any type allowed by UnityOSC List
List<object> values = new List<object>();
values.AddRange(new object[]{1.0f, 30.0f});
OSCHandler.Instance.SendMessageToClient("AnotherServer ", "address/folder", values);
```

Beware you must call first the UpdateLogs() from OSCHandler in order to read the
incoming messages that are received at the servers in your application (e.g. In your Update() loop):

```csharp
OSCHandler.Instance.UpdateLogs();
```

After the UpdateLogs() call, then you can process the received/sent messages of a defined server/client and get them by calling the properties of OSCHandler. They return a dictionary:

```csharp
Dictionary<string, ClientLog> clients = new Dictionary<string, ClientLog>();
clients = OSCHandler.Instance.Clients;
Dictionary<string, ServerLog> servers = new Dictionary<string, ServerLog>();
servers = OSCHandler.Instance.Servers;
```

That can be useful to filter out certain messages and trigger or control events in your application, for instance, like the data output coming from the OSC mapping of a kinect device or wiimote controller.

It can be tricky to implement, but you could also dynamically create new servers or define a proxy server to allow different users connect to your application (using mobile phones, for instance).

To obtain more details about the arguments that can be passed to the methods, or how to operate with the library in other ways than the mentioned above, please head to the oxygen html documentation.
This affordable, short-throw, 3,000-lumen projector can be mounted closer to the screen for fewer shadows when presenting.

The EB-430 has optional Wi-Fi and can be mounted 72cm from the screen for a high-quality, 178cm picture with fewer shadows, making it the perfect partner for schools’ existing interactive whiteboards. It can also be used as an alternative to an interactive whiteboard with its optional interactive module.

High-quality picture

Powered by 3LCD technology, the EB-430 achieves accurate colour reproduction, important in education for natural-looking flags, maps and diagrams. A high White and Colour Light Output of up to 3,000 lumens assures a clear picture, even in daylight, while a contrast ratio of 3,000:1 delivers sharp detail.

Interactive lessons

Attach the optional Epson ELP-IU03 interactive module to captivate the class with interactive presentations. The module works with the projector to make any surface interactive - no screen required. Notes or additions can be created and saved with the included digital pen.

Convenient to control

The EB-430 is compatible with the Epson ELPCB01 Control and Connection Box, which puts Epson projector controls within easy reach at the front of the classroom. It also keeps cables tidy.

Engaging presentations

Presentations can be made more engaging by plugging in the Epson ELP-DC06 USB document camera to show 3D objects close-up, or by using the HDMI input to display high-definition content. The built-in 16W speaker and microphone input ensure students won’t ever struggle to hear.

Easy to use
## Development of a full-body interaction video game for children with Autism

---

### PRODUCT SPECIFICATIONS

**TECHNOLOGY**
- Projection System: DLP Technology
- LCD Projector

**IMAGE**
- Colour Light Output: 5000 lumens - 2,100 lumens (accuracy)
- White Light Output: 5000 lumens - 3,100 lumens (accuracy)
- Resolution: HD, 1080 x 768, 6.0

**Lamp**
- Sturdy, 250W, 4000 hours durability, 50000 hours durability (accuracy mode)

**OPTICAL**
- Projection Mode: Manual
- Zoom: Digital, Focal: 1.50
- Image Size: 37 inches - 108 inches

**CONNECTIVITY**
- USB Display Function: 3 in 1: Image / Mouse / Sound
- Interfaces: USB 2.0 Type B, USB 2.0 Type A, RS-232, Ethernet and LAN (100BASE-TX / 100BASE-T), USB (optional), VGA (DVI-D), HDMI, Composite, Component (Y, Pb/Cb, Pr/Cr), Stereo mini-jack audio in, Stereo mini-jack audio out, USB host, Audio in

**OTHER**
- Warranty: 6 months On-site service, Lamp: 12 months or 1,000 hours
- Optional warranty extension available

### LOGISTICS INFORMATION

**SKU**
- Y11H060020
- Y11H060010

**SAN code**
- 8171004550598

**Dimensions Single Carton**
- 762 x 452 x 534 mm

**Carton Weight**
- 15.9 kg

**Multipurpose Quantity**
- 1 units

**Country of Origin**
- China

**Packing Unit**
- 30 UNITS

---

*Warranty offers are not available for all countries. Please contact your local Epson representative for more information.*

For further information please contact your local Epson office or visit www.epson-europe.com.

---

**WHAT'S IN THE BOX**
- VGA cable
- Main unit
- Remote control incl. batteries
- Setup guide
- Software CD
- Power cable
- User manual CD

**OPTIONAL ACCESSORIES**
- Wall Mount - ELPM07
- Wireless LAN Adapter Digtial - ELWAP07
- Interactive Unit - ELPU03
- V12H955220
- Setting Rod - ELPF24
- Casing Mount Mini - ELPM623
- V12HD955232
- Lamp - ELPLP11
- V13H013B61
- Quick Wireless Connect USB Key - ELPA029
- V13H025549
- Control and Connection Box - ELPC601
- V12H955220
- Air Filter - ELPA736
- V13H013A2E

---

Trademarks and registered trademarks are the property of Seiko Epson Corporation and/or their respective owners. Product information is subject to change without prior notice.
10.6. Kinect detection ranks plans

In the following figures more plans about the detection ranks and the disposition of the equipment can be seen.

Notice that the plans are regarding only one projection, as it is completely the same for the resting projector and Kinect.

In the first figure, the location of the Kinect can be seen, taking into account the distance left between the Kinect and the start of the projection in order to detect the entire playground.

In the second figure, the disposition of the projector and the Kinect in the built structure can be seen.
10.7. Defining the number of particles in the particle system

The number of particles that form the particle system can be selected; this way the particle system is variable and can adapt to different needs.

In order to set the number of particles the following process has to be done:

- Go to the Assets folder ‘Source’, where all the scripts can be found.
- Select the ‘CubeClone’ script.
- Change the value of the following four variables by the number of particles desired:
  - cloneNumber
  - position
  - clone
  - matrix