Use of Computer Vision techniques in order to create virtual scenes and characters in a two-dimensional videogame.

Author: Diego Maciá Torregrosa

Tutor: José Ribelles Miguel

Bachelor's thesis for

Video Game Design and Development

29 June 2018

Abstract

This report is part of the Final Degree Project of Video Game Design and Development. Its aim is the use of Computer Vision techniques in order to recognize scenes and characters made by the users and its later incorporation in a videogame. Users will be provided with sheets showing a particular template where both to draw by hand their own characters and to build the scene by placing different platforms or traps. The scene detection will be divided into two parts: background and platforms. Then, users will use a computer camera to detect these elements. Finally, computer vision techniques will be used to extract drawings that will be integrated in a 2D videogame. The last analyzed technique will allow the user to interact with the main menu and even play a mini-game by detecting the position of the user's face.

Key Words

Multiplayer Games, Computer Vision, feature recognition and extraction.

Table of Contents

1. Technical Proposal

1.1 Introduction and project motivation

The use of computer vision techniques can have several applications. The main objective of this project is to expose the uses of different computer vision techniques. These techniques provide many different possibilities that can be applied in the world of video games. Regarding these possibilities, those that can be incorporated into a 2D video game have been chosen. More specifically, its use is proposed in a two-dimensional platformer.

This videogame is used as a testing ground of different techniques. The final goal is not the video game itself, but the testing of the techniques and their results. Thus, the objective is to put into practice the different techniques and their subsequent incorporation in the context of an original video game.

This video game confronts up to four players. Before playing the platformer game itself, users have to draw their characters and backgrounds. Then, this art will be shown while playing the platformer game. Players appear in a start box and they have to arrive to a goal box by placing different platforms between them. This platforms can be placed when it is the turn of a player by using the platform detection. As a result, the scene in which the users will be playing will be completely formed by the implemented computer vision techniques.

Initially, two techniques were considered essential for this project: character and platforms detection. First of all, it is needed to design two different sheets depending on if it is a character or a platform. In the case of a character there will be included several rectangles in which the player will be able to draw. In respect of the platforms, it must be designed a manual framework that allows users to incorporate the different platforms intuitively. Figure 1 shows a low fidelity prototype of these two sheets.

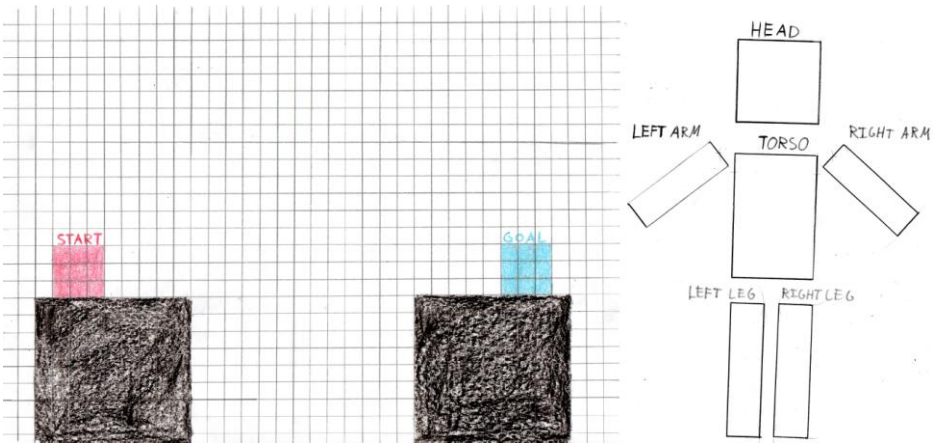

Figure 1: low fidelity prototypes

The background detection will be used to obtain the background drawings that the users will see while playing the game. This technique is similar to the character detection and it should use the same process. Users must draw their background using several sheets that represent the background layers.

The last technique is the face detection. This allows the users to interact with a scene in many ways, such as moving an object that follows the player's face. In this case, the moved object is the sight that is used to select options in a menu and shoot other objects in a mini-game. There is no need to use any type of sheet in this technique.

Therefore, it will be used a Computer Vision library in order to obtain the real world information and include it in a 2D video game developed with a game engine. Three different sheets must be used by players: character, background and platforms. This videogame will provide similar mechanics to *Ultimate Chicken Horse* (Clever Endeavour Games, 2016). Figure 2 shows a screenshot of this video game.

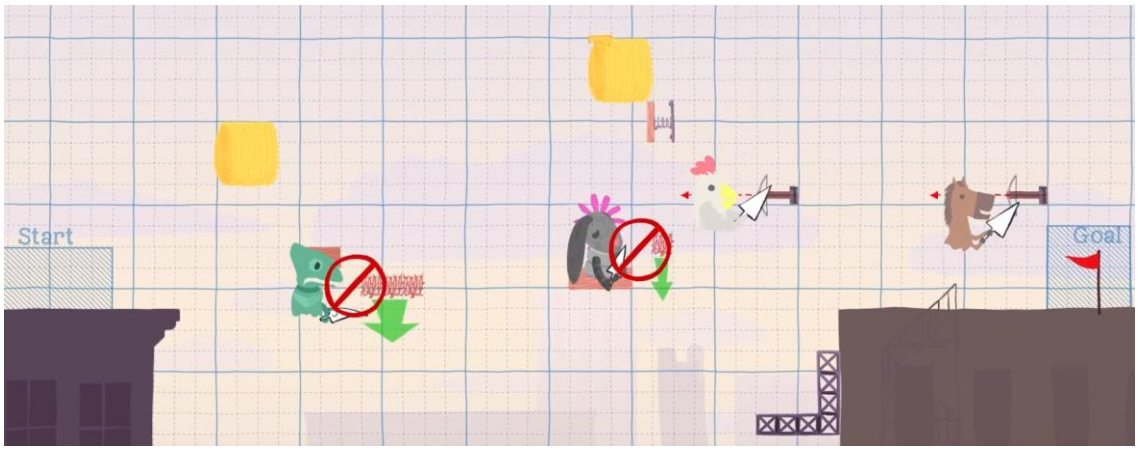

Figure 2: Ultimate Chicken Horse screenshot

1.2 Related subjects

VJ1221 - Computer Graphics, VJ1231 - Artificial Intelligence, VJ1227 - Game Engines.

1.3 Objectives

The objectives will be divided into two: main and feasible extensions. The main objectives are the most important and will be implemented first.

- Main objectives:
    - Use Computer Vision techniques in order to obtain the user designs, both characters and scenes, of the different sheets.

    - Implement a system both to extract drawings and to introduce the obtained data in the 2D videogame.

    - Develop a 2D videogame that allows to show easily how the developed technologies work.

- Feasible extensions:
    - Develop a framework that allows joining what is related to Computer Vision and what depends on the game engine.

    - Analyze the different possibilities that could be included within the game once the technology is developed (e.g. allow players draw scene backgrounds).

1.4 Project planning

Task	Hours
Analysis and design document	20
Report	55
Prepare public presentation	10
Implement character recognition	60
Implement scene recognition	70
Implement game in Unity	65
Design and create players' sheets	20

Table 1: division of tasks

1.5 Expected results

Obtain a system that is able to extract useful information by capturing images from sheets that the players have filled so it can create characters and scenes completely integrated in a videogame automatically.

1.6 Tools

OpenCV, Unity, Visual Studio, Git (TortoiseGit and GitLab).

2. Design Document

2.1 Game Design

The design of the menus and the flow between the different scenes must be intuitive for the player. The first scene contains three phases. First, it will be shown some texts that will introduce the player to the video game. Then, a mini-game will start. The objective of this game is to obtain some points by shooting at missiles and coins. The missiles grant a point and the coins five points. To aim at these targets, users have to move their faces in the direction in which they want to move the sight. Finally, the main menu will be shown when the player obtains the required number of points. Users can select between the different options by using the same procedure as in the mini-game.

This main menu displays three options: play game, create scenery and create player. Players are not allowed to play the game until they have created their player and the scenery. There is no need to create the scenery before the players or vice versa. When selecting any of these options the scene will change. The scenes that are displayed when creating a player or a scenery are very similar.

When entering the create player scene, an add player button is shown. Users can put their character sheet in front of the camera and push this button. If the video game has correctly detected the drawing, it will be displayed in the screen and saved. This button can be pushed up to four times. If one player has been at least added, the confirm and remove player buttons will be displayed. Confirm button accepts the saved players and returns to the main menu. Remove player button deletes the player that is currently been shown in the screen and displays the previous player drawing.

Regarding the create scenery scene, it uses the same procedure as described above. The sheet that corresponds to this scene is the background one. The add layer, remove layer and confirm buttons have the same purpose as their homologous. Nevertheless, there are some differences that must be taken into account. In this scene, there is not a limit in the number of times that the add layer button can be pushed. This allows to create as many layers as the users want. Conversely, the add player button was limited by the maximum number of players. Another difference is the way that the new layers are displayed. Every time a new layer is added, it is displayed at the top of the others that already are shown in the screen. All the layers can be seen at the same time. When deleting a layer, just the one at the top is removed. The confirm button has the same functionality as in the other case.

The last button is the play game. It will enter the main playable scene. This consists of a two dimensional horizontal platformer. It can be played by up to four players at the same time. The player and background drawings previously saved will be used in this scene to create a environment designed by the players. Figure 3 resumes the flow between scenes.

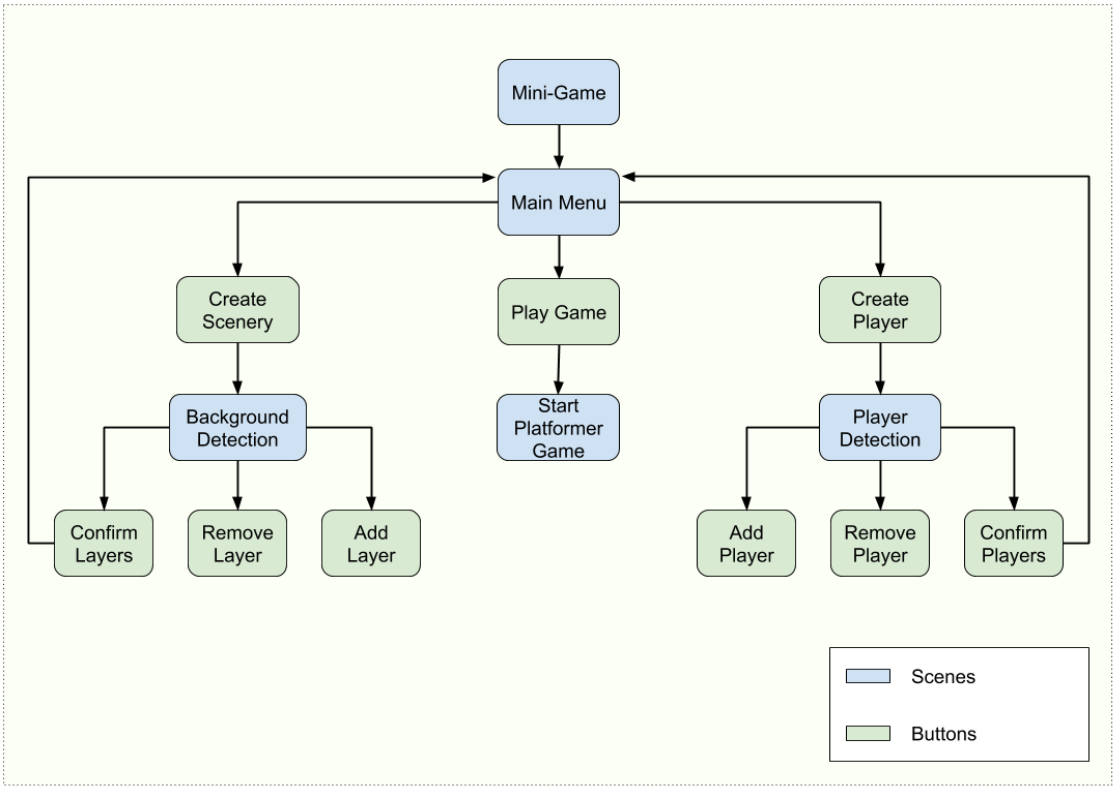

Figure 3: general game flow

2.1.1 Platformer Description

This scene is the most similar to a classical video game. The main objective that the players have to achieve in this platformer is to move to a certain position ahead of them. All the players start at the same position. Between the start position and the objective position exists a gap that cannot be jumped. To fill this gap is also the task of the users. In order to achieve this, they can place platforms in this allowed space. It is divided into several boxes forming a grid. The moment in which they can place them is at the start of each round. Figure 4 shows the layout of this elements.

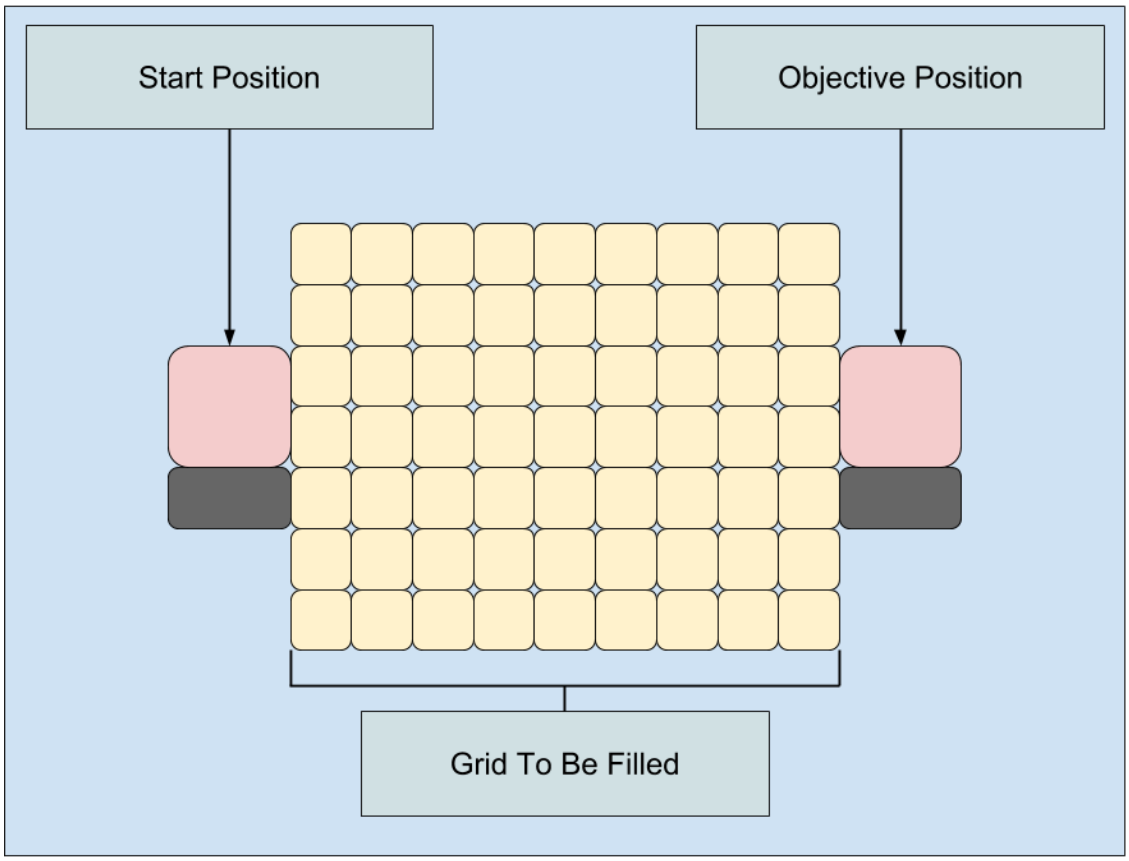

Figure 4: platformer scene layout

First of all, the amount of platforms that the user has to place is shown. It will also display the name and an image of the platform type that will be placed. Placing the platforms is as easy as push the create platform button, draw in the platform sheet the required amount of platforms and put it in front of the camera. If the process correctly detects the position an amount of platforms, it will continue with the next player. This procedure repeats until every player has placed their platforms. Then, the users can start playing the round and try to reach the objective position using the placed platforms.

A round can be considered as each cycle that starts with the placement of the platforms, continues with the users playing the game and finishes with the dead or victory of all the players. When each round ends, users receive points depending on their performance. A game consists in several rounds.

As mentioned above, each round is a cycle that always follows the same flow. When a game starts, players place the platforms in turn. Once the players have placed the platforms they can start playing the round. Each of the players controls a unique character that starts in a specific point when the round begins and has to reach the level finish. They will use the previous platforms to reach the finish and prevent the rest from doing it. Finally, when all players have

reached the exit or died, they will receive different points. Whoever gets the necessary points will win. Figure 5 shows this game flow.

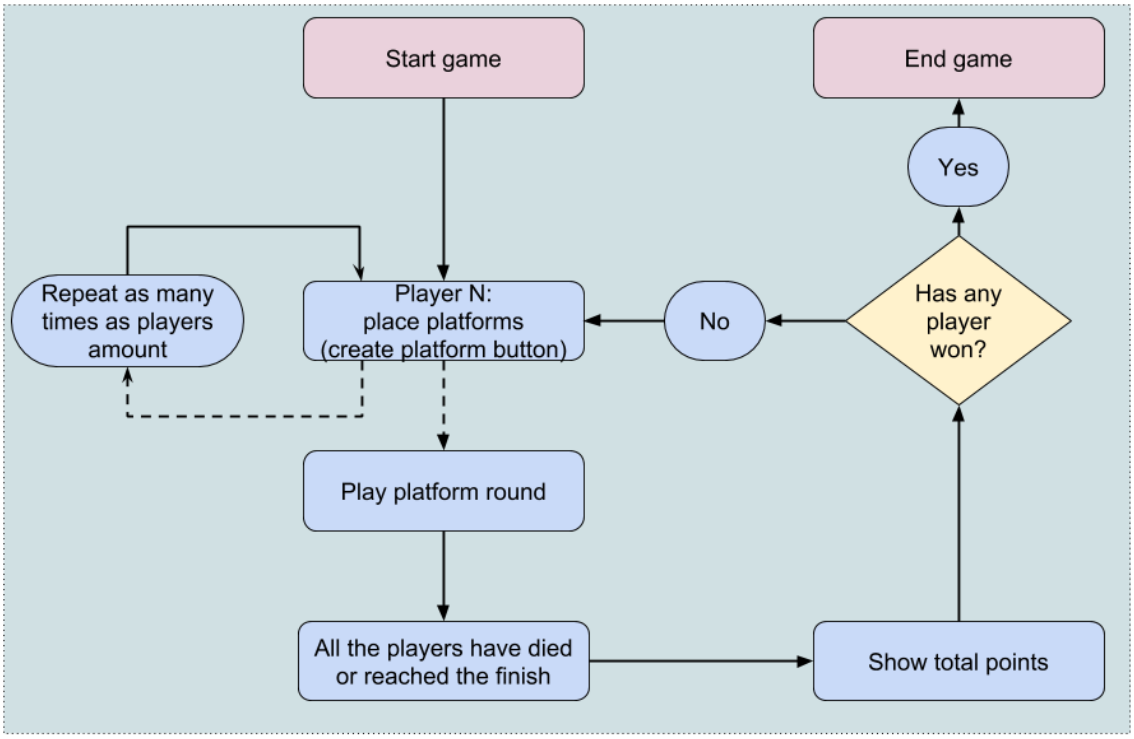

Figure 5: platformer game flow

2.1.2 Player's Interaction

A player interacts with the videogame using an input device (keyboard or controller) or a sheet. With regard to the input device, it is used to control the player's character and interact with the menus. It has the following controls:

Keyboard	Playstation Controller	Function
← or →	◄ or ►	Lateral movement
↓	▼	Crouch
Left Shift	■	Sprint
Space	**x**	Jump
B	●	Give Up

Table 2: game controls

There are two special movements which are similar to others platformers usual movements. A player can hit a wall and then jump in order to reach inaccessible platforms. The player can also sprint and jump so s/he will go further.

On the other hand, the players also interact with the game world by drawing in a sheet their characters when it is requested by the game. Create a character is as easy as fill the rectangles properly and then take a photo with the computer camera. There is no need for virtual character editors. Thus, this interaction with the sheets is more similar to a board game than a videogame.

The world environment can be modified by putting construction elements in a physical grid. There is a correlation between this physical grid and the virtual one, so both of them have the same size (e.g. 9x7). The players have to take a photo when a new construction element has been placed, so this will appear in the game world. Each player can put a new random one when a new round starts. These elements have height and width. The construction elements can be platforms, traps or rewards. A player will die if s/he is damaged by a trap or falls outs. The following Table 3 shows their categorization:

Name	Size	Type
Little wood block	1x1	Platform
Normal wood block	3x1	Platform
Large wood block	6x1	Platform
Spiky block	3x1	Platform/Trap
Sticky wheels	3x1	Platform
Trampoline	2x1	Platform
Cannon	1x1	Trap
Crossbow	3x1	Trap
Ball throwing machine	2x3	Trap
Coin	1x1	Reward

Table 3: platform categorization

2.1.3 Points

It is important to define the rewards that the users will receive when a round finish. The default points needed to win are fifty. The players that reach the exit will earn points. Knowing this, a player can obtain the following points:

Name	Description	Points earned
Goal	Reach the exit	+10
Solo	Only one player reaches the exit	+5
First	Be the first to arrive	+3
Coin	A player has arrived with a coin	+2

Table 4: points

2.2 Main Tools

The programming of this software includes two modules: OpenCV and Unity. The first manages everything that is related with the computer vision library and the second manages what is related with the game visualization and its internal logic. It is needed to find a way to communicate both of the modules because each of them processes different data and has to send the result to the other one. This will also make the development easier. All of these can be observed in Figure 6 and will be explained in this chapter.

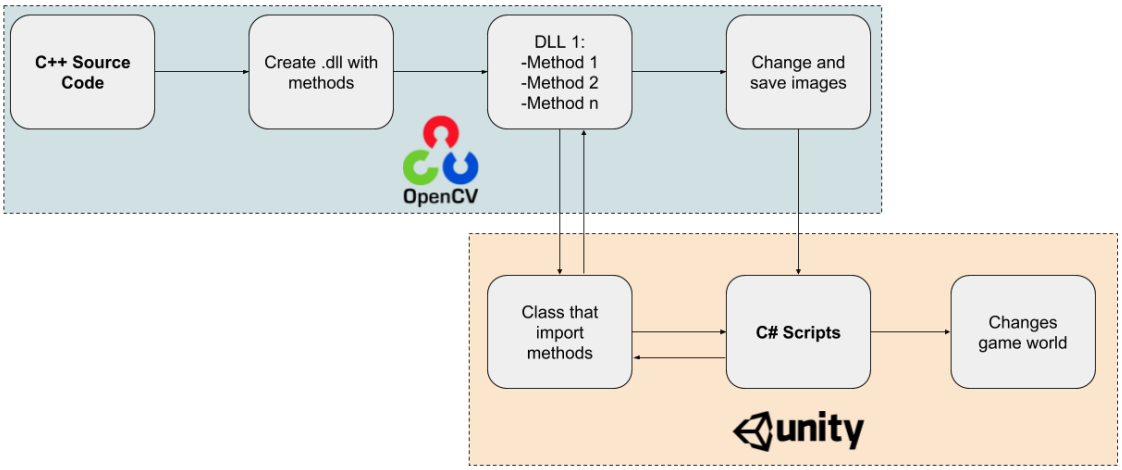

Figure 6: communication between modules

2.2.1 OpenCV

This module is responsible for detecting the real world elements that are necessary. This detection is a process with several phases that starts with the capturing of an image. The final objective is the extraction of the required information through the processing of images. The information must be saved

using a format that is useful for the Unity module. The process is conditioned by the parameters that Unity has established and has sent to this module.

This module is involved in most parts of the video game. It is necessary to use OpenCV whenever it is needed to obtain information outside the computer using the camera. In this video game, this situation occurs in four cases. First, the detection of the user's face to obtain the position that will be used to move an object. This technique takes place in the main menu, both in the mini-game as in the option selection. Second, the character detection that is needed when pushing the add player button in the create player scene. Third, the background detection that is necessary when pushing the add layer button in the create scenery scene. Finally, the platform detection is used in the play game scene. This allows the users to place the platforms at each round.

The managed Unity module must receive data directly from the unmanaged OpenCV module and vice versa. It is really important to understand that the OpenCV module and the Unity one are independent, so everything that is done here needs to be sent somehow. Both modules do not know what is happening in the other side, so it is necessary to take special care of the data that both modules exchange. This is a task that both modules share.

2.2.2 Unity

This module includes the structure that manages the video game and OpenCV functionalities. While the OpenCV module is only responsible for obtaining the real world information, this manages the entire graphic load and represents what the user can see and play with. The Unity module is in charge of using the information and images that the OpenCV module has obtained. Calling an OpenCV functionality and managing the result should not be confused with the process that takes place in OpenCV itself. It is important to notice that Unity does not know anything about what is happening in the OpenCV side. It just calls the methods and reacts depending on the results of the OpenCV procedure.

The graphic interface that the users can see at any moment is also displayed by the Unity module. This interface is formed by every button that controls the game flow between scenes. This buttons can be principally found in the main menu, the create player and the create scenery scenes. The buttons that are used to call the OpenCV functionalities are the responsibility of this module too. The add layer and add player buttons are examples of this type of buttons. Thus, the process will always follow the same procedure when detecting a sheet: push a button, put the appropriate sheet in front of the camera, wait the OpenCV detection and show the result in the screen.

The logic behind the video game is an important part of the Unity module. It manages the change between scenes and the interaction with users. This interaction is exemplified in the main menu mini-game and the platformer game scene. In this scenes, users control their characters by pushing the input devices or moving their heads. Then, this module accordingly responds modifying the needed game elements and screen representations.

3. Game Development

3.1 OpenCV

This part includes everything related to computer vision and the OpenCV library. All the techniques implemented will be explained below. However, these processes share a series of characteristics. These functions allow OpenCV to obtain a series of data that will later be used in Unity. It is important to remember that the OpenCV part does not know what is happening in the Unity part. To call a "C++" function from a controlled environment, OpenCV must expose it using the "C" style of names. A ".dll" file will be generated and it must be stored in the Unity project. This ".dll" will include every method that has been implemented and can be used as if they were typical methods in Unity. The "plugins" folder has been used to save this file and all the dependencies it may need. When exchanging information, pointers and references can be used to avoid the creation of heavy copies of data when dealing with the two modules.

The same procedure should always be followed when using an OpenCV function. Each necessary step is in a different method in this module. However, the execution in the correct order will be the task of the Unity controlled module.

This procedure should always start with the opening of the video camera. This is simply a camera connected to the computer. It must be returned if the process has been a success (the camera is open) or a failure (the camera is still closed). It is important to know if it has been a failure in order to act in the right way in the Unity module. This is one of the general rules when working with these connected but independent modules. It is very necessary to know if what happens in this uncontrolled module is what the controlled module expects or a problem has occurred.

The next step is to obtain the necessary information. This depends on the technique that is running. The Unity module will send the necessary information that this method needs to run. OpenCV captures a frame of the camera and checks if it is not empty. If it is correct, OpenCV will use this frame and any other information that is necessary to perform this phase. Success will be returned to Unity if the result of this phase is correct.

Finally, the camera must be closed when it is no longer required. Unity will not do anything else in this method, so it is not necessary to return anything. All these steps are shown in Figure 7.

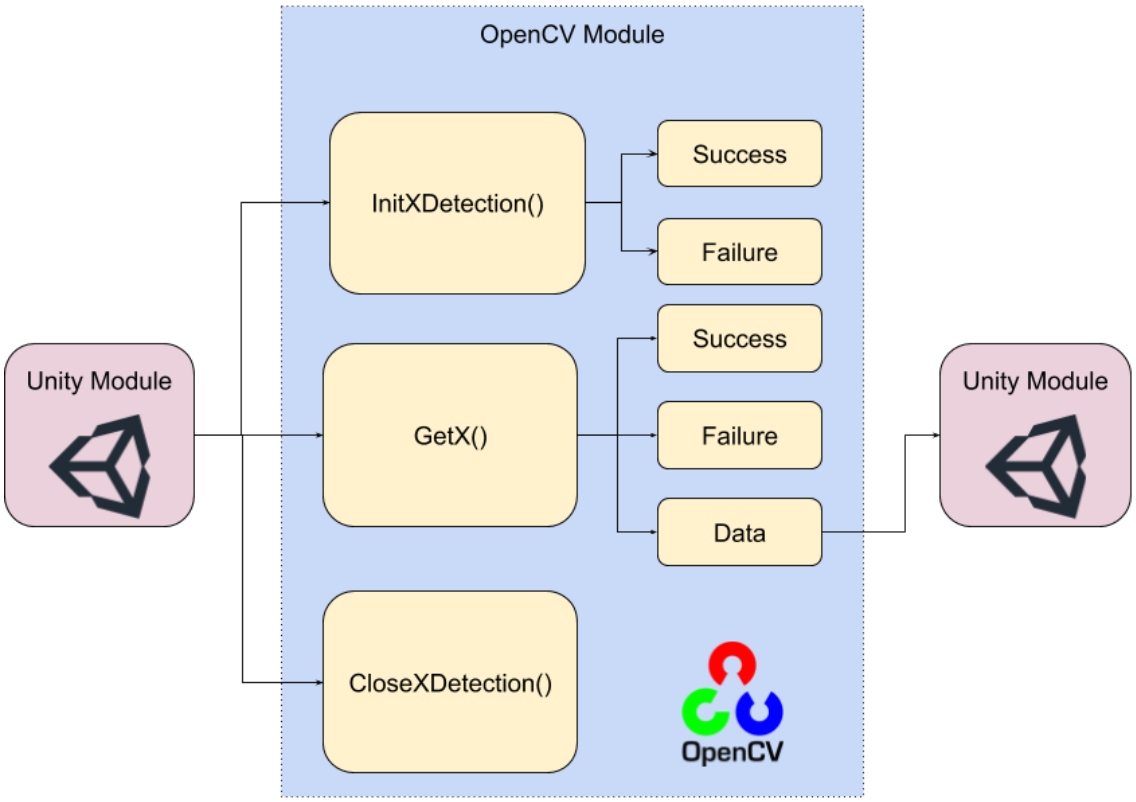

Figure 7: OpenCV general structure

3.1.1 Character and Background Detection

3.1.1.1 Theory and Implementation

These two techniques use a similar method to obtain the necessary data. First, it was developed to detect a character drawn by the players. Subsequently, the algorithm was also generalized to capture the background drawings. The implementation is inside "*Source.cpp*". If it is a character, the function "*GetNewCharacter()*" will be called with the number of the character that is wanted. This number allows OpenCV to store different characters for several players. If it is a background drawn by a player, "GetNewScenery ()" will be called with the layer number. This allows storing different layers that will be used later in Unity. Each of these methods will call "*analysisPhoto()*" with the necessary parameters. These parameters are: the image to analyze, the file where the result images will be stored, their size and the number of rectangles that are expected to be found.

In both cases, the objective is to find a series of rectangles and extract the information they contain correctly. The character's sheet is formed by seven rectangles. The top six rectangles belong to the body parts of the character.

The seventh rectangle is the smallest one and will be used to improve the detection of the drawings. The typical input image can be seen in Figure 8.

"analysisPhoto ()" is the main method that contains all the necessary steps to process the image. First, the input image is converted to greyscale to make the edges easier to detect in the image. The optimal detector known as the Canny Edge Detector is used to find the edges [1]**.** This method obtains an image with marked edges. This image will be used to find the points that form the contours. For each one of the contours the number of points can be reduced using the Douglas-Peucker algorithm and obtain an approximation [2]. This allows the algorithm to discard approximations of curves that do not resemble the objective form. This approximate shape can be categorized into a polygon according to the number of sides. Only the rectangles are necessary for this process. When a rectangle is found, the matrix that it contains is cropped. The order in which these rectangles are found is from bottom to top. It is also important to leave a small gap between the contour of the found rectangle and the edge of the matrix to be stored. This avoids storing the black edge of the rectangle in the cropped matrix.

When all the rectangles have been found, this quantity is compared to the number of rectangles expected to be found. This operation avoids subsequent errors since a character will always have the same number of body parts. These rectangles containing the cropped matrix can be rotated. The next step is to calculate this rotation and find the 2D rotation matrix. This rotation matrix can be used to apply an affine transformation to the matrix of the cropped rectangle. The matrix will have the correct rotation after this calculation. Next, each of the seven matrices is resized. Therefore, the images will have the desired size when they are read from Unity.

When a user fills in a rectangle some spaces will not be drawn. This must be taken into account since it is not tolerable that they appear in the final image. The bottom right rectangle can be used to delete these white spaces. For this reason, this reference rectangle cannot be drawn. The white colour of this rectangle will be used as reference in the other rectangles. First, the image must be converted to greyscale. Pixels with a tone similar to the reference rectangle are stored in a mask. A threshold is used to know if the tone is similar. Finally, the mask is used to iterate through the pixels of the rectangle and modify the alpha. This obtains images without the white background of the sheet, just the drawing. Each one of the images of the rectangles will be saved in the right path.

The different images and filters can be observed in the following Figure 8. The example used is the head of the character but the result is similar for the rest of the elements.

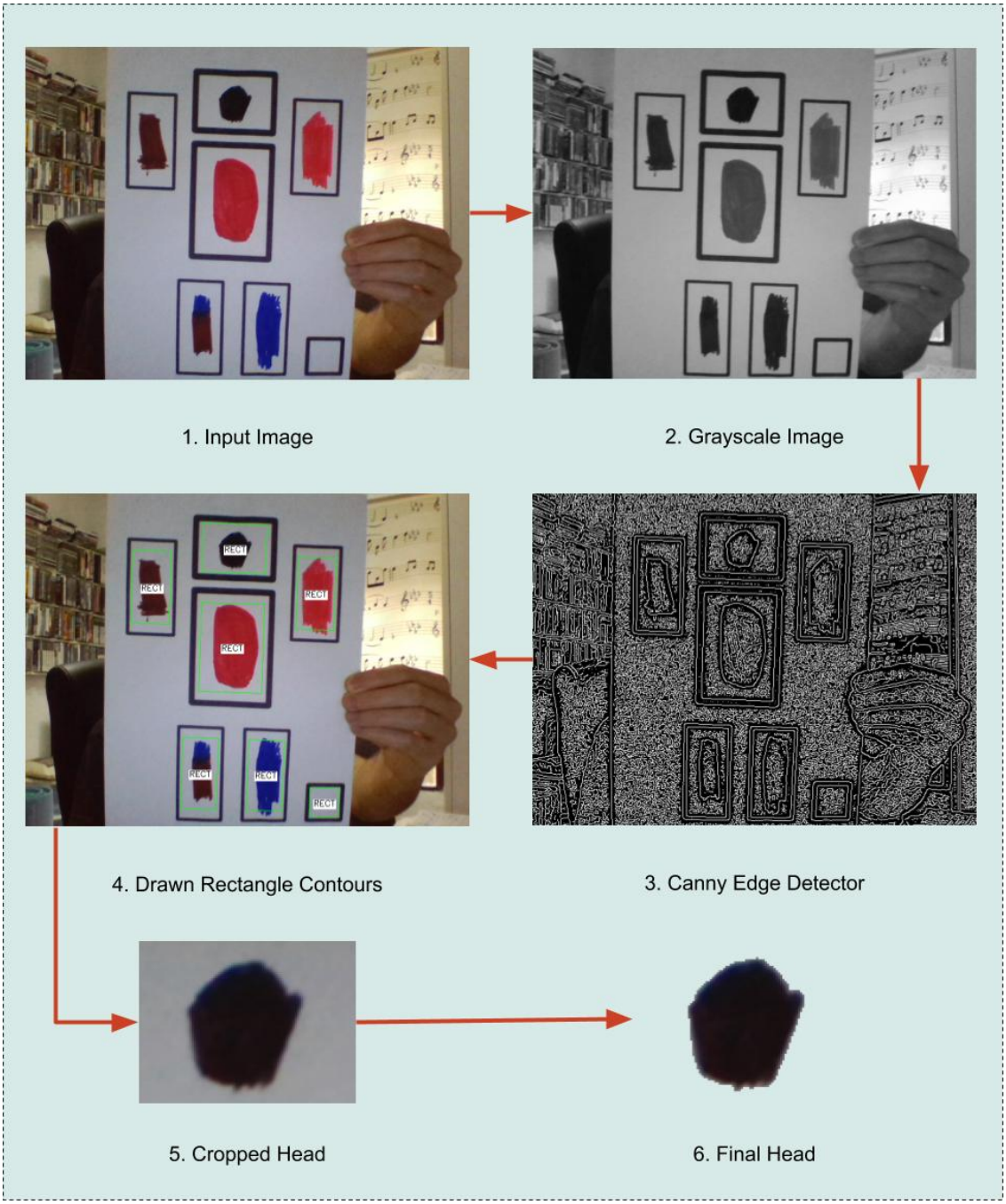

Figure 8: character detection

The process is similar but using a single rectangle in the case of the backgrounds detection. The result will be a single larger image. Each one of the obtained images will be a layer of the complete image. Figure 9 shows the capture of the layers and the result image.

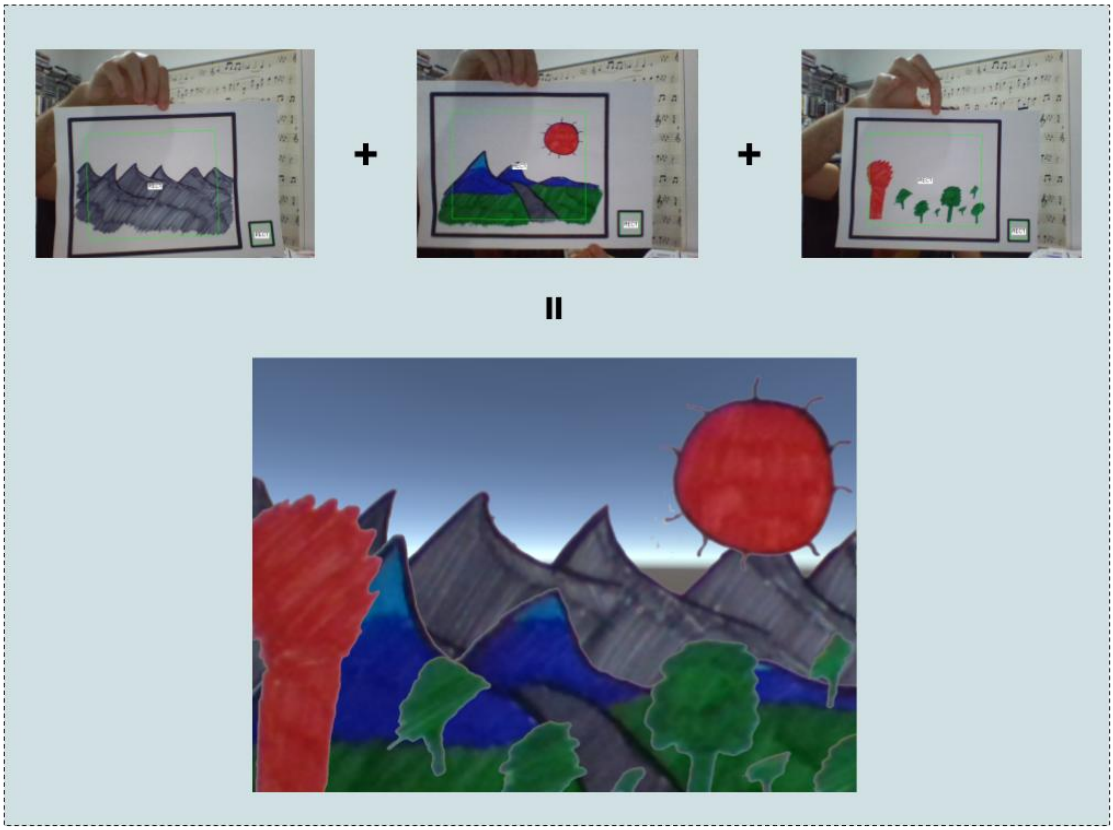

Figure 9: scenery detection

3.1.1.2 Development and Test

The first phases focused on capturing all the desired rectangles. It was also interesting to check the order of detection of the contours used by OpenCV. The left sheet of Figure 10 was used to find this out. Its simplicity avoids perspective distortions and allows to easily know the order in which the images are stored. It can be seen how the first humanoid in the second image of this Figure 10 is different from the sheet used as the input image in Figure 8. This one has closer legs and lower arms. The developed algorithm iterate the image from bottom to top and stores the rectangles according to the lowest centroid. The disposition of the character in the second image can produce mistakes when associating the rectangles and the different parts.

The disposition of the body parts of the character is not trivial and cannot be random. The third image of Figure 10 increases the size of the legs and places the arms higher. This greatly facilitates the distinction of the different parts since the greater distance between the centroids makes them more independent to possible deformations. These are caused by the placement of the sheet by the user and can be both rotations and perspective distortions. The white rectangle for the removal of the white background is introduced in this phase. The last image in Figure 10 shows the greatest distance between the legs. This

improves the detection of the legs and their sorting. This is the last change related to the morphology of the character.

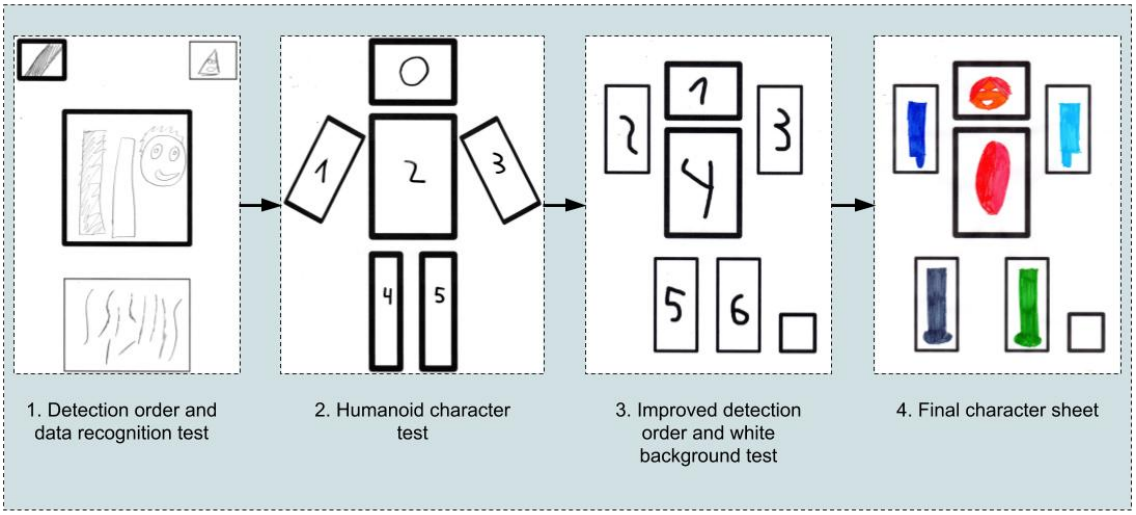

Figure 10: character improvement tests

Another important factor is the colour and paint used. Light colours can be easily confused and eliminated while erasing white background. The used paint must be uniform and not leave white fragments. A paint that reflects light also causes problems. The crayons are an example of not recommended paint. A marker allows to draw intense, uniform and opaque colours so it would be a recommended paint.

There may be problems related to the room lighting when a character or a scenario is captured. These problems are not specific to these two techniques and can happen in any other one. Homogeneous lighting is needed. If there is a much stronger light source than the general one, it can burn a part of the image. This can also cause problems with the removal of white background in this technique. Therefore, intense lights such as direct sunlight or spot lights should be avoided. The process can fail under extreme lighting conditions because it cannot find the necessary rectangles. However, tests have been made with both natural and artificial light to find a correct threshold when eliminating the white background. The small bottom right rectangle that is used as a reference allows to know the colour that the sheet has due to the light colour. This allows to know which colour should be deleted from the rest of the rectangles.

The improvement of the background detection did not need previous tests since it was done after the improvement of the characters technique. The algorithm has been generalized so that it allows to obtain images with alpha following this procedure.

3.1.2 Face Detection

3.1.2.1 Theory and Implementation

This technique allows OpenCV to find the position of a face and its radius. This information can be used later for any type of movement in Unity. The procedure can be divided into two: capture of the current position of the face and smoothing of these positions. The methods related to this technique of the OpenCV module can be found in "*SourceFace.cpp*". The "*Detect()*" method has the following parameters: a pointer to a struct that stores X and Y coordinates and the radius, the maximum number of faces to find and a reference to the number of found faces.

The struct allows to store the data correctly. There is a similar struct in the "*OpenCVInterop.cs*" class of Unity that must be declared sequential and with the same size of bytes. In this case, 3 integers of 4 bytes each one will occupy 12 bytes. This Unity struct will be passed to the OpenCV method and the changes it makes will be stored in the same memory location.

The OpenCV method "*Detect()*" captures an image from the camera and converts it to greyscale. Then, it is resized to improve performance. The next step is to equalize the histogram of the greyscale image to normalize the brightness and increase the contrast of the image [3]. This last image is used to detect faces using a cascade classifier. This is composed of several simpler classifiers that are applied subsequently to a region of interest until at some stage the candidate is rejected or all the stages are passed. Using the new "C++" interface allows using LBP (Local Binary Pattern) features in addition to the Haar-like features [4]. In this program, the "lbpcascade_frontalface.xml" has been specifically used to detect front faces. However, there are other classifiers to detect other parts such as, for example, the eyes. The classifier used allows a quick detection of the object (a face in this case) so that it can be used in real time [5]. This is really important since this algorithm will be used to move objects in a video game in which extremely low fps cannot be tolerated. Once the faces have been detected, they are stored in the memory position that Unity will read. The first part of the process related to the detection of positions ends here.

In Unity, "*CameraFaceDetection.cs*" is in charge of correctly storing the positions of the detected faces and normalizing them. These positions are then used in "*PositionAtFaceScreenSpace.cs*" to apply the desired movement. The second part related to the smoothing of positions starts here. OpenCV detection produces positions that may vary slightly between frames. In some frames, faces can also be detected in places that are not. These positions cannot be applied directly to an object in Unity because it would produce an undesired

vibration in the object. Therefore, a smoothing algorithm must be applied to avoid this.

The process begins by checking if a face has been found and storing its position in the game world. The second step is to check if the new position is within a threshold. This is calculated using two radii from the previous position of the object. Small radius prevents shaking caused by noise in detection. The large radius avoids moving the object to positions that are falsely detected as faces. This is because a face will normally be detected near its position in the previous frame. This process has been exemplified in Figure 11. However, there may be certain problems with the large radius. A face can move quickly or exit the camera's view and re-enter outside the allowed threshold. This can be solved by eliminating the restriction of the large radius when a supposed face has been found outside this radius in a certain number of subsequent frames. Once the position has been restored, the large radius of the threshold must be activated again. A Unity smooth damp function is used to smooth all movements.

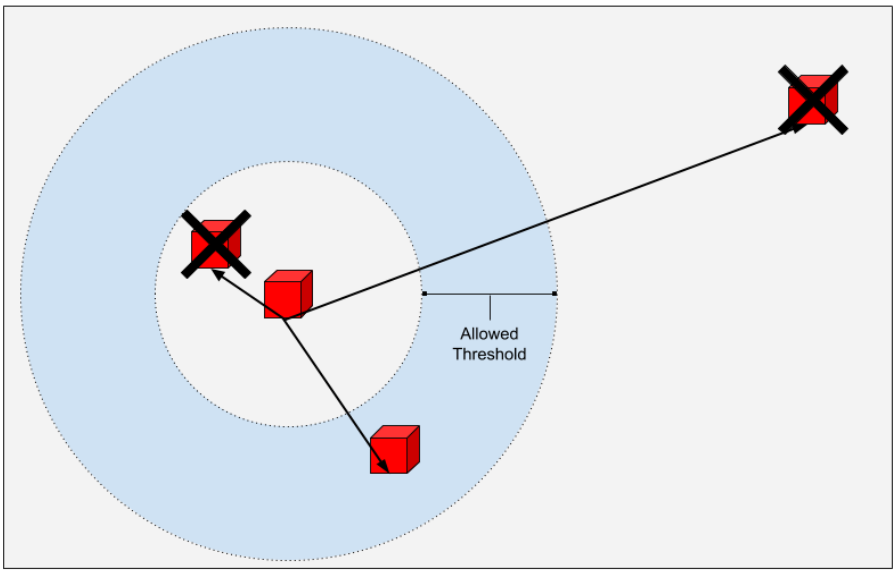

Figure 11: smooth movement

3.1.2.2 Development and Test

In this case, most of the tests were designed to improve the user experience and interaction. This movement is used in a mini-game to move the player's camera and aim. Therefore, it is necessary that the movement is comfortable and intuitive.

The first tests to achieve the smoothing were implemented in OpenCV. However, the necessary code was too complex for the final result. It also produced problems that were difficult to debug. Therefore, it was decided to

divide the process into two parts as discussed above. Smoothing movement in Unity is facilitated by the vector library and methods it provides. This allows the developer to focus on the distance and movement calculations needed in order to obtain a better result. As in other techniques, the light should be homogeneous and the extremes should be avoided.

3.1.3 Platform Detection

3.1.3.1 Theory and Implementation

The objective of this technique is to find the positions of the platforms that a user has drawn. The sheet that the user will use is formed by four corners and a grid. The user can fill the grid boxes in black to place the platforms. An example of this sheet can be found in the input image of Figure 12. Afterwards, this information will be encoded and stored so that the Unity module can know these positions. The technique has been implemented in the "*SourceGridDetection.cpp*" class of the OpenCV project. The "*GetNewGrid()*" method will receive the number of platforms that the user must enter as a parameter from Unity. This method will call "*findGrid()*" with the following parameters: the captured frame of the camera, the maximum width and height of the set of platforms, and the number of desired platforms.

"*findGrid()*" starts by converting the image to greyscale and applying a Gaussian Blur. This blurred image will be used to find four circles in it. These circles represent the four corners of the rectangle that contains the platforms in the video game. The method used is "*HoughCircles()*". It applies the Hough Transform technique to find isolated elements with a certain shape (a circle in this case) in an image [6]. This returns both the position of the centre and the radius. The computation of the radius can be wrong sometimes [7]. However, in this technique it is only necessary to use the centre. The last three parameters of the method are the most important. The first one is the minimum distance between the centres of the circles. Half the number of rows in the image is a good number because it ensures that the sheet is placed close to the camera. The last two parameters must be modified carefully. Smaller numbers allow OpenCV to find smaller circles, but if they are too small they can produce false positives. Next, it is checked if the number of found circles is correct before continuing the process.

The next step is to store the centres found by the method. It is important to remember that circles are not ordered. However, it is necessary to order them to continue with the process. Sorting points means establishing an arbitrary order. It has been decided that the order is the following: top left, top right, bottom right, bottom left. The "*orderPoints()*" method is responsible for it. First, it sorts the points using the X coordinate and divide them into two groups:

leftmost and rightmost. The leftmost ones are sorted using the Y coordinate to obtain the top left point and the bottom left point. The rightmost points are sorted using the Y coordinate to get the top right point and the bottom right point.

Next, a new image will be needed. This must follow the same order of corners as the distorted image. The width of the new image is computed using the sorted points of the distorted image. This will be the maximum distance between bottom-right and bottom-left x-coordinates or the top-right and top-left x-coordinates. It is also necessary to know the height of the image. This will be the maximum distance between the top-right and bottom-right y-coordinates or the top-left and bottom-left y-coordinates. This width and height are used to build a new image that will be the destination of the image formed by the four points. The four corners represent a rectangular sheet deformed by perspective. The transformation matrix of this perspective can be computed and then applied in the new image. The new image will store the image formed by the corners without the deformation of the perspective. It is necessary to maintain the same order of points for this reason. Each one of the corners of the distorted image is associated with the same corner of the image without distorting.

This correctly cropped image without deformation allows to continue with the process. The next step is to convert the image to greyscale and apply a Gaussian Blur. Next, itinerate the pixels of the image and choose one that represents each of the boxes. If the pixel is black, it means that there is a platform. The previous blur avoids detecting false positives since it makes the painting of the box more homogeneous and fills white fragments. This information is stored in a text file called "*CVMap.txt*". It has been coded using an "X" for the platforms or an "O" for the empty places. Once the number of black boxes has been counted, it is compared with the number of platforms expected. If these numbers are not equal, the process is discarded and error is returned. This means that the user has coloured the wrong number of boxes. Unity will read this information later. The different phases of the process can be seen in Figure 12.

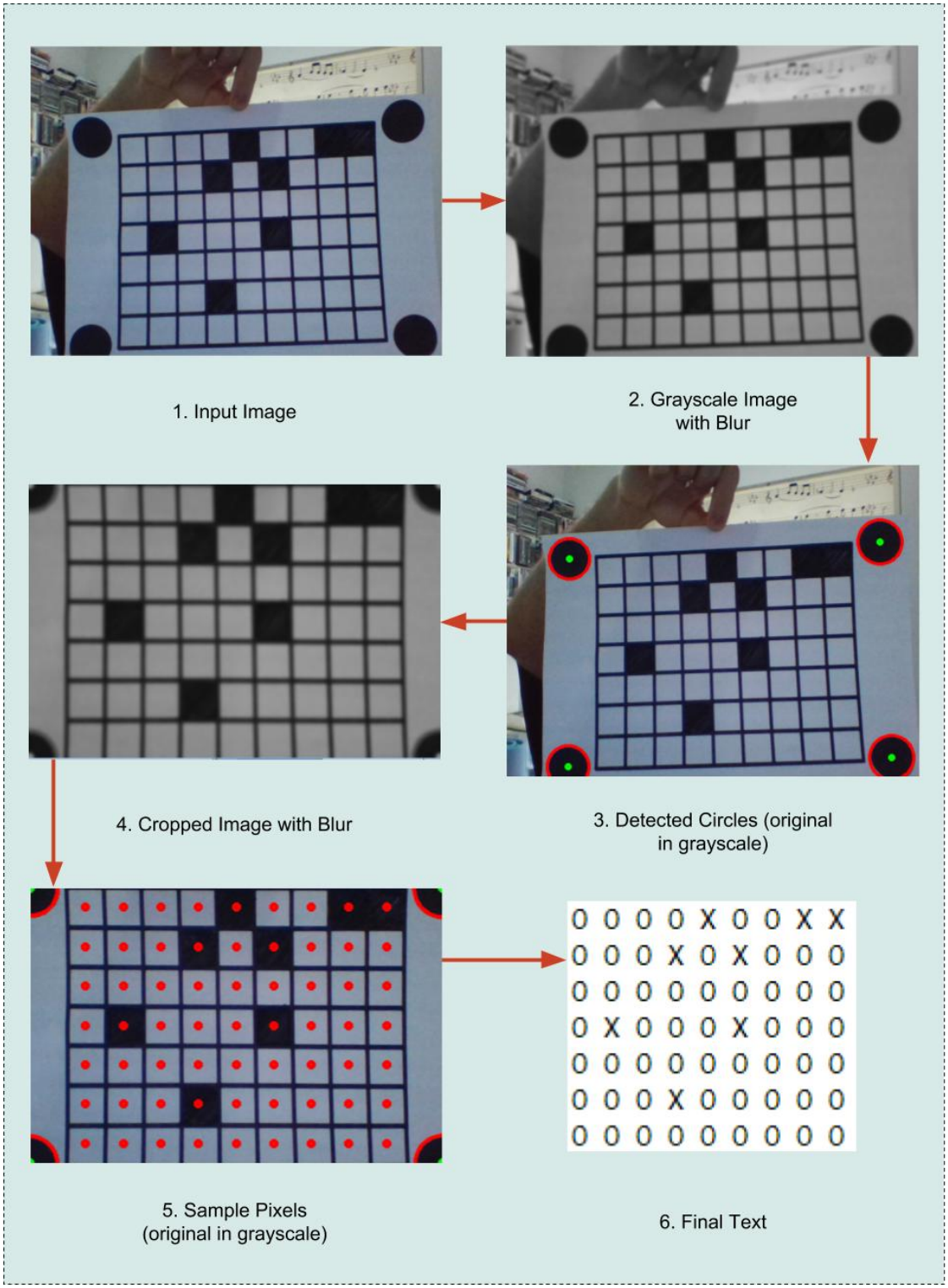

Figure 12: platform detection

3.1.3.2 Development and Test

This is one of the techniques that has changed the most since its initial conception. Initially, grid detection was going to be done without circles. The grid lines would be detected using the OpenCV method "*HoughLines()*".This method obtains a large number of lines that must then be classified and merged to understand the image information. Next, the intersections between them would be computed to know the positions of the boxes. Numbers would be used to know if a box contains a platform. These numbers would indicate the type and position of the platform. A knn classifier would be used to know what number is it. This design can be seen in the image on the left of Figure 13. This process could be simplified by taking advantage of the possibility of deciding the format of the sheet. Therefore, the placement of circles and lines allows to execute the process explained above. This facilitates the task of detecting the positions of the platforms. Regarding the numbers, it has been decided that the video game will choose which is the next platform. Therefore, the numbers have been replaced by black boxes.

Another factor that has been proven is whether the corner circles could be filled or not. No kind of difference has been found between the two in the tests. The grid of these two sheets is incomplete since it is irrelevant for these tests. These sheets are observed in the second and third images of Figure 13. The third image contains a large black box. This has been used to check if the orientation and detection of black pixels was correct. Finally, the fourth image is the one used as the final sheet design. The drawn grid is simply a guide for users. The algorithm does not use these boxes. This is because the space between the corners is discretized following certain divisions of height and width. These resulting points coincide with the boxes that guide the user. However, the developer can choose any other way to discretize this box and increase or decrease this number of points. Figure 13 shows the different sheets used to carry out tests in the different phases.

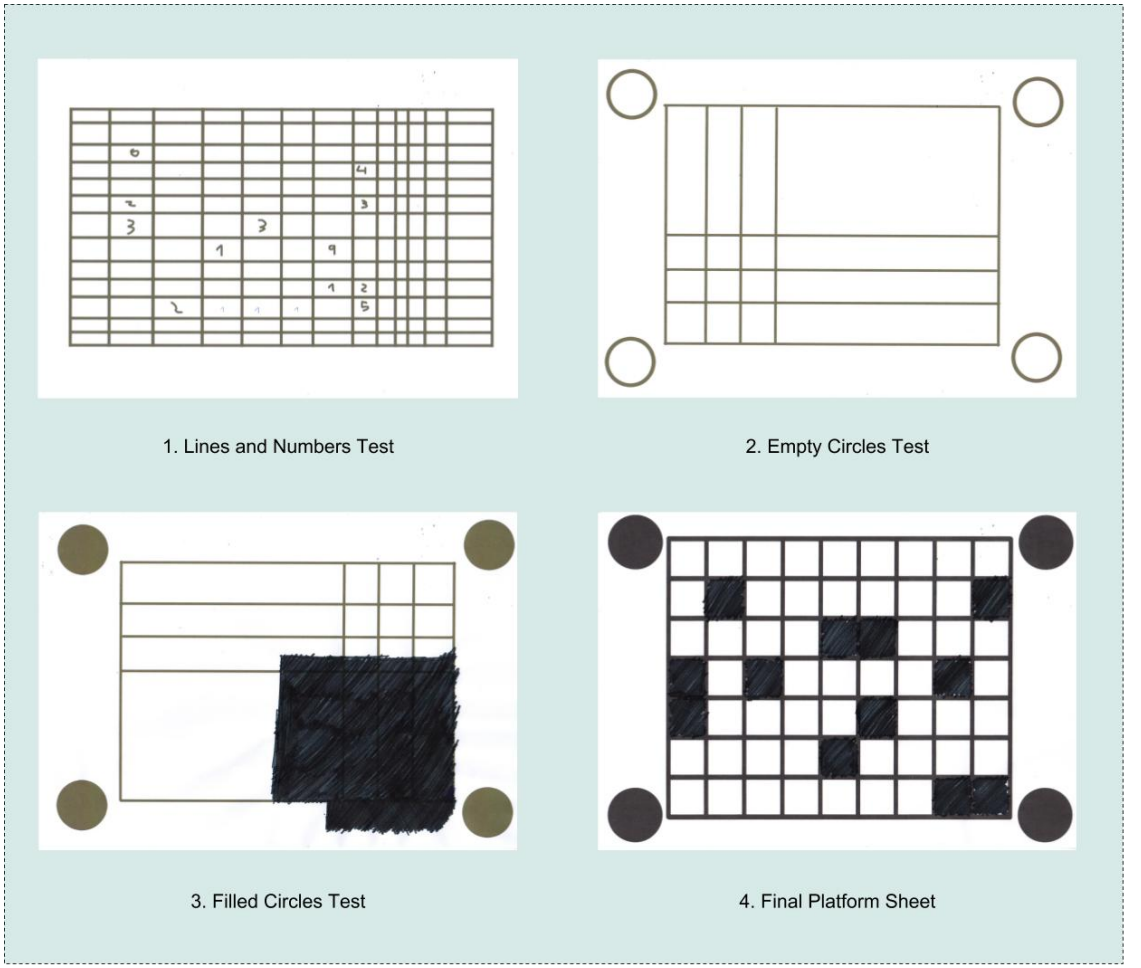

Figure 13: platform improvement test

3.2 Unity Videogame

3.2.1 General Structure

The video game contains four different scenes. Each of them is designed to show one of the four different OpenCV techniques. The main menu contains a mini-game to show the possibilities of facial detection. This is the scene "*TecnicaCara*". When this mini-game is overcome, the main menu is displayed. This allows access to the other scenes. The user must create the players and the background of the game scene before being able to play a game.

As mentioned previously, all techniques must follow the same procedure. Therefore, Unity classes that implement processes with the camera must inherit from the "*ICameraProcess.cs*" interface. This interface ensures that the classes implement the "*OnEnable()*" and "*OnDisable()*" methods. When a class that uses the camera is enabled, it must prepare and open the camera. The camera must be closed when the class is disabled.

The process that a class follows to call an OpenCV function that must obtain some data must always use a coroutine. Unity runs at a certain speed to maintain a tolerable fps rate. However, when Unity calls an OpenCV function it also passes control. This means that Unity will wait for OpenCV to finish what it is calculating and give it back control. If this process lasts a few seconds, Unity will stay frozen that time. To solve this problem the functions with important computational load of OpenCV must be called using a Unity coroutine.

On the other hand, the OpenCV techniques usually need several frames to obtain the necessary information. This capture of subsequent frames cannot be done in a loop in OpenCV since it would cause the freezing of Unity. To avoid this, Unity calls the OpenCV function in each game engine cycle. This is similar to recording a video in which each of the frames coincides with a game engine cycle. The video ends when a frame has allowed to obtain the necessary information or when too much time has passed.

3.2.2 Character and BackGround Detection Scene

The character detection scene is "*MejoraCaptura*" and the background detection scene is "*CapturaEscenario*". Both are very similar. The user has several buttons to add or remove elements. The created elements can be confirmed and saved when the user likes the result. In the case of the characters, the maximum number that can be created is four. This is the maximum number of players that can appear on the platforms scene. By contrast, the user can create as many backgrounds as he wants. These will be

added as overlapping layers. Both scenes allow users to check the result of the detections and decide if they like it.

The implementation uses a class with methods that both scenes share. This class is "*SpriteManager.cs*" and is responsible of creating the sprites from the images obtained from OpenCV. "*SceneryManager.cs*" and "*CharacterManager.cs*" inherit from this class. Each of these classes stores the references to the *SpriteRenderers* that will be used to assign the sprites.

3.2.3 Face Detection Scene

This scene begins with a series of texts to introduce the player. The objective is to teach aiming with the camera and shoot. When the texts end, a series of points appear at the end of the scene. Both coins and missiles will come out of there. These points are generated using the "PointsGenerator.cs" class. This class instantiates points within a width and a height. It also allows to change the resolution of points in these two axes. The curvature can be changed even though it has not been used in this case (the background of the scene is a plane). An object pool has been used to not instantiate and destroy one object each time it is needed. The user must move the camera to one of the objects and shoot. This displacement can be both lateral and vertical. However, due to the detection of the face, a single horizontal line has been created from which the objects will come out. This avoids problems in the detection.

These spawner points will launch both missiles and coins. The missiles grant a point and the coins five points. When the player has achieved the required number of points the mini-game is disable and the main menu appears. To activate any scene the users just have to shoot it as they have done before.

3.2.4 Platform Detection Scene

This scene contains the platform video game. The users have previously captured their characters and backgrounds. Both elements will be displayed together in this scene. This will create a typical platform scene but originally drawn by the players themselves. The characters can be moved by the players and they are animated. The division of the character into different body parts in the detection allows this. Regarding the background, it is animated with a parallax effect ("*Parallax.cs*"). This is possible because different overlapping layers have been captured in the detection.

Up to four characters can play. The objective is to reach the goal and get points. The order of the rounds will always be the same: place the platforms in turns, play the game together and observe the obtained points. Players can place the

platforms or elements they want using platform detection. When it is a player's turn, the type of platform and the amount is displayed. Both parameters are random in each turn. Therefore, players should only worry about the position of the platforms. It is not allowed to place more platforms than the maximum amount.

"*PlatformCreator.cs*" is responsible of reading the file "*CVMap.txt*" that stores the information of the platforms encoded by OpenCV. This class will store two maps: the one created by OpenCV and the map of the platforms that already exist in the scene. The second one will be filled in as users play. If a position that must be occupied is read on the OpenCV map, the platform will be placed on it regardless of what already exists in that position.

The game begins when the players' turns of placing platforms end. There is no time limit. This phase only ends when the players reach the end or die. If a player dies the camera will stop following him. This camera keeps all players on screen. The camera can be displaced, zoomed in or zoomed out. This keeps all the players on screen and take advantage of the space. Players get some points when they reach the end as explained in section 2.2.3.

Different players can interact with their characters using the keyboard and a controller. Three players can use the keyboard. However, the fourth player needs to use the controller because the space on the keyboard is physically limited. Players can jump in addition to the lateral movement. The final controls are the following:

	Lateral Movement	Jump
Player 1	A or D	W
Player 2	J or L	I
Player 3	← or →	↑
Player 4	◀ or ▶	**x**

Table 5: final controls

4. Final Results

4.1 Performance

The performance of this video game is closely related to the OpenCV processes. The Unity video game does not have a large workload so it can run at more than 60 fps. However, the biggest problem can occur when running an OpenCV process. As previously mentioned, Unity invokes a coroutine to record a video from which OpenCV will extract information. The first factor that limits the fps is the framerate of the camera. The framerate of the video game will depend on the framerate of the camera while it is being used to obtain information. In the case of the used camera, this framerate was 30 fps.

The calculations that OpenCV makes during each frame are another important factor. If they last longer than the time between frames of the camera, the fps of the video game will be reduced. In the tests performed the fps were around 30 fps at all times while using the camera. This indicates that the speed was limited by the framerate of the camera but the OpenCV process occurs with a correct performance. All the techniques have shown a similar framerate. However, face detection has sometimes run at 20 fps. This may be caused by excessive calculations to detect faces in a given environment. In any case, this framerate is not a problem to play the mini-game and use the menu.

For the above reasons, it is important to always follow the same procedure: open the camera, perform the computations and close it. This avoids using the camera when it is not necessary and maintaining a high framerate. The result is an acceptable performance at all times and in any technique.

4.2 Facial Detection Results

Before playing the mini-game and displaying the main menu, the video game shows some texts that introduce the player. The appearance of them can be seen in Figure 14.

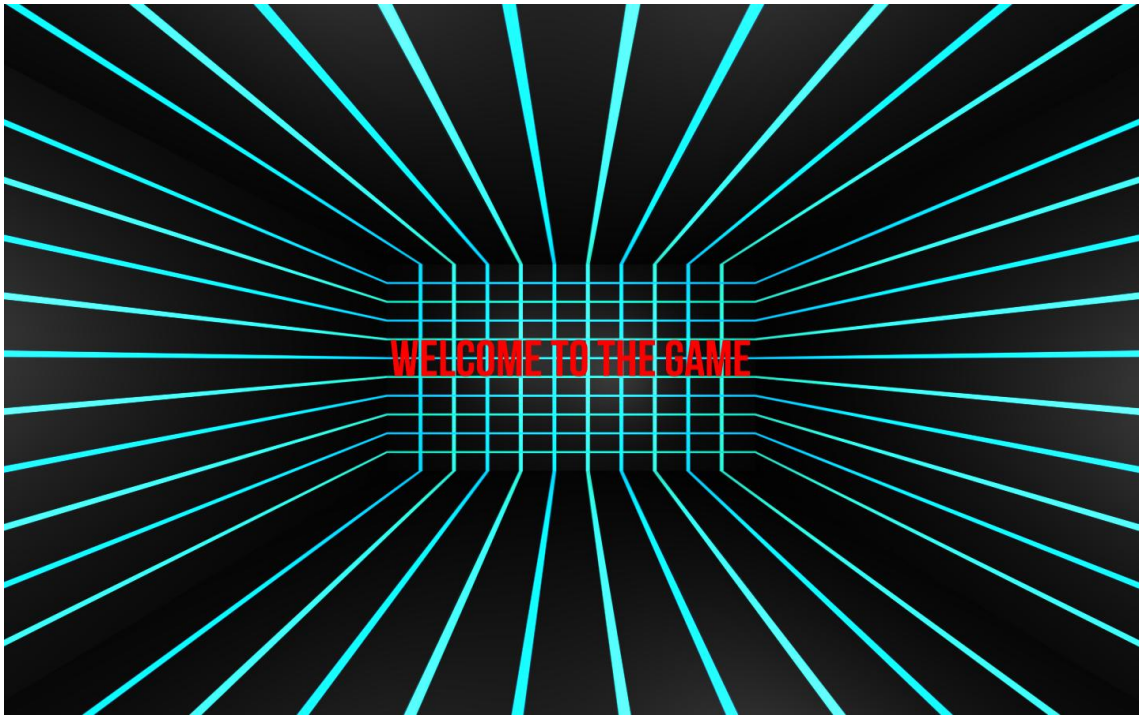

Figure 14: introduction texts

The facial detection technique is especially affected by extreme lighting conditions. Therefore, a soft and homogeneous light is needed. The success rate is acceptable in most cases. However, it is important to remember that "*lbpcascade_frontalface.xml*" has been used. As its name suggests, it allows detecting frontal faces. This means that if the user rotates the face, his position will be lost. That is, the user should always look in the direction of the camera.

In the different tests it has been proven that this rotation occurs when the user is forced to move too much in relation to his centre. This is especially serious in the case of forcing the user to move the face up and down while looking at the camera. If the users are standing, they will be able to bend the knees and move laterally. However, these movements are difficult if the user is sitting. Therefore, in the mini-game scene it has been decided that only a horizontal line of spawner objects appears. Thanks to this, the user must make fewer movements and the detection is more stable. If all these conditions are met, the result of the detection allows to easily control the aiming with the camera. Figure 15 shows the final appearance of the mini-game.

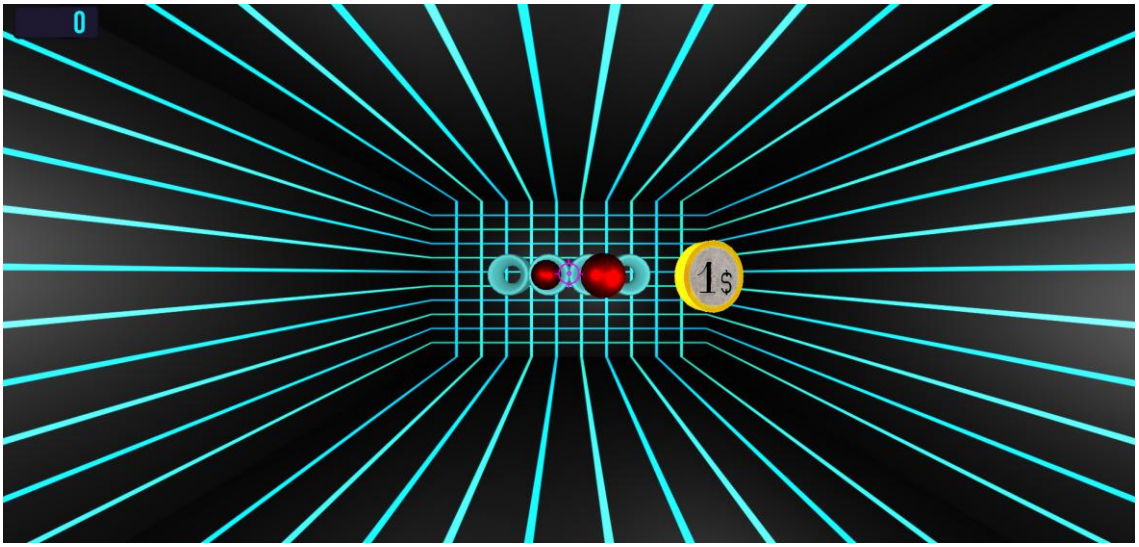

Figure 15: facial detection mini-game

When this mini-game is completed, the main menu is displayed. Figure 16 shows the buttons that guide the user between the different scenes: create scenery, play game and create player.

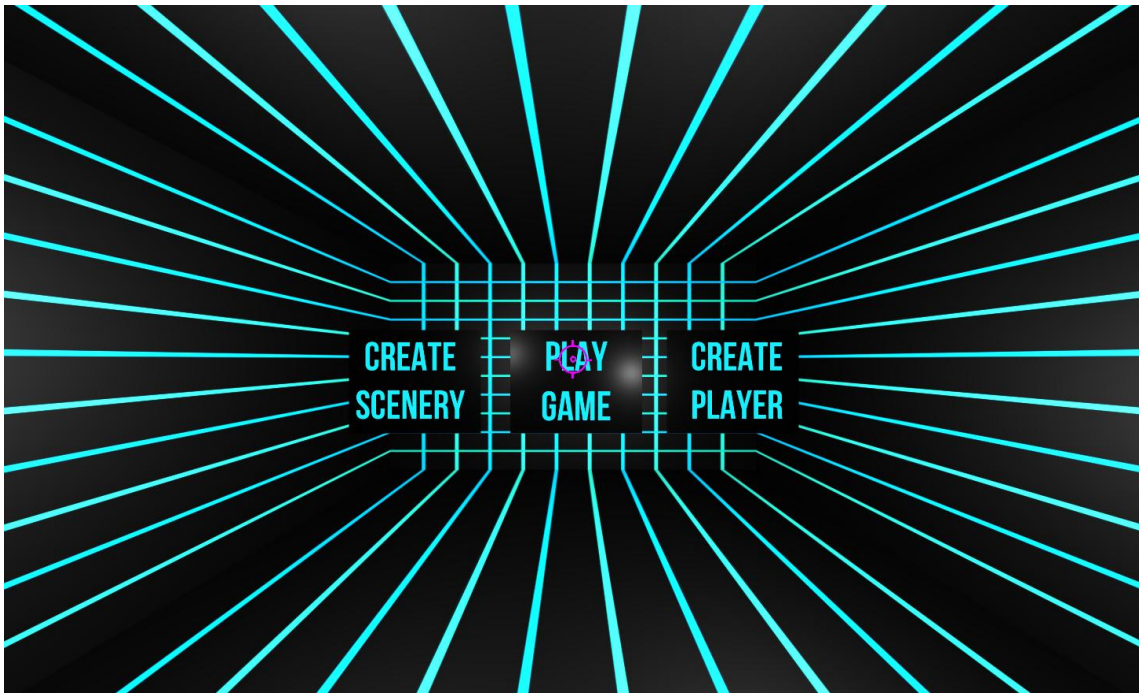

Figure 16: main menu

4.3 Character and Background Results

The characters and backgrounds are detected correctly in most cases. However, some conditions must be met to avoid problems. The lighting cannot be extremely dark or bright. The technique can detect the intermediate range of illumination and apply it to the algorithm. The colours that have been tested that are best detected are the following: black, grey, light blue, dark blue, dark green, red, dark brown.

A generic system for capturing sprites has been implemented. This can detect different parts in a single image to allow more functionalities (e.g. the different parts of a character). The two types of sheets are intuitive and they allow the user to easily draw the different elements. The Unity interface facilitates this task by guiding the user through the different characters and layers of the background. The sheet of the characters has a structure that avoids an incorrect sorting of the parts. Users can immediately see the result of their detections. However, the sheets must be placed with the correct rotation to avoid unexpected results. Users can easily repeat the detection if this happens. Figure 17 shows the final result of capturing both a character and a background. This figure also displays the button layout that both scenes share: add element, remove element and confirm. The user's drawing is located at the center of the screen.

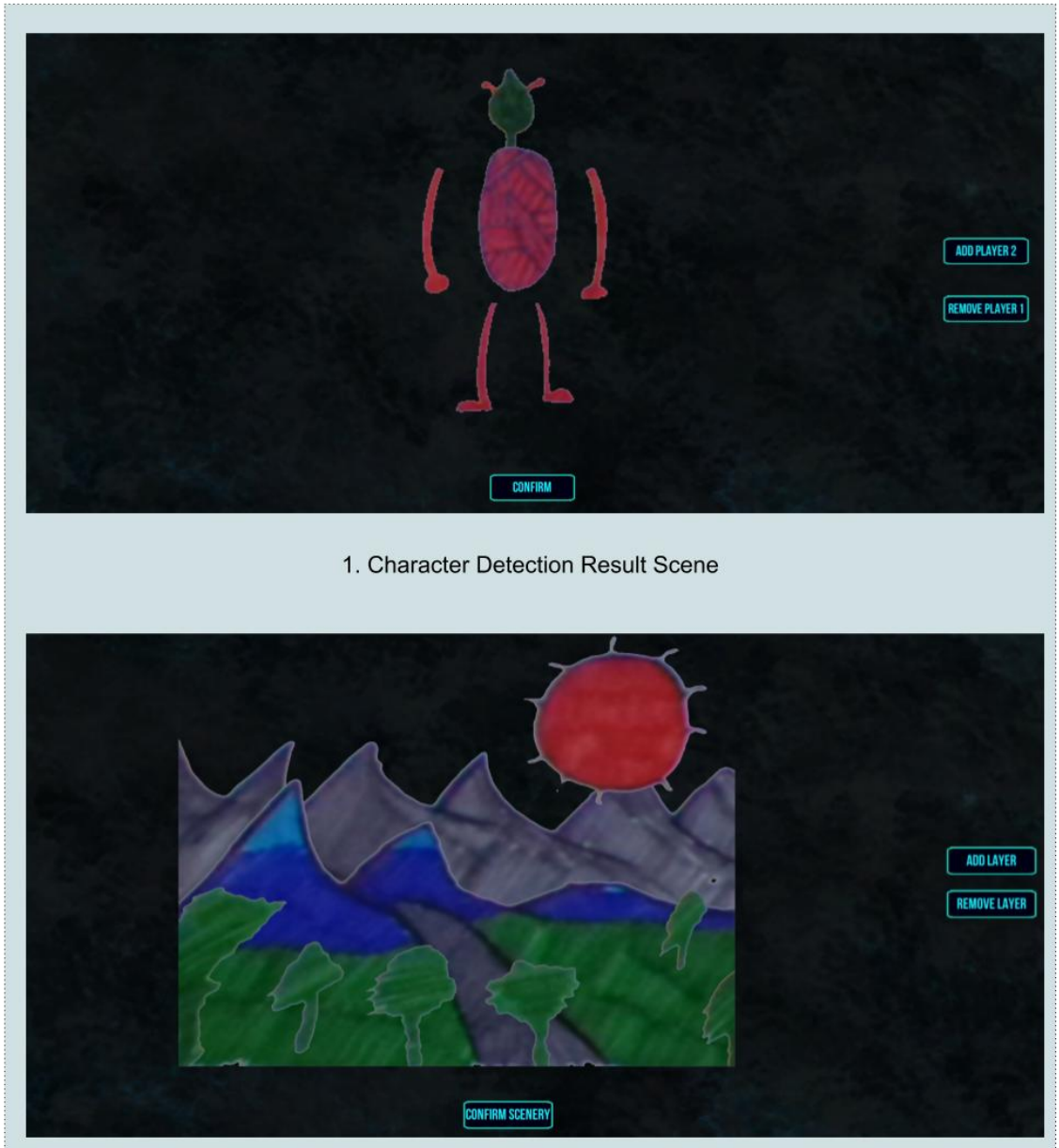

Figure 17: character and background result scenes

The figure 18 shows two more examples of the character detection. Each of them includes the initial design drawn by the user and the result screen that confirms the user that the drawing has been saved.

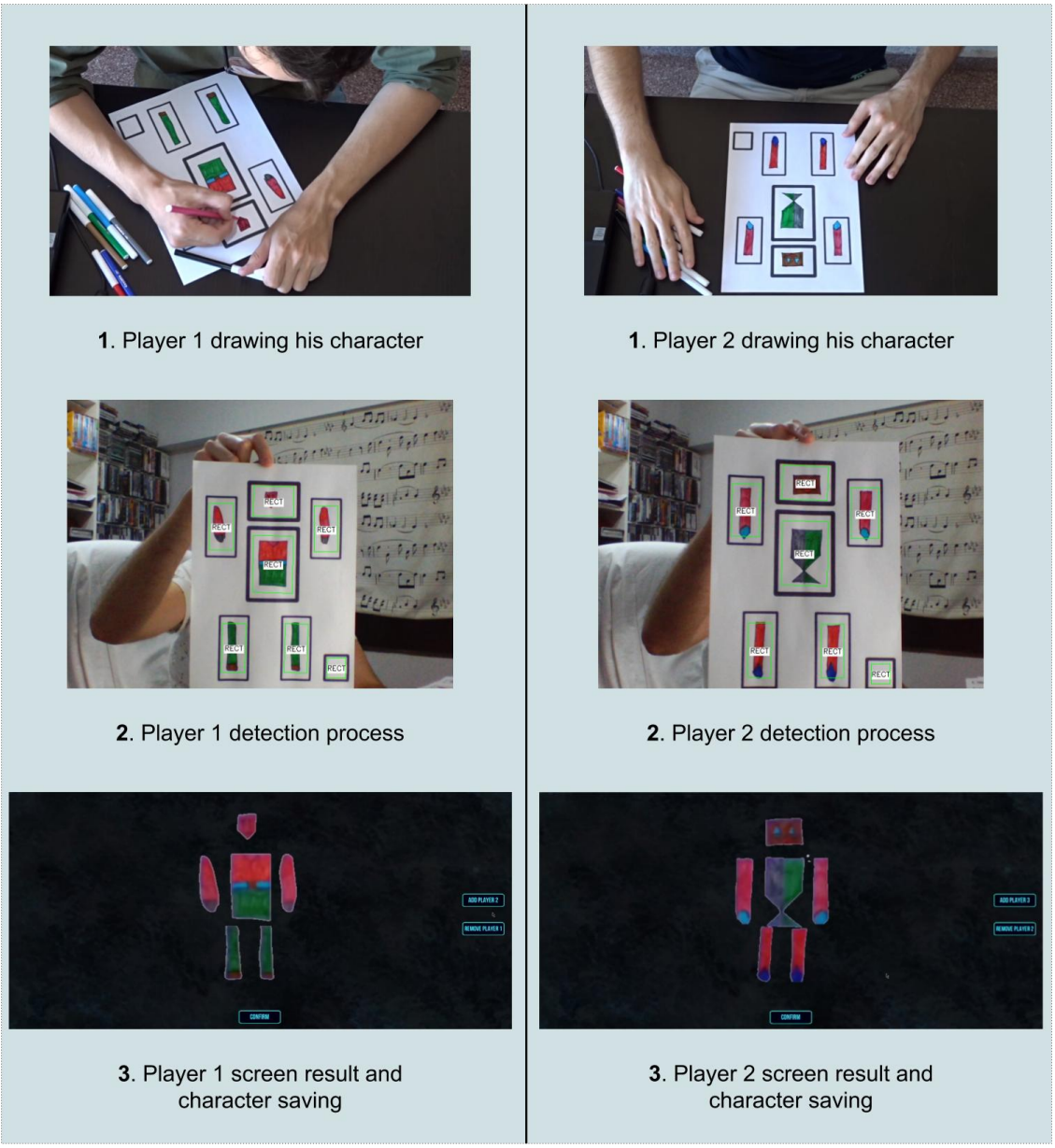

Figure 18: character detection examples

Regarding the background result, it needs more detections than the character process, one for each layer. Figure 19 shows the detection of several layers and the final background result.

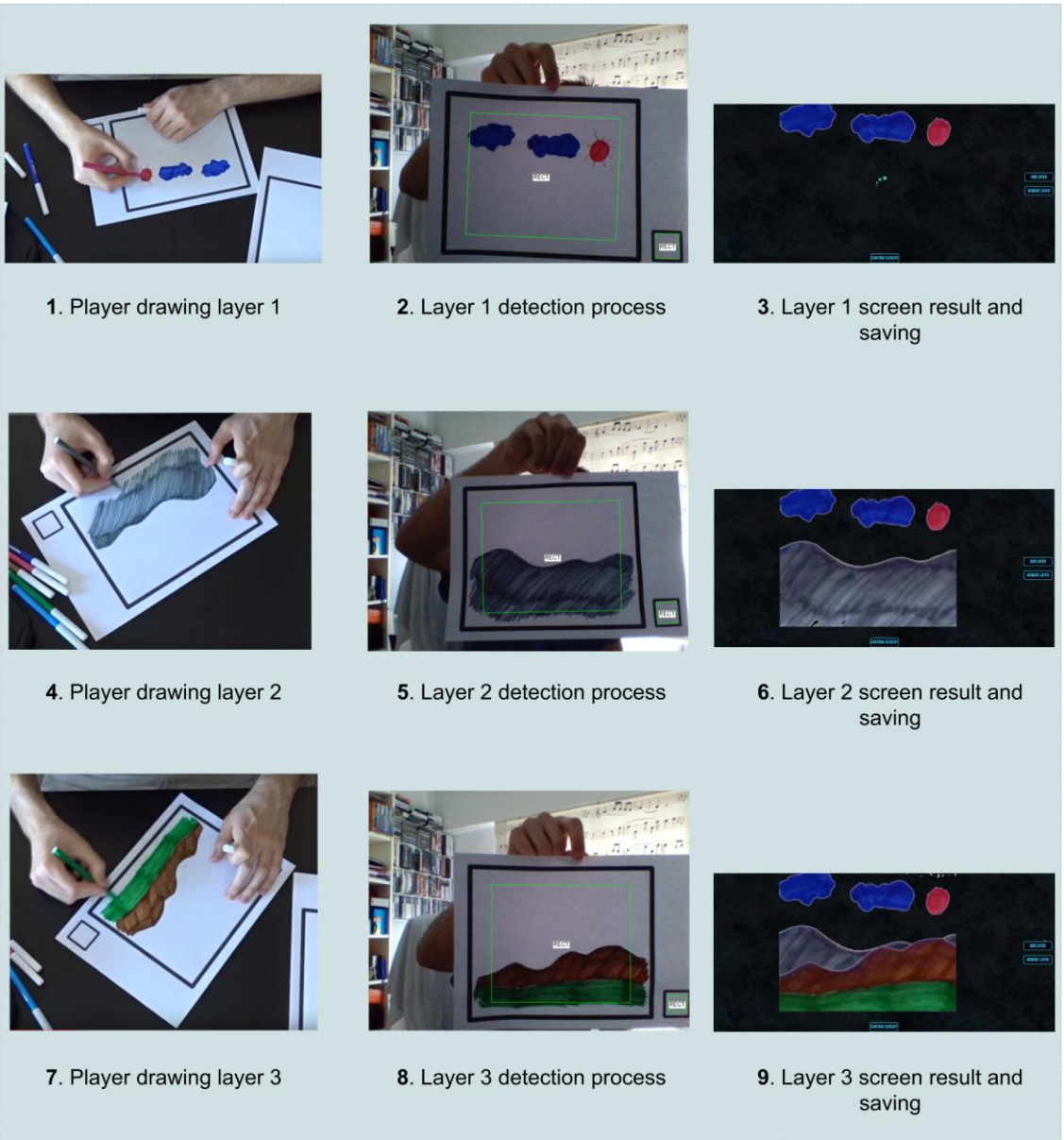

Figure 19: background result example

4.4 Platform Detection Results

This is the technique that has varied the most since the initial approach. The algorithm for obtaining a correct result has also changed on multiple occasions. The elements that form the final sheet facilitate the process of extracting information. The finally developed process gets good results in most cases. This is important because the methods tested previously to solve this problem were much less reliable. Bad reliability causes frustrations in players.

In the techniques of characters and backgrounds an unexpected result is not fatal since the user can delete it and repeat the process. However, this possibility has not been included in the platform scene since it is very difficult for this to happen. In any case, this would just cause an incorrect position of the platforms detected, never more than allowed. During testing, the only possibility of error was caused by the wrong detection of a grid rectangle as false positive (OpenCV though it was a circle). However, the "*HoughCircles()*" parameters have been tuned to avoid this possibility. The result is a reliable process that allows to show another possibility of OpenCV.

Figure 20 shows the initial appearance of the platform game scene. This is the first thing that the users can see when entering the platform game. It displays the grid that the players can use as a reference. This has the same width and height as the grid of the sheet. Figure 23 shows this final platform sheet. The yellow grid tagged as finish is the objective position. The players will start at the left platforms. The left texts indicate the player turn, the number of platforms to be placed, the type of them and an image of the next platforms. The create platform button can be seen below this texts.

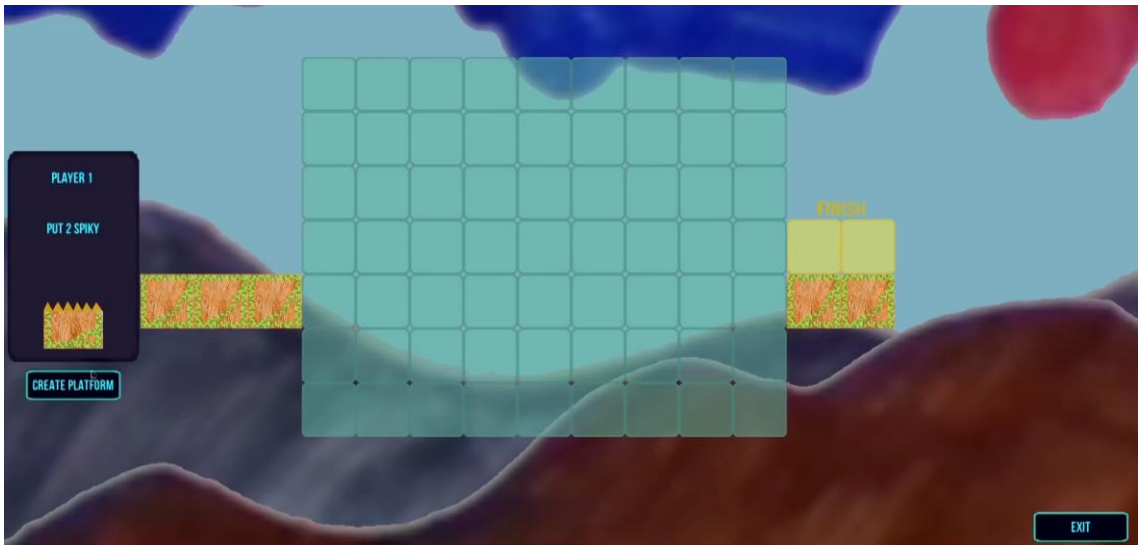

Figure 20: platform game initial appearance

The platform detection phase is shown in Figure 21. Each detection produces the placement of the current platforms. Thus, users can see how the screen is being filled with their platforms in different turns. Figure 21 just shows two of them, but there may be up to four turns, one for each player. The fourth image is the final platform layout that the players will use when playing the game.

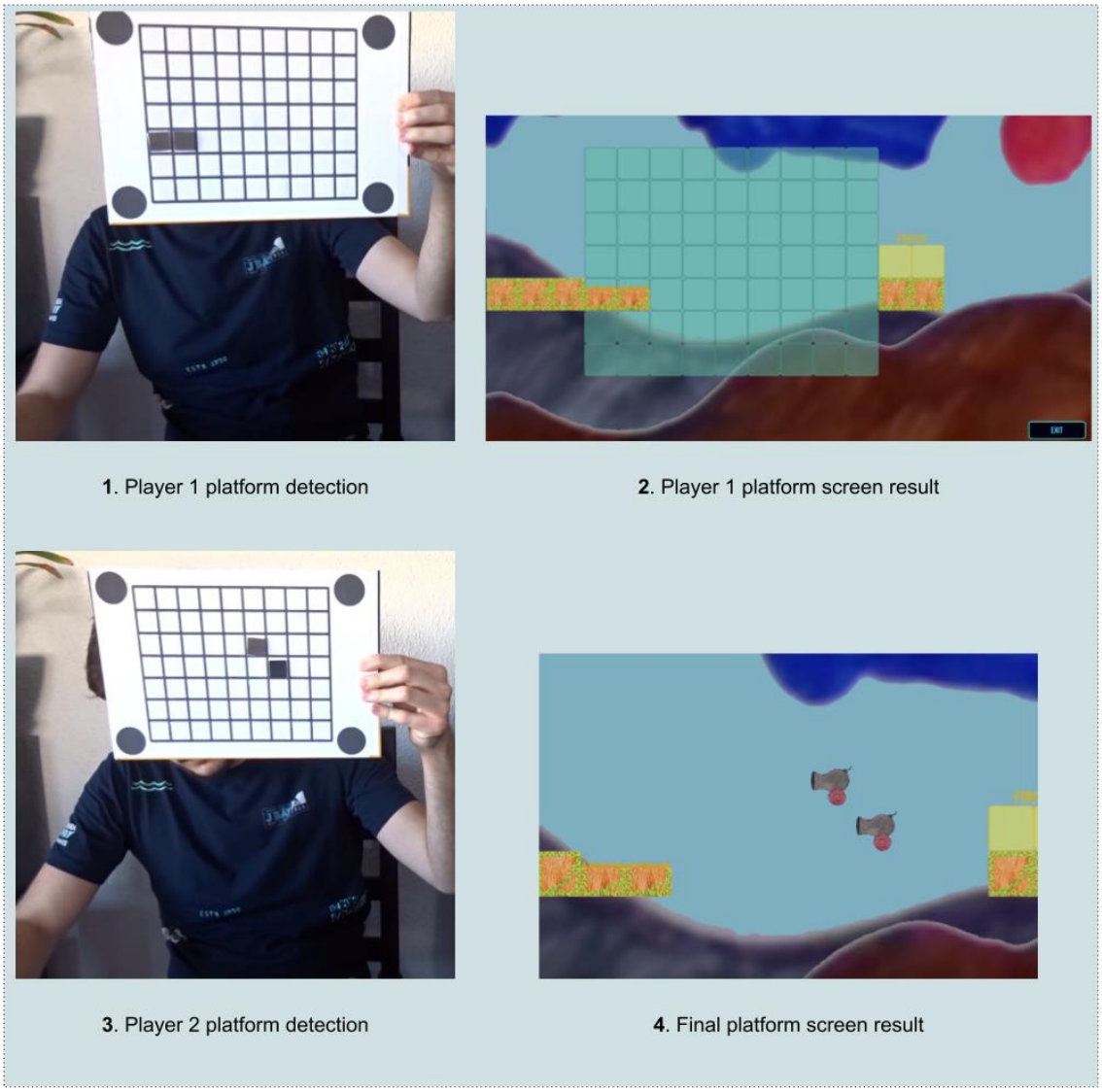

Figure 21: platform detection result

Figure 22 shows two users playing the game. All the elements that the players have created can be seen together in this picture. The background and the characters have been drawn and designed by the players. The two spiky platforms have been also placed by the users in the previous phase.

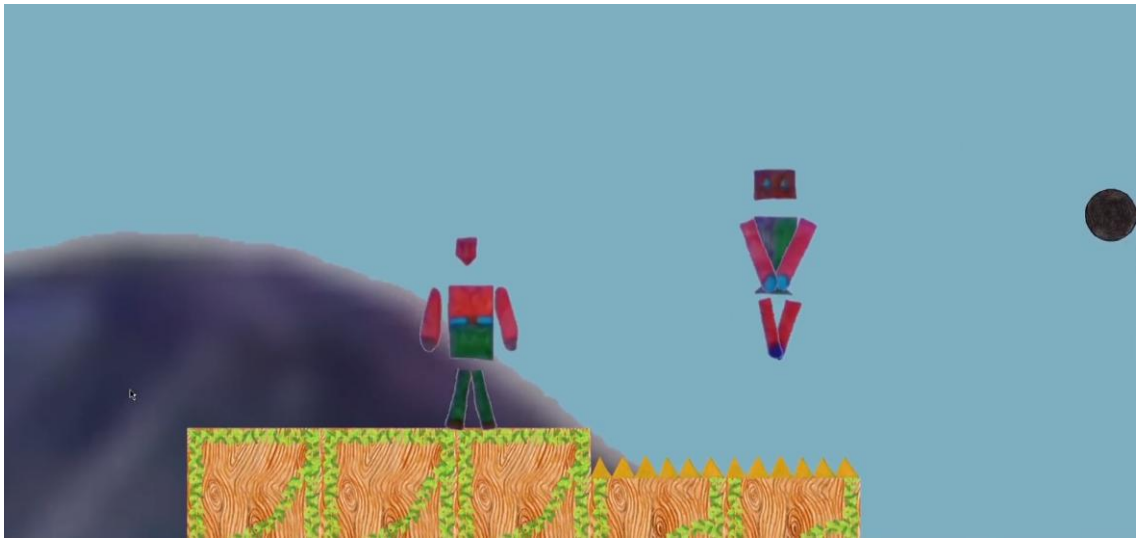

Figure 22: platform scene result

On the other hand, some types of characters movements initially proposed have had to be avoided. These movements need specific animations that are hard to create for generic characters. The final movements of the characters are the lateral displacement and jump. The variety of platforms has also been reduced. All of them can just occupy one box of the grid. The development has focused more on implementing the main objectives of the project and feasible extensions, all of them related to OpenCV.

The final set of sheets can be seen in Figure 23.

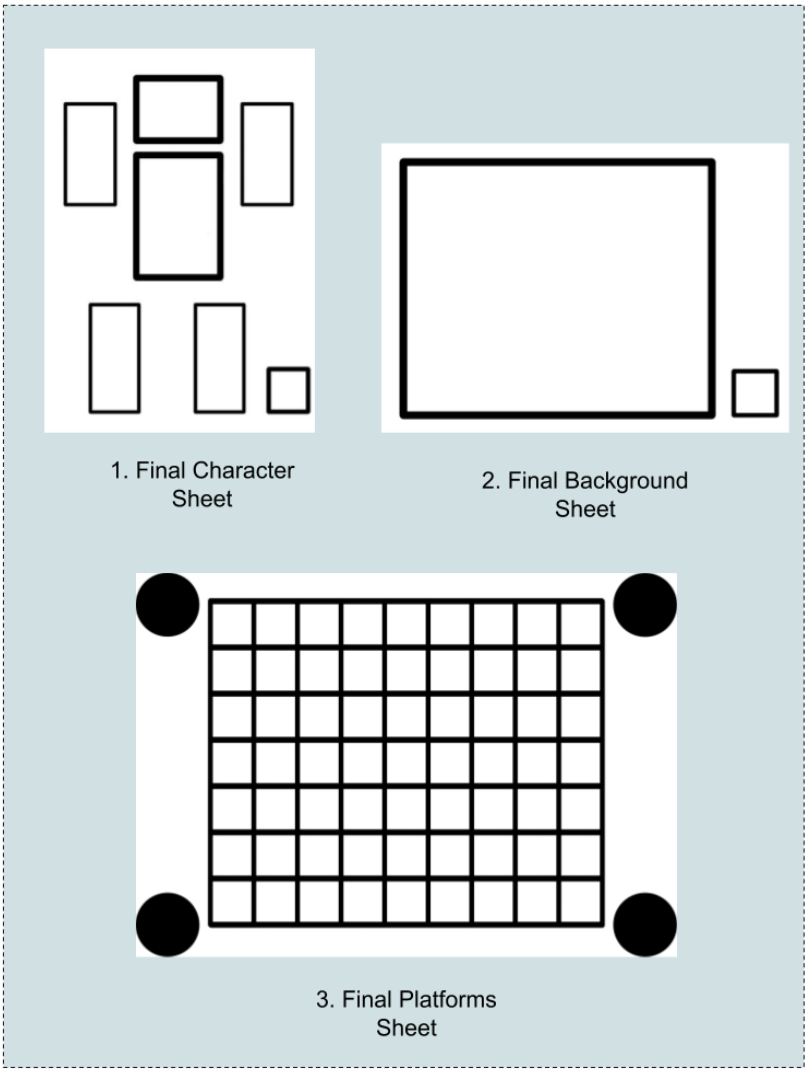

Figure 23: final sheets

4.5 Demonstration

The summary of this results can be seen in the next demo video:

<https://youtu.be/YPlcG-xbRuw>

The executable file can be download in:

<https://gitlab.com/al315318/TFG_Executable.git>

In order to play this game, the user must use the sheets and follow the instructions that can be found in the "instructions" folder.

5. Conclusions

Regarding the main objectives of this project, all of them have been completed. The character and platform detection have allowed to obtain the user drawings. This has required the design of the corresponding sheets. The detection process extracts the drawings and introduces them in the video game. All of this has been included in a video game that shows the functionalities of the developed techniques.

The feasible extensions have been also developed. The created framework has allowed easily joining the Computer Vision techniques and the game engine. This has improved the development speed and could be used in future projects. In addition to the main detection techniques, two more have been added: the background and face detection. These techniques show more possibilities that Computer Vision can offer when applying it to video games.

The shown techniques can have different uses. The fact of showing them on a platformer is just a way to exemplify their use. These techniques can be easily applied in many other game genres. For example, it would be easy to adapt the detection of the platforms to a minesweeper or to a battleship. The detection of the characters and backgrounds allows storing any type of sprite to be used later in a video game. By demonstrating that its use is possible in this example, a world of possibilities and uses of these techniques is opened in other interactive applications.

However, these techniques have some limitations currently. One of the users who tested the detection of the characters drew one with short legs and arms. This causes the character to appear levitating as it does not occupy all the length that would be expected. A possible solution to this would be to create different sheets according to the way the user plans to draw: tall characters, low, thin, thick... This drawing is shown in the left image of Figure 24. Instead, the image on the right was drawn by another user who used lines to fill body parts. This can be clearly seen in the torso. The developed algorithm does not distinguish the place where the white pixels are located. Therefore, it will eliminate all those that have the tonality of the sheet even though they are inside a surface that represents a complete object.

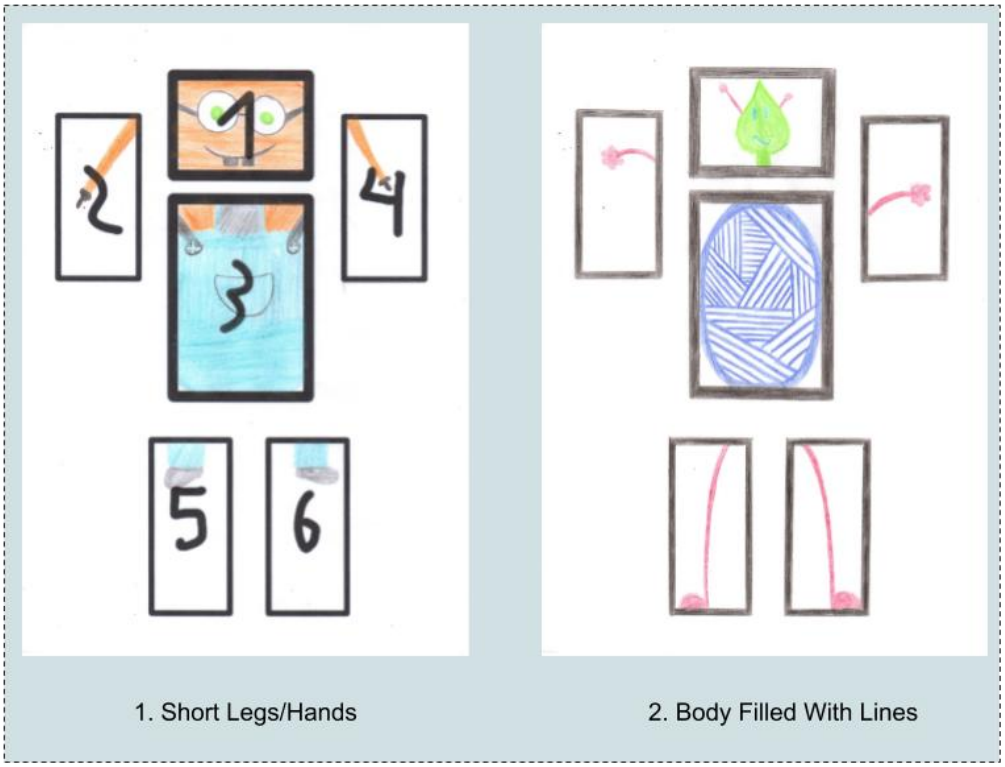

Figure 24: users tests

Another possible improvement is to apply another axis of movement to the detection of the face. Currently, the movement occurs in a plane parallel to the screen. However, the camera could be moved according to the player's distance to the screen. This could be done using the face radius. The smaller the radius, the greater the distance. In the developed mini-game that movement does not make sense, although it can be done in other applications.

Regarding the detection of platforms, users can only decide the position of the new platforms. However, it could be interesting that they choose which platform to place. This could be achieved by using numbers or colours that represent each of the types. Choosing the rotation of the platforms could also be useful. The rotation of all the instantiated objects is the same in the developed videogame.

6. Annexes

[1] OpenCV Project. <https://gitlab.com/al315318/TFG_OpenCV.git>

[2] Unity (2017.1.1f1) Project. <https://gitlab.com/al315318/OpenCV.git>

[3] Instructions and Printable Sheets. <"Instructions" folder>

7. References

[1] OPENCV DEV TEAM. *Canny Edge Detector.* <https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html> [Accessed: 7 June 2018].

[2] WIKIPEDIA. *Ramer-Douglas-Peucker algorithm.* <https://en.wikipedia.org/wiki/Ramer%E2%80%93Douglas%E2%80%93Peucker_algorithm> [Accessed: 7 June 2018].

[3] OPENCV DEV TEAM. *Histograms.* <https://docs.opencv.org/2.4/modules/imgproc/doc/histograms.html?highlight=equalizehist> [Accessed: 8 June 2018].

[4] OPENCV DEV TEAM. *Cascade Classification.* <https://docs.opencv.org/2.4/modules/objdetect/doc/cascade_classification.html> [Accessed: 8 June 2018].

[5] VIOLA, P. & JONES, M. (2001). *Rapid object detection using a boosted cascade of simple features*. In Computer Vision and Pattern Recognition, 2001. <http://wearables.cc.gatech.edu/paper_of_week/viola01rapid.pdf> [Accessed: 8 June 2018].

[6] FISHER, R.; PERKINS, S.; WALKER, A.; WOLFART, E. (2003). *Hough Transform.* < http://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm> [Accessed: 8 June 2018].

[7] OPENCV DEV TEAM. *Feature Detection.* <https://docs.opencv.org/2.4/modules/imgproc/doc/feature_detection.html?highlight=houghcircles#houghcircles> [Accessed: 9 June 2018].