UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Escola Superior d'Enginyeries Industrial,
Aeroespacial i Audiovisual de Terrassa

Augmented Reality: Live Chess

DEGREE THESIS

University degree in Audiovisual Systems

Student: Víctor Vila Bonnin

Director: Jorge Martin Gimenez

Delivery date: 28/09/2020

INDEX:

FIGURE LIST:

TABLE LIST:

ACRONYMS LIST:

INT -> Integer

FLOAT -> Numeric values with floating decimal points

BOOL -> Boolean

AI -> Artificial intelligence

C# -> C Sharp

VR -> Virtual Reality

AR -> Augmented Reality

RM -> Mixed Reality

XR -> Combination of all realities: VR plus AR plus RM

DOF -> Degrees of freedom

SDK -> Software development kit

APK -> Android application package

GPU -> Graphics processing unit

1. INTRODUCTION

1.1. Acknowledgement

First of all, I would like to thank Professor Jorge Martin for helping me choose this project and offering me his help and support throughout the development. I would also like to thank him for his patience in the stressful moments I have experienced throughout the development of the project.

I would also like to thank my family for the help and support throughout the career, without them I would not have reached this point.

Finally, I would like to thank Gerard Busquets, Jose Maria Vila and Jhessenia Adomeit for their unconditional support, for motivating me to keep going in the moments when things were getting difficult and for the patience they have had with me during these last years.

1.2. Abstract

In this research project, a chess video game has been designed, coded and implemented into a virtual reality environment. The design of the pieces has been modelled with Autodesk Maya, while the textures of them all have been done with Substance Painter. The game engine used has been Unity, and the chosen virtual reality test device was Oculus Go.

Virtual Reality is quite a recent, but already well-known and extended concept in the technology sector. Nonetheless, an important legacy is already being forged after it, since it is such a promising medium, all that remains is to seek the limits that are possible to achieve, and how much can we participate in its development.

The aim of this work will be to execute all the phases from the system development life cycle. The results of each phase throughout the development of this project will be shown; as well as the changes from stage to stage. These results will be displayed both in a graphical format, and in code snippets.

1.3. Declaration of honour

I declare that,

the work in this Degree Thesis is completely my own work,

no part of this Degree Thesis is taken from other people's work without giving them credit,

all references have been clearly cited,

I understand that an infringement of this declaration leaves me subject to the foreseen disciplinary actions by The Universitat Politècnica de Catalunya - BarcelonaTECH.

Víctor Vila Bonnin	_____	29/06/2020
Student Name	Signature	Date

Title of the Thesis : Augmented Reality: live chess._____

_____

_____

_____

1.4. Object of the work

The objective of this end-of-degree project is the development of a chess video game in a virtual reality environment, executing all the phases from the system development life cycle: scope definition, requirement analysis, design, implementation, testing and integration, and delivery.

1.5. Scope of the work

The extension of this project covers the following areas:

1) Design and modelling of all the objects we will need in a 3D format. This includes the 6 chess pieces and the game board.
2) Texturing of the figures.
3) Programming of the video game in a 3D environment.
4) Code debugging and testing.
5) Creation of an artificial intelligence for single-player gameplay.
6) Code correction and optimization.
7) Creation of a virtual reality environment and insertion of the video game in this environment.

The scope areas from the project are subdivided in the following milestone:

- State of the art research. Before starting with the project, research must be done to know the technologies currently used for making virtual reality video games. The research includes equipment and software.
- Installation of necessary resources. After the initial investigation, the next step to be complete is the installation of the necessary resources for the development of the project. At the same time, is also evaluated which device is the most suitable for this project and that it will be bought later.
- Modeling of the pieces and the board. Once all the necessary programs for this project have been installed, the next step during the development will consist of creating a 3D model of each of the 6 pieces found in the chessboard (king, queen, rook, knight, bishop and pawn) and the board.
- Texturing of the pieces and the board. Once the pieces and board are designed, next step is adding textures to them.
- Programming the first version. Scope from the first iteration is to develop a finished game that can only be played by one player against the other.
- Testing and correction from the first iteration. After the first iteration, game will be tested to fine tune and avoid bigger bugs during further iterations.
- Programming the second version. The scope from the second iteration ad Artificial intelligence to the game.
- Testing and correction from the second version. After the second iteration, game will be tested to fine tune the Artificial Intelligence.
- Creation of the virtual reality environment. At this point, 3D environment will be move to a virtual reality environment by changing the type of camera and inputs expected by the program.

- Gathering information. Review all the information that I have been recollecting during the project, as well the management of this information for possible future consultation. This activity will also allow me to finalize the writing of the report.
- Drafting of the final report of the Degree Thesis. This activity will start in parallel with the gathering information activity, and will be extended during the entire project until the finalization from the last version.

1.6. Requirements of the work

The requirements to develop this project are the following:

- Knowledge of programming in C#.
- Software dedicated to 3D graphics development.
- Texturing program for 3D models.
- Multiplatform game engine.
- Computer.
- Virtual reality viewer.

1.7. Usefulness of the work

Today, simulation technology is used as a tool in many areas and is an increasingly used resource in business environments. We can see how this technology finds its market potential as a tool for education, as well as a tourist attraction for advertising events or for guided tours of properties. However, their use is not as common among individuals as in companies. With this project I pretend to encourage the creation and use of immersive virtual reality for domestic use with a recreational purpose.

I want to use my project as a gateway to a new environment by creating the feeling of entering an unknown world and getting a totally different experience from playing video games.

Another important point in the project is the creation of an Artificial Intelligence. Artificial Intelligence makes our daily life easier in many areas such as customer service or predictive writing. In this project I hope to give a solution to the problem of playing chess alone, the objective of this Artificial Intelligence is to have the ability to make decisions against a board layout and choosing the optimal move to reach victory.

The final motivation of this final degree work is to apply different fields studied during the career in a single project, seeing how each of the parts affects the final set.

1.8. Project approach

In a professional environment, we find that the development of a video game goes through a team of people and not just one person. Depending on the size of the project we find entire departments dedicated to a single task such as the lighting or the coding.

Because of the impossibility of working in a team for the development of this project, I have chosen to focus on having a playable version of the game in each stage trying to not advance to the next one until I am convinced that there are no errors.

The minimum standards that I have tried to achieve at each phase are based directly on my knowledge on that particular point, my experience and my skill. I have tried to get as close as I could to a professional level to be able to consider the final result as marketable product.

2. DEVELOPMENT

2.1. Review of the state of the art

Virtual reality video game development is based on four mainly things:

1) Computer.
2) VR headset.
3) Game engine.
4) Software development kit or SDK.

2.1.1. Computer

First step was to check if the computer I had was powerful enough to be able to carry out my work without problems or if I had to do some kind of update.

The minimum requirements for the computer depended a little bit on each game engine and each VR headset you were going to use, but to develop this project the minimum system requirements are the following:

- Operating system version: Windows 7 (SP1+) and Windows 10, 64-bit versions only
- CPU: X64 architecture with SSE2 instruction set support
- Graphics API: DX10, DX11, and DX12-capable GPUs
- Additional requirements: Hardware vendor officially supported drivers

2.1.2. VR headset

For the selection of the VR headset is mandatory to take into consideration essentially two elements that will have a direct impact on the development of our videogame:

- Degrees of freedom.
- Autonomy.

2.1.2.1. Degrees of freedom

The degrees of freedom, or DoF, refer to the number of ways a rigid object can move through three-dimensional space. There are six total degrees of freedom that describe every possible movement of an object:

- 3 degrees of freedom for the rotary movement around the x, y, z axes.
- 3 degrees of freedom for transitional movement along these axes.

When we refer to 3-Dof we find that we can only perform the rotational movements of the camera but we do not control the transitional movement; when we talk about 6-DoF we refer to the possibility of moving the camera in rotation and in transition.

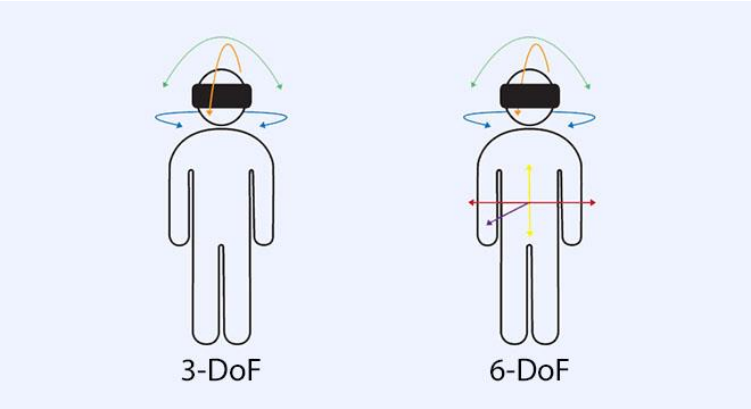

Figure 1. Degrees of freedom (DoF)

Some of the VR displays that allow 3 degrees of freedom are:

- Google Cardboard
- Oculus Go
- Merge VR
- Samsung Gear VR
- Google Daydrem

Some of the VR displays that allow 6 degrees of freedom are:

- Oculus Rift
- Oculus Quest
- HTC Vive
- Windows Mixed Reality

2.1.2.2. Autonomy

Autonomy is another factor that we have to consider. Most devices need a connection to a computer in order to be used, even though there are several models that do not need it.

This connection to the computer also affects the development of a project, because devices that require direct power with a computer can be used as testing devices without the need to install the application on the display. In case you have a standalone viewer you need to install it losing time between tests.

We also find some hybrid models that allow both connections, with the computer and in a self-sufficient way, but most of them are displays that need a mobile phone to work making them not very useful if you don't have a latest generation mobile phone.

On the other hand, displays that depend on a mobile phone are gradually losing market against the self-sufficient devices. Last year, Oculus and Google announced that they were ending their support for devices that required a smartphone to work, giving priority to the development on the rest of their displays.

The displays that need a direct connection to a computer in order to be used are usually high-end devices. These, as we have previously mentioned, are very useful when testing the different tests, saving time between them.

2.1.3. Game engine

The game engine is the platform that we will use to handle all the necessary aspects for the development of our project, from the physical to the graphic section. They are the set of programming routines that allow us to design, create and standardize the performance of a videogame.

We found multiple game engines of free use in the market:

- Unity (free only in the development of non-profit projects)
- Unreal Engine 4
- Godot
- Source2
- CryEngine
- UbiArt Framework
- Urho3D
- ApertusVR
- CopperCube
- Torque3D

And others of payment like for example:

- Unity (for the development of applications with purpose of profit)
- AppGameKit VR
- Skyline

Many of them have an integrated graphic editor within the platform itself, but we find that for the correct development of the project we usually use one different from the integrated one. Some of the graphic editors we find in the market are:

- Autodesk Maya
- Autodesk 3ds Max
- Blender
- 3D Slash
- Meshmixer
- DesignSpark

In this case, some of them are totally free (like for example Blender) and others have a premium version and a free student version (like for example Autodesk Maya).

2.1.4. Software development kit (or SDK)

The Software development kit (or SDK) is a set of software development tools that allows a developer to create a computer application for a specific operating system. The SDKs usually

depend on the virtual reality device to which a project is oriented, although there are some that are multiplatform.

The most known are SteamVR, GearVR and Oculus Integration, but we find some like Open-Source Virtual Reality (OSVR) or VIRO that are opening the market little by little trying to include all the possible virtual reality devices.

2.2. Decision

Knowing the minimum requirements needed for my computer to develop the project, I did not have to upgrade any aspect of the machine, although it conditioned some possible choices if I did not update it. For example, to be able to use the Oculus Rift or Oculus Rift S you need at least an NVIDIA GTX 1050Ti/AMD Radeon RX 470 or greater in the GPU; in my case I had a GeForce GTX 760 removing from my future possibilities these devices.

In this project I will use Unity as a game engine. This choice is based on the flexibility of Unity, as well as the huge community behind it and the large amount of documentation available about virtual reality on its official website. On the other hand, Unity has developed a tool that facilitates the management of the SDKs and allows the connection between them so you don't have to make multiple versions of the same application depending on the device in which it will be used.

I will also use Autodesk Maya as a 3D graphics developer even though Unity has their own integrated platform. This choice is motivated by the familiarity I have with the program, the amount of tools it offers and the easy interconnection it has with Unity as well as with other programs.

On the other hand, for the texturing of the figures I will not use the tool offered by Autodesk Maya. Instead, I will use the Substance Painter program because the final result is more realistic, it allows you to make light tests in the same program without the need to import it to Maya and it allows you to paint directly over the figures as if you had a brush in your mouse. On the other hand, importing and exporting figures from Autodesk Maya to the Substance Painter and back again is almost straightforward with no intermediate steps, making it easy to move the files.

Finally, the visor I've decided to buy is the Oculus Go. These are the low-end virtual reality glasses from Oculus and only allow you to have one control connected at the same time; they also do not offer the possibility to connect the device to the computer for testing so every time you have to do some checking you have to connect the device to the computer, install an APK, disconnect it and test it. On the other hand, it doesn't have minimum requirements in terms of GPU so when using this device I didn't find the need to improve my computer.

I chose this device over others because of the large developer community that exists around this device. On the other hand, even being the simplest glasses, they have everything I need to develop this project even though they don't have the computer connection that would facilitate the testing work. Another point to take into account is the price, being the most affordable among the entire Oculus brand.

2.3. Development

As I said before, the main challenge I faced was that I had to develop this project in a totally individual way when this is usually done by a team of several people.

2.3.1. Modeling the pieces and the board

As I mention in section *2.2. Decision*, the program chosen to model the chess pieces and the chessboard is Autodesk Maya.

The first step to start modelling is to find a reference image to use as a guide during the creation of the 3D model parts. The image finally chosen was the one that can be seen in figure 2. The size chosen for the width of the pieces was 3/4 of a metre in order to be able to adjust the result more easily to the Unity environment.

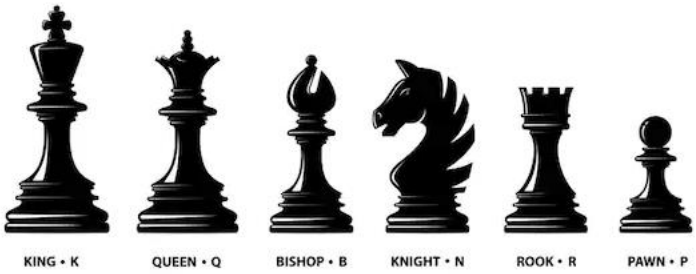

Figure 2. Reference image for the 3D model parts

To model accurately, it is necessary to find two images as we need one for the front and one for the side. In my case, having to model chess pieces, it was not strictly necessary because the pieces are usually identical frontally and laterally except for the knight. The distribution of the panels with the reference images can be found in figure 3.

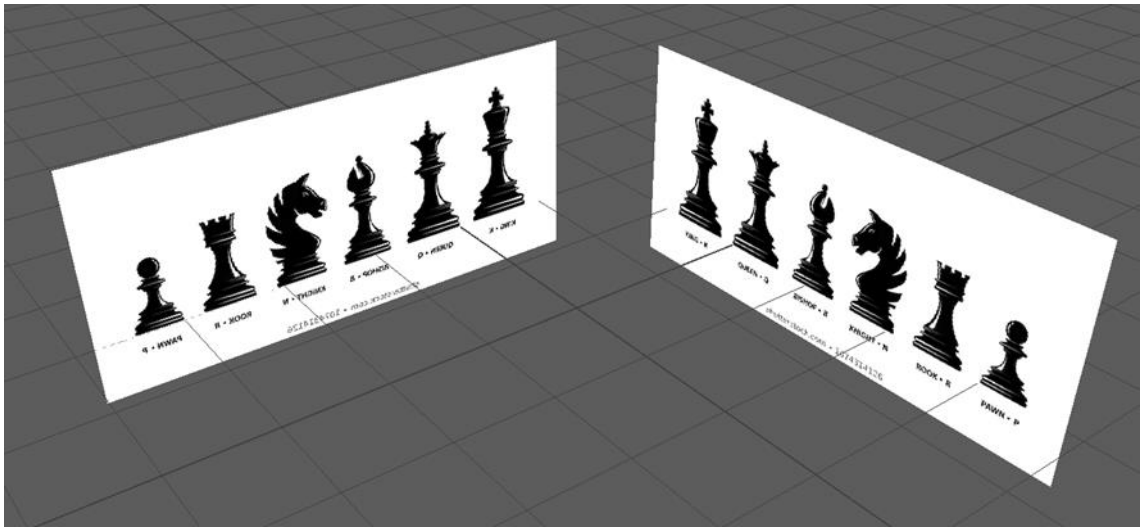

Figure 3. Distribution of the panels with the reference images

Initially I created a base (figure 4) which I later adjusted by scaling it for each figure. This base was used to model the king, queen, bishop, rook and pawn due to the similarity between the pieces. Once the most similar pieces are finished, I started working with the knight.

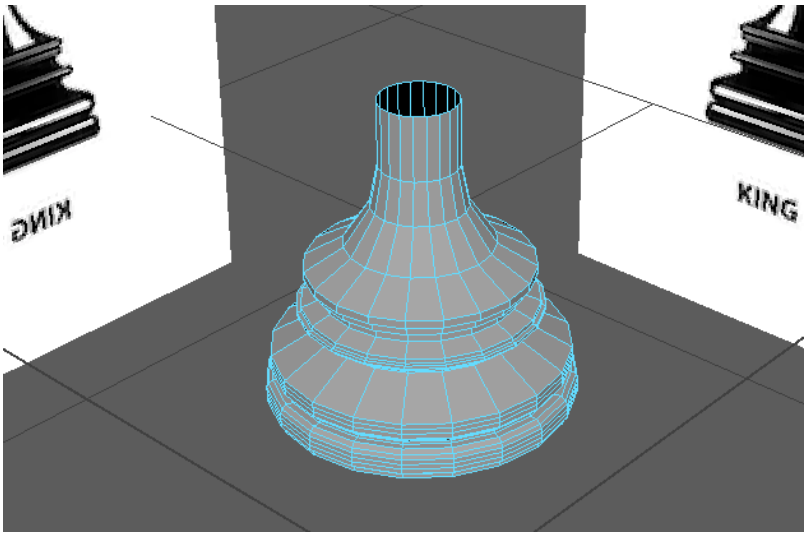

Figure 4. Base used to model the king, queen, bishop, rook and pawn

The objective in the development of the pieces was to try to minimize the number of polygons of each figure maintaining a smooth line in order to overload the game as little as possible later, if the game has less amount of polygons, system will need less processing capacity to load them.

Autodesk Maya offers the *Soften Edge* tool that softens the edges making the object rounder trying not to add too many polygons; this tool is very useful to achieve smoother lines once you have completed the figure. An example is found in figure 5 where you can see that the orange figures have the *Soften Edge* tool applied while the blue ones don't, the result is that the orange objects are much smoother but you don't perceive an increase of polygons in the figures.

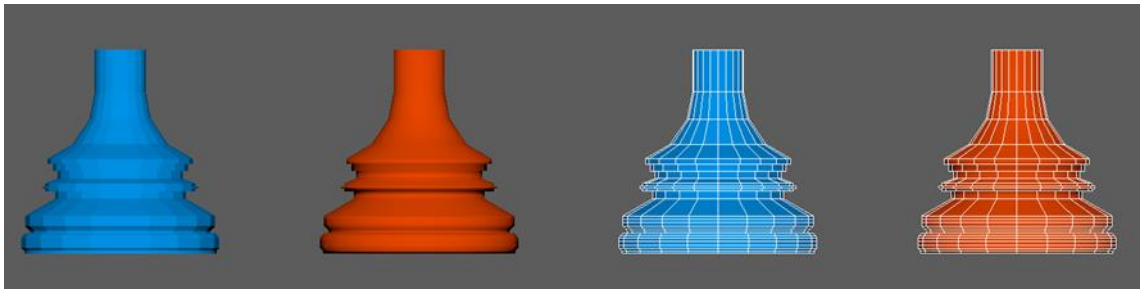

Figure 5. Differences between applying the *Soft Edge* tool (orange objects) and not applying it (blue objects)

To finish the modelling of the pieces, the *object point* was placed in the center of the base of the figure. The *object point* is the reference point that the other programs will take to place the figure, this point will be the one they will refer to when we place a piece in a 3D environment. The final result is the one we find in the figure 6.

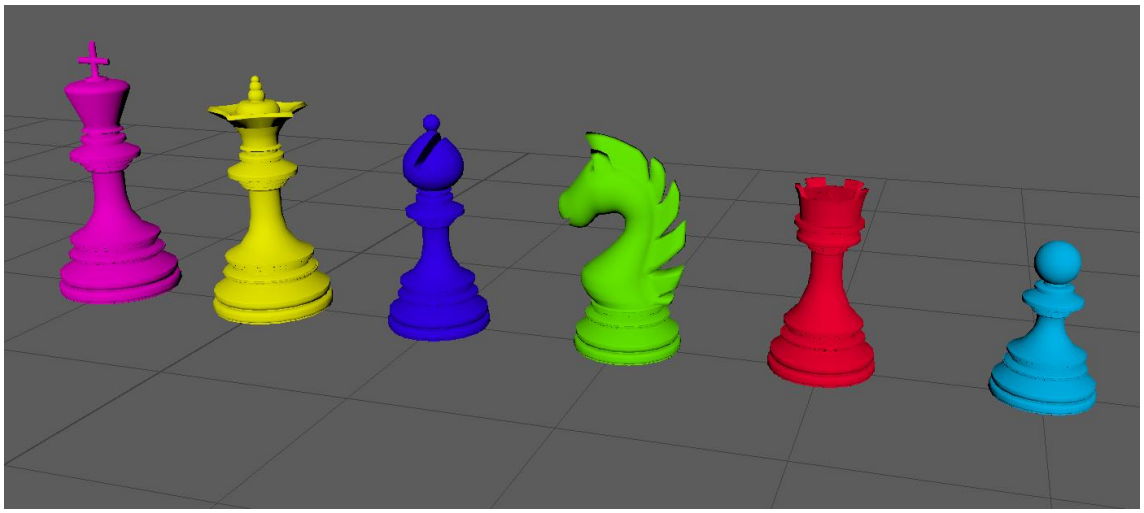

Figure 6. Final model of the pieces

Once the figures were finished, the board was modelled without a reference image due the simplicity. When creating the board, the size of the pieces was taken into consideration, creating cells of 1 meter long by 1 meter wide. The cells are not totally regular; they have a small cut-out in the upper edges that will cause a more visual separation. The final result of the board is shown in the figure 7.

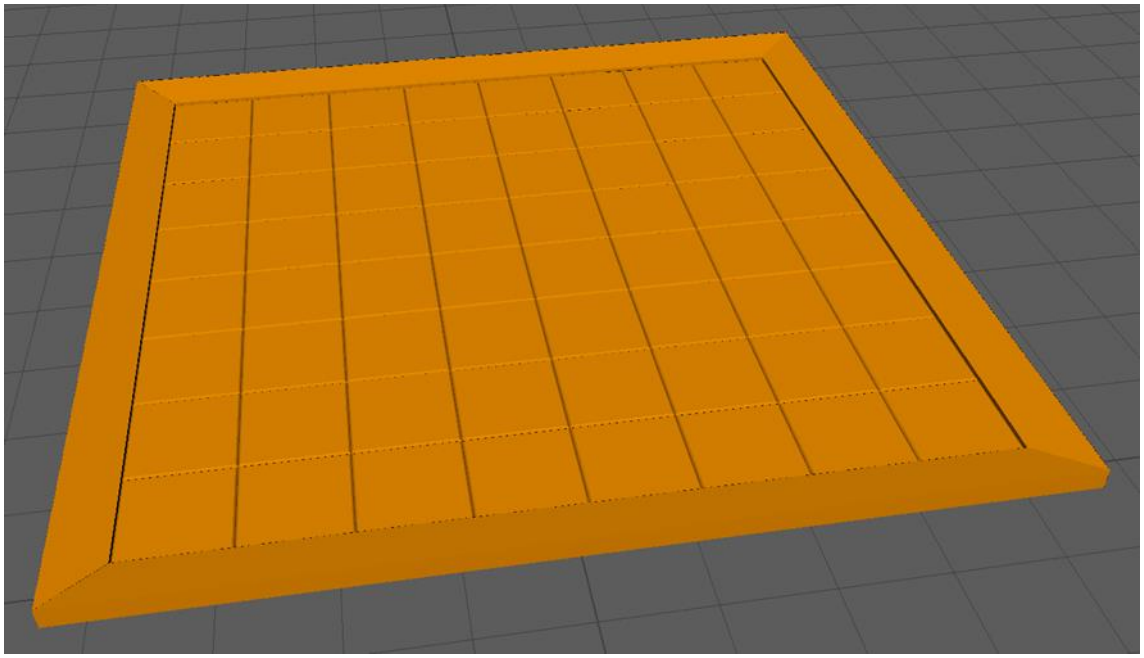

Figure 7. Final model of the board

2.3.2. Texturization of the chess pieces and the chessboard

For the texture of the pieces and the chessboard, I used the Substance Painter program, but previously I had to map the UV of each of the figures in Autodesk Maya. To do this I used the *Unwrap* technique, used in any 3D design program to texture a model, it consists in unfolding the whole surface of the model in a two-dimensional plane where a flat texture can be added, and in this way we can make a flat image wrap the figure.

Autodesk Maya has anautomatic tool to make this task easier, but the results are not very optimal in most cases causing that you have to do it manually for each figure. An example between the use of the automatic tool and a manual *Unwrap* can be seen in the figure 8.

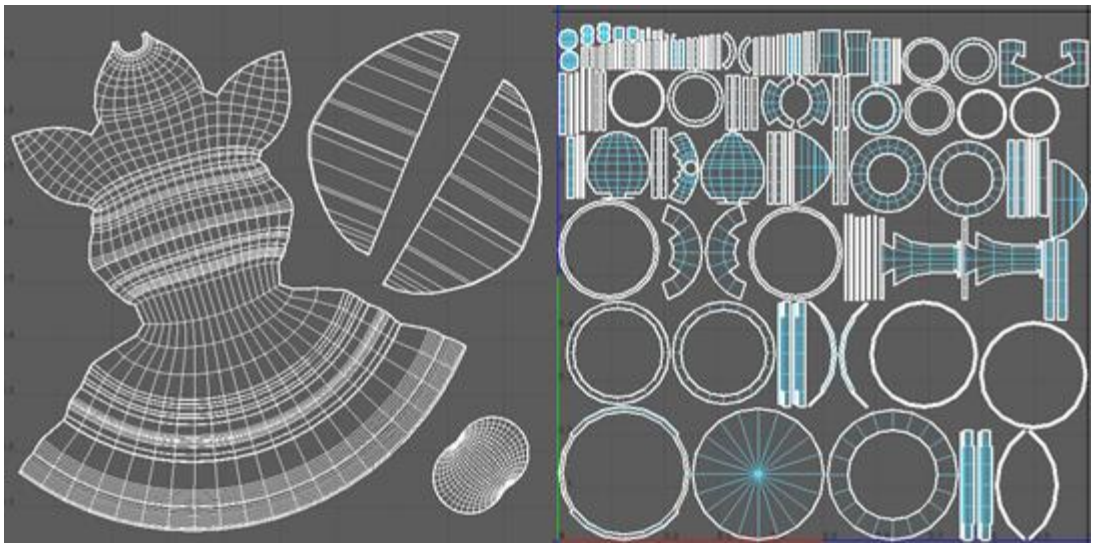

Figure 8. Left: Manual Unwrap. Right: Automatic Unwrap.

If we make a bad *Unwrap* of the model, there will be some *Mark Seams* left that will directly affect the texture of the objects. The *Mark Seams* (figure 9) are the marks that remain visible by dividing the sections of the surface of the model; these marks will result in the creation of shadows and imperfections that we can only solve inverting a large amount of time in the correction of the textures.

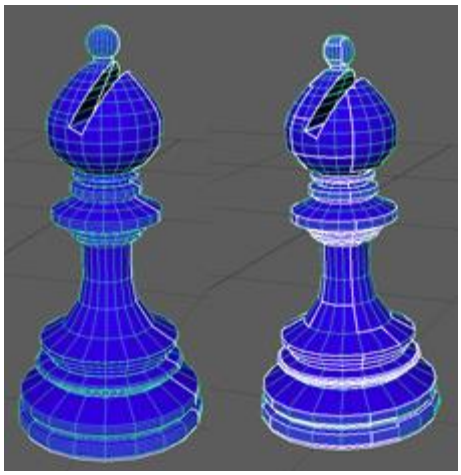

Figure 9. Left: Mark Seams for a manual Unwrap. Right: Mark Seams for automatic Unwrap.

After the *Unwrap* of the figures, we can already use the Substance Painter giving a texture to the figure. For this occasion I have used a smart material, specifically *Wood Beech Veined* for the white pieces and *Wood Walnut* for the black pieces, the same material is used for the board adding *Wood Ship Hull Nordic* for the edge of the board.

The final result of the white pieces is found in figure 10, the black pieces in figure 11 and the board in figure 12.

Figure 10. Final result of the white pieces

Figure 11. Final result of the black pieces

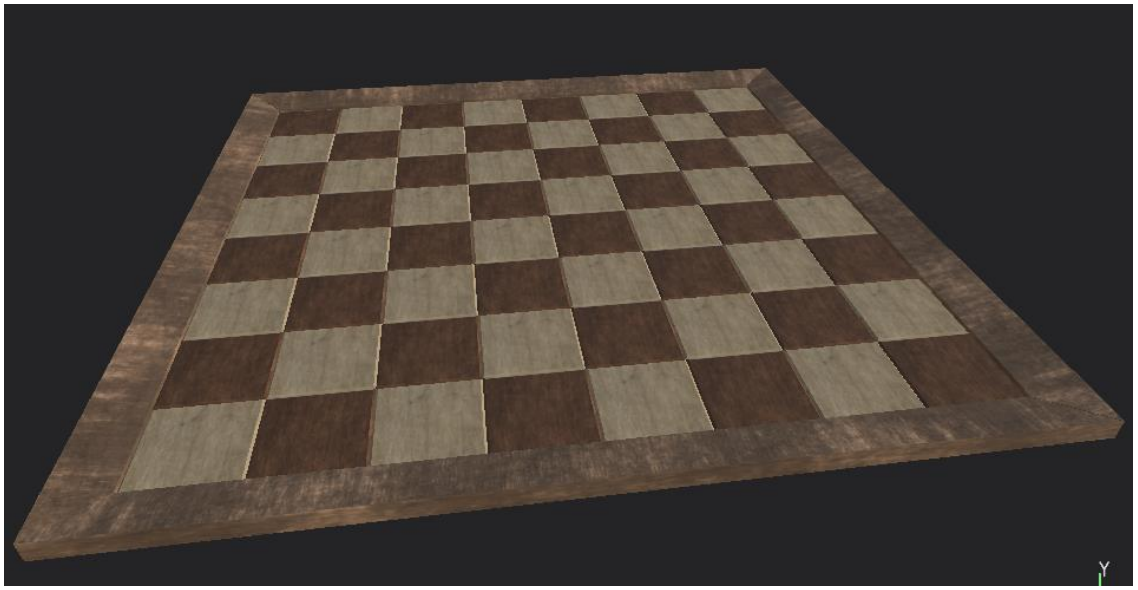

Figure 12. Final result of the board

2.3.3. First version of the game

Due to my lack of knowledge in programming with the C# language and using the Unity software, this first version is based on the Chess Game Tutorial made by Michael Doyon [N3K EN as nickname] that can be found on YouTube. It is a very complete tutorial that helped me to learn the language as well as the operation of Unity. At the end of the tutorial, I was able to

transform the code provided it in the tutorial to adapt it to the needs of this project but the base is very similar to his proposed solution.

I will add a note at the beginning of each sub-chapter whether that script is based on the code Michael Doyon [N3K EN] offers as a solution. As you can see, both in this version and in the following ones, many scripts created from scratch reuse some sentences that he teaches but in another way to create new functionalities.

For this first version, the game was created like a multiplayer computer game: the camera was fixed, the inputs that received the code came from the mouse and you could only play one player against another player.

2.3.3.1. Project creation

When you create a new project in Unity, the scene appears completely empty with a camera located at the origin (figure 13). Also a simple tree of folders is created and you can increase and order it as you want, all the files that are inside these folders can be used by the Unity project.

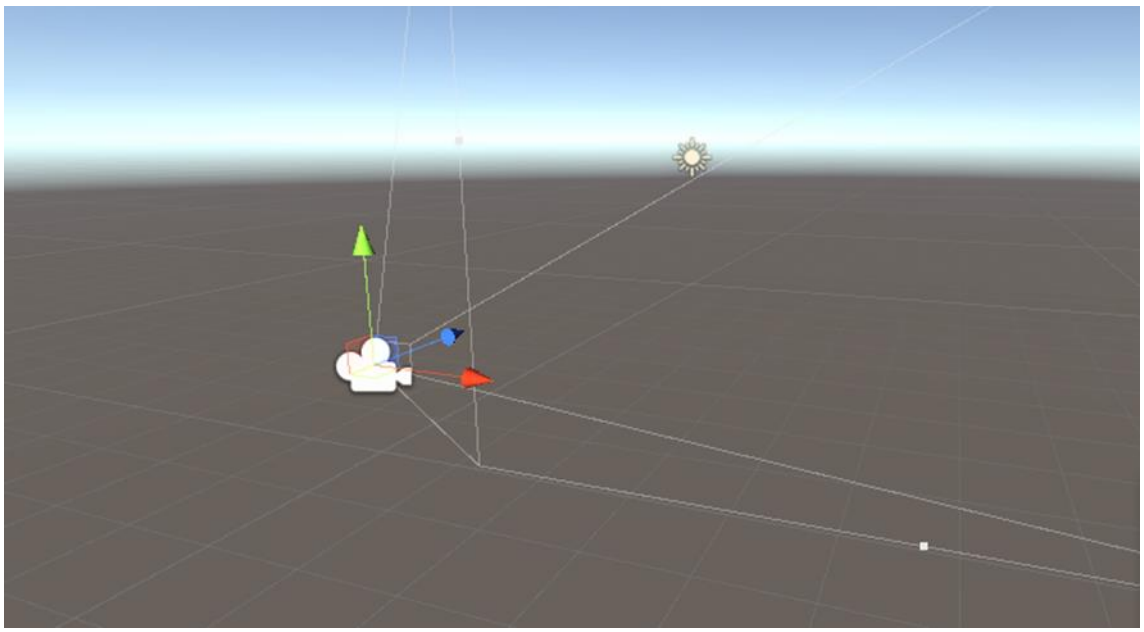

Figure 13. Exemple of a new Unity project

2.3.3.2. Import models

The first step to start a project in Unity is to import the models created in Autodesk Maya. To make this step easier, Autodesk Maya has the option to extract each of the figures and save them as a "3D object"; this type of file preserves the original shape and saves the texture associated with it.

In order to make use of these files within the project, the object has to be dragged from the folder that is saved to the scene. When the object is placed inside the scene, the figure changes the file type to *Prefab Asset*. After the file type is changed, you can return to drag that

figure to the final folder of the project (figure 14). This change of format is necessary to be able to call the object from the code; otherwise it would be an inert object.

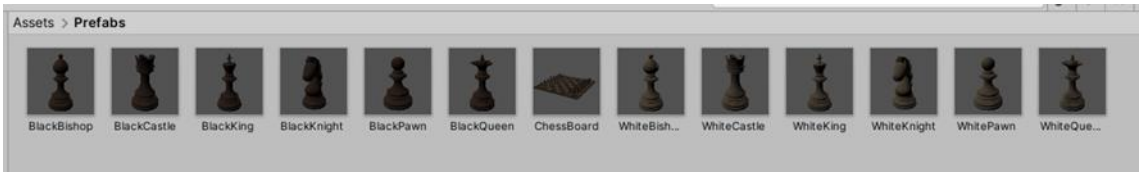

Figure 14. Prefabs assets of the project

In this same step it is also necessary to place the *Layer Mask "ChessPlane"* which will be used to interact with the *Raycast Point*. This *Layer Mask* is a square of 8 meters by 8 meters totally invisible located exactly where the board will go; to facilitate its location in the code; one of the corners is at the origin while the sides are parallel to the z axis and the x axis.

2.3.3.3. Global variables

These variables are declared as global because they are widely used throughout the program and have to be accessible from any part of the project. Many of them are declared and updated from different functions.

The global variables that we will use in this first version are the ones we find in the table 1.

Name and type of the variable	Definition of the variable
bool[,] possibleMovements { set; get; }	It is a boolean matrix that keeps all the possible movements of the selected piece. It is updated every turn when you access to the *SelectPiece* function.
Chessman[,] Chessmans { set; get; }	It is a *Chessman* type matrix that keeps all the pieces and their positions. It is initialized in the *SpawnChessmans* function and updated in *MovePiece*.
Chessman selectedChessman;	It is a *Chessman* variable that keeps the selected chessman. This variable is updated every turn when you access the *SelectPiece* function
const float SQUARE_SIZE = 1.0f; const float SQUARE_OFFSET = 0.5f;	These are the constants that mark, on the one hand, the width of a box (*SQUARE_SIZE*) and, on the other hand, the difference until reaching the centre of it (SQUARE_*OFFSET*).
int xSelection = -1; int zSelection = -1;	These are the variables that will store the position of the mouse on the X axis as well as on the Z axis. Both are initialized to -1 when the game starts.
List<GameObject> chessPiecesPrefabs;	This GameObject type list links the graphic section with the code, positioning each piece in an index.
List<GameObject> livingPieces;	This GameObject type list keeps all the pieces that are still active in the game. It is initialized in the *SpawnChessmans* function and updated in *MovePiece*.

int[] enPassantMove { set; get; }	This int type matrix stores the last move made by a pawn in the case that it advances two squares, exactly memorize the position that is skipped after advancing these two squares. It is initialized and updated in the *MovePiece* function.
bool isWhiteTurn = true;	It's a Boolean that indicates if it is black's turn or white's. It is updated in the *MovePiece* function. It initializes as true because it always starts with white moving.
List<GameObject> selectionLightsPrefabs;	This GameObject type list links the graphic section with the code, positioning the lights in an index.
List<GameObject> selectionLights = new List<GameObject>();	This GameObject type list will serve to update the light position in order to graphically indicate where the mouse is located. It is initialized in the *SpawnLight* function and updated inside the function *LightCursorPosition*.
private int startDragx, startDragz;	These variables will be used to save the start of the animation when we select a piece. They will be initialized and updated inside the *SelectPiece* function.

Table 1. Global variables of the first version of the game

2.3.3.4. Classes

Inheritance is an essential property of Object-Oriented Programming, which consists of creating new classes from existing ones. The classes that inherit from base classes are called as derived, and these can be a base class for other derived classes. In this case the base class is Chessman and all the other classes that refer to chess pieces are the classes derived from this one.

2.3.3.4.1. The Chessman class

Note: The Chessman class is based on the solution developed by Michael Doyon [N3K EN] on the YouTube video Chess Game Tutorial • 3/5 • [Tutorial][C#]. In this version the variable firstMove and the function FirsrMoveDone() has been added to my custom code.

The Chessman class is the basic class of the game, from this class we will derive to each one of the pieces. As we can see in the in the code below, the variables declared within this class are:

- CurrentX, CurrentZ: int variables that they save the current position in X and Z.
- isWhite: bool variable that indicates the chessman team.
- firstMove: bool variable that indicates if the piece has made the first move or not.

```
public abstract class Chessman : MonoBehaviour
{
    public int CurrentX { set; get; }
    public int CurrentZ { set; get; }
    public bool isWhite;
```

```
    public bool firstMove { set; get; } = true;

    public void FirstMoveDone()
    {
      firstMove = false;
    }

    public void SetPosition(int x, int z)
    {
      CurrentX = x;
      CurrentZ =z;
    }

    public virtual bool[,] PossibleMove()
    {
      return new bool[8,8];
    }

  }
```

On the other hand, we also find these functions:

- FirstMoveDone: updates the state of the variable *firstMove* to false.
- SetPosition: updates the position of the piece.
- PossibleMove: returns an 8x8 Boolean matrix that will indicate all the possible movements of that particular piece.

One of the options that derived classes have is to overwrite one of the functions of the base class, for that reason the function *PossiblesMoves* is only expressed with a return of an empty 8x8 Boolean matrix since this function will be overwritten by each one of its derived classes.

All derived classes inherit the variables from the base class and at least declare the *PossibleMoves* function which will return an 8x8 Boolean matrix with the boxes where the piece can access marked as true.

From now on, when we describe the classes of the different chess pieces we will only refer to the *PossibleMoves* function as it is the only function they will contain apart from the auxiliary functions that will allow evaluate positions without unnecessarily repeating code.

2.3.3.4.2. The Pawn class

Note: The Pawn class is based on the solution developed by Michael Doyon [N3K EN] on the YouTube video Chess Game Tutorial • 5/5 • [Tutorial][C#]. In this version the Pawn attack and the Pawn Movement have been modified from the tutorial version.

The variables used in this class are:

- Chessmans: global variable consisting of 8x8 matrix of *Chessman* type that contains the position of all pieces

- e: contains the global variable *inPassantMove*. This global variable registers the last pawn moved, allowing the passant capture if the piece is in the right option.
- r: an 8x8 Boolean matrix that will be the return of the *PossibleMove* function.
- c, c2: *Chessman* type auxiliary variables that will be used to evaluate the placement of a piece on a square.

Because of the peculiarity of the pawn's move, which can only be moved frontward and captured diagonally, the *PossibleMoves* function of this class differentiates whether you are evaluating the possible moves of a black piece or a white piece. The difference is that if the piece you are evaluating is white you increase the counters, while if the piece is black you decrease the counters; an example is shown in the code piece below. Due to the similarity of the whole code I will only comment on white team's move and highlight the difference with black team's move.

```
//For white team
r[CurrentX, CurrentZ + 1] = true;
//For black team
r[CurrentX, CurrentZ - 1] = true;
```

The first condition that checks the function is whether the pawn can make an *en passant* capture. *En Passant* capture is a special move in which a pawn captures another pawn immediately after the pawn has moved two positions forward from its starting position. The capturing pawn will move to the square where the opponent's pawn would have been if it had only advanced one square.

Once it has been checked the *en Passant* move, the *PawnAttack* function is called twice with the coordinates of the boxes located diagonally in the upper row (for black pieces, the coordinates are those of the lower row). Right after that, it checks if a square can be moved forward by sending the coordinates of the upper row (or lower row for the black piece) to the *PawnMove* fuction.

```
if (CurrentX != 0 && e[0] == CurrentX - 1 && e [1] == CurrentZ + 1)
{
   r[CurrentX - 1, CurrentZ + 1] = true;
}

if (CurrentX != 7 && e[0] == CurrentX + 1 && e[1] == CurrentZ + 1)
{
   r[CurrentX + 1, CurrentZ + 1] = true;
}

PawnAttack(CurrentX - 1, CurrentZ + 1, ref r);
PawnAttack(CurrentX + 1, CurrentZ + 1, ref r);
PawnMove(CurrentX, CurrentZ + 1, ref r);
```

The *PawnAttack* function checks that the coordinates received are inside the board and, if they are, it updates *c* with the position on the board. It then checks to see if that square is occupied by an opponent's piece and if it is marks the position of *r[x, z]* as true.

The PawnMove function checks again if the coordinates are inside the board and if the marked square is occupied by a piece or not, if not it marks the square as true.

```
public void PawnAttack(int x, int z, ref bool[,] r)
{
    Chessman c;
    if (x >= 0 && x <= 7 && z >= 0 && z <= 7)
    {
        c = BoardManager.Instance.Chessmans[x, z];
        if (isWhite != c.isWhite)
        {
            r[x, z] = true;
        }
    }
}

public void PawnMove(int x, int z, ref bool[,] r)
{
    Chessman c;
    if (x >= 0 && x <= 7 && z >= 0 && z <= 7)
    {
        c = BoardManager.Instance.Chessmans[x, z];
        if (c == null)
        {
            r[x, z] = true;
        }
    }
}
```

The last condition to evaluates is the initial movement: in the case that your current position on Z is 1 (or 6 in the case of black pieces), check that there are no pieces placed on the square in front (*c*) or two beyond (*c2*). If both answers are null it means that you do not have any pieces in front of you and marks the option as true on the matrix which will later return.

```
    if (CurrentZ == 1)
    {
        c = BoardManager.Instance.Chessmans[CurrentX, CurrentZ + 1];
        c2 = BoardManager.Instance.Chessmans[CurrentX, CurrentZ + 2];
        if (c == null && c2 == null)
        {
            r[CurrentX, CurrentZ + 2] = true;
        }
    }
```

2.3.3.4.3. The Castle class

Note: The Castle class is based on the solution developed by Michael Doyon [N3K EN] on the YouTube video Chess Game Tutorial • 4/5 • [Tutorial][C#]. In this version part of the code has been rewritten from the tutorial version to be more effective.

This class does not use any other global variable apart of *Chessmans*. The local variables that it uses are:

- r: an 8x8 Boolean matrix that will be the return of the *PossibleMove* function.
- c: a Chessman type auxiliary variable that will be used to evaluate if a piece is located on a square or not.
- i: an int type counter that will allow us to check the column or the row of the rook.

The movement of the rook consists in that it can advance vertically or horizontally until it finds a piece, it can be captured if the chessman is from the opposing team. This movement has a very simple evaluation that consists of the same code repeated four times with these differences:

1- The counter is located in the *CurrentX* position and adds one unit in each iteration evaluating the *r[i, CurrentZ]* position. It values the positions located in the same column above the chessman.
2- The counter is located at the *CurrentX* position and subtracts one unit at each iteration by valuing the *r[i, CurrentZ]* position. It values the positions in the same column below the chessman.
3- The counter is located at the *CurrentZ* position and adds one unit in each iteration evaluating the *r[CurrentX, i]* position. It values the positions located in the same row on the left of the chessman.
4- The counter is located at the *CurrentZ* position and subtracts one unit at each iteration evaluating the *r[CurrentX, i]* position. It values the positions located in the same row on the right of the chessman.

Each of the iterations is repeated until it finds a piece or the end of the board, in the case that the piece belongs to the opponent it marks that square also as a possible move.

```
i = CurrentX;
while (true)
{
   i++;
   if (i >= 0 && i <= 7)
   {
      c = BoardManager.Instance.Chessmans[i, CurrentZ];
      if (c == null)
      {
         r[i, CurrentZ] = true;
      }
      else if (isWhite != c.isWhite)
      {
         r[i, CurrentZ] = true;
```

```
                break;
            }
            else
            {
                break;
            }
        }
        else
        {
            break;
        }
    }
```

2.3.3.4.4. The Knight class

Note: The Knight class is based on the solution developed by Michael Doyon [N3K EN] on the YouTube video Chess Game Tutorial • 4/5 • [Tutorial][C#]. In this version part of the code of the KnightMove function has been rewritten from the tutorial version to be more effective.

This class uses the global variable *Chessmans* and, as local variables, uses *c* as the auxiliary variable and *r* as the response matrix already named in other classes.

This class has the particularity that it only has 8 possible movements so, as you can see in the code below, it evaluates them one by one. It uses the *KnightMove* function where it evaluates if the sent position if it is inside the board; if it is, it checks if there is any piece; if it is, checks if there is another chessman; and if it is, if it is or not from the same team. If it is inside the board and it is an empty position or occupied by an enemy piece it marks the position *r[x, z]* as true.

```
public class Knight : Chessman
{
    public override bool [,] PossibleMove()
    {
        bool[,] r = new bool[8, 8];

        KnightMove(CurrentX - 1, CurrentZ + 2, ref r);
        KnightMove(CurrentX + 1, CurrentZ + 2, ref r);
        KnightMove(CurrentX - 1, CurrentZ - 2, ref r);
        KnightMove(CurrentX + 1, CurrentZ - 2, ref r);
        KnightMove(CurrentX - 2, CurrentZ - 1, ref r);
        KnightMove(CurrentX + 2, CurrentZ - 1, ref r);
        KnightMove(CurrentX - 2, CurrentZ + 1, ref r);
        KnightMove(CurrentX + 2, CurrentZ + 1, ref r);

        return r;
    }

    public void KnightMove(int x, int z, ref bool[,] r)
    {
        Chessman c;
        if (x >= 0 && x <= 7 && z >= 0 && z <= 7)
```

```
    {
        c = BoardManager.Instance.Chessmans[x, z];
        if(c == null || isWhite != c.isWhite)
        {
            r[x, z] = true;
        }
    }
  }
}
```

2.3.3.4.5. The Bishop class

Note: The Bishop class is based on the solution developed by Michael Doyon [N3K EN] on the YouTube video Chess Game Tutorial • 4/5 • [Tutorial][C#]. In this version part of the code has been rewritten from the tutorial version to be more effective.

This class evaluates possible movements in a very similar way to the Tower class using two counters instead of one because of the particularity that it moves diagonally. Apart from the global variable *Chessmans* also uses the following local variables:

- r: an 8x8 Boolean matrix that will be the return of the *PossibleMove* function.
- c: a Chessman type auxiliary variable that will be used to evaluate if a piece is located on a square or not.
- i, j: an int type counter that will allow us to check the diagonals.

The movement of the bishop consists that it can advance diagonally until it finds a piece and, if the piece is contrary, can capture it. To evaluate this movement we find the same code repeated four times with these differences:

1- *i* counter subtracts one unit in each iteration while *j* adds one unit. Values the positions located in the top left diagonal.
2- Both *i* and *j* counters subtract one unit in each iteration. Values the positions located in the bottom left diagonal.
3- *i* counter adds one unit in each iteration while *j* subtracts one unit. Values the positions located in the bottom right diagonal.
4- Both *i* and *j* counters add up to one unit in each iteration. Values the positions located in the top right diagonal.

In all versions both of the counters are located in *CurrentX* and *CurrentZ* respectively and the evaluated position in each iteration is *r[i, j]*. Each of the iterations is repeated until a piece or the end of the board is found, in the case that the piece belongs to the opponent it marks that square also as a possible move.

```
    i = CurrentX;
    j = CurrentZ;
    while (true)
    {
        i--;
```

```
        j++;
        if (i >= 0 && i <= 7 && j >= 0 && j <= 7)
        {
            c = BoardManager.Instance.Chessmans[i, j];
            if (c == null)
            {
                r[i, j] = true;
            }
            else if (isWhite != c.isWhite)
            {
                r[i, j] = true;
                break;
            }
            else
            {
                break;
            }
        }
        else
        {
            break;
        }
    }
```

2.3.3.4.6. The Queen class

Note: The Queen class is based on the solution developed by Michael Doyon [N3K EN] on the YouTube video Chess Game Tutorial • 4/5 • [Tutorial][C#]. In this version part of the code has been rewritten from the tutorial version to be more effective.

The particularity of the queen is that it moves like a rook and a bishop at the same time, so the resulting matrix to the evaluation of the board consists of the addition of the conditions of these two classes without any additions.

2.3.3.4.7. The King class

Note: The King class is based on the solution developed by Michael Doyon [N3K EN] on the YouTube video Chess Game Tutorial • 4/5 • [Tutorial][C#] . In this version the move of the King has been modified from the tutorial version and the castling movement has been added to my custom code.

The King's move is exactly the same as the Queen's but taking into account that only moves one square in each direction. For this reason the King has only 8 possible movements so, as you can see in the code below, it evaluates them one by one as in the Knight class.

On the other hand we also find a special movement for the King: *castling*. *Castling* consists of moving the King two squares along the first rank toward a Rook and then placing the Rook on the last square that the king just crossed. *Castling* can be done if the King and the Rook involved have never been moved, the squares between the King and the Rook involved are

unoccupied, the King is not in check, and none of the squares the King will pass through are under attack.

The code snippet below reviews these conditions by reusing the counter *i* and the auxiliary variable *c* used in the basic movement of the King.

```
public override bool[,] PossibleMove()
{
    bool[,] r = new bool[8, 8];

    Chessman c;
    int i;

    KingMove(CurrentX - 1, CurrentZ + 1, ref r);
    KingMove(CurrentX, CurrentZ + 1, ref r);
    KingMove(CurrentX +1, CurrentZ + 1, ref r);
    KingMove(CurrentX - 1, CurrentZ, ref r);
    KingMove(CurrentX + 1, CurrentZ, ref r);
    KingMove(CurrentX - 1, CurrentZ - 1, ref r);
    KingMove(CurrentX, CurrentZ - 1, ref r);
    KingMove(CurrentX + 1, CurrentZ - 1, ref r);

    c = BoardManager.Instance.Chessmans[CurrentX, CurrentZ];
    if (c.firstMove == true)
    {
        i = 1;
        while (i < 4)
        {
            c = BoardManager.Instance.Chessmans[CurrentX - i, CurrentZ];
            if (c != null)
            {
                break;
            }
            i++;
        }
        c = BoardManager.Instance.Chessmans[CurrentX - 4, CurrentZ];
        if (c != null)
        {
            if (i == 4 && c.firstMove == true)
            {
                r[2, CurrentZ] = true;
            }
        }
        i = 1;
        while (i < 3)
        {
            c = BoardManager.Instance.Chessmans[CurrentX + i, CurrentZ];
            if (c != null)
            {
                break;
            }
```

```
            i++;
        }
        c = BoardManager.Instance.Chessmans[CurrentX + 3, CurrentZ];
        if (c != null)
        {
            if (i == 3 && c.firstMove == true)
            {
                r[6, CurrentZ] = true;
            }
        }
    }
    return r;
}

public void KingMove(int x, int z, ref bool[,] r)
{
    Chessman c;
    if (x >= 0 && x <= 7 && z >= 0 && z <= 7)
    {
        c = BoardManager.Instance.Chessmans[x, z];
        if (c == null || isWhite != c.isWhite)
        {
            r[x, z] = true;
        }
    }
}
```

2.3.3.5. BoardManager script

Note: The BoardManager script is based on the solution developed by Michael Doyon [N3K EN] on the YouTube video Chess Game Tutorial • 5/5 • [Tutorial][C#] . In this version a lot of code has been modified from the tutorial version and some more has been added to my custom script, you will find a note indicating if it is extracted from the tutorial or not.

The main part of this project goes through the *BoardManager* script that contains mostly all the code that the game needs to be played except of the pieces classes and the type of lights.

The creation of any new script is done with two functions that will be important throughout the writing of the code: *Start* and *Update*. The *Start* function will only be called once when the project is started, but the *Update* function will be updated in each frame allowing the recursive use of the functions inside. For that reason, the *Start* function will contain all the code needed to start the game while the *Update* function will contain all the other functions needed to play.

2.3.3.5.1. The Start function

Note: The Start function is based on the solution developed by Michael Doyon [N3K EN] on the YouTube video Chess Game Tutorial • 1/5 • [Tutorial][C#]. SpawnLight function has been added to my custom code.

The *Start* function starts by making an instance of the *Start* function itself allowing any method from any part of a program to gain unrestricted and unchecked access to the internal mechanisms of an object; in this way the different functions can interact with each other without needing to request permission.

Furthermore, it calls the *SpawnChessmans* function that places all the pieces on the chessboard in the initial position and, finally, it also calls the *SpawnLight* function that places a light that will be used as a pointer to see where our mouse is located on the chessboard.

```
private void Start()
{
    Instance = this;
    SpawnChessmans();
    SpawnLight();
}
```

2.3.3.5.2. The Update function

<u>Note:</u> The Update function is based on the solution developed by Michael Doyon [N3K EN] on the YouTube video Chess Game Tutorial • 1/5 • [Tutorial][C#]. LightCursorPosition function has been added to my custom code.

The *Update* function is the main function of the code and is updated every frame. The functions it calls without any conditions are *BoardCursorPosition* and *LightCursorPosition*, these functions keep the mouse position updated in coordinates as well as graphically.

If a piece is selected (*selectedChessman != null*) we call the function *UpdateChessmanDrag* with the selected piece as input parameter. This function makes the selected piece chase the mouse wherever it moves while it is inside the chessboard.

In the case that the left mouse button is pressed (*Input.GetMouseButtonDown(0)*), the function checks if it is located on the chessboard. If it does not have any selected piece, it calls the function *SelectPiece* sending as input the location parameters of the mouse saved into the *xSelection* and *zSelection* variables.

When the left mouse button is released (*Input.GetMouseButtonUp(0)),* the functions confirms if the mouse is located on the board and if is selected any chessmans (*selectedChessman != null*). If it does, it calls the function *MovePiece* passing as an input *xSelection* and *zSelection* parameters that indicate again the actual position of the mouse on the chessboard.

```
private void Update()
{
    BoardCursorPosition();
    LightCursorPosition();

    if (selectedChessman != null)
    {
        UpdateChessmanDrag(selectedChessman);
    }
```

```
    if (Input.GetMouseButtonDown(0))
    {
        if (xSelection >= 0 && zSelection >= 0 && xSelection < 8 && zSelection < 8)
        {
            if (selectedChessman == null)
            {
                SelectPiece(xSelection, zSelection);
            }
        }
    }

    if (Input.GetMouseButtonUp(0))
    {
        if (xSelection >= 0 && zSelection >= 0 && xSelection < 8 && zSelection < 8)
        {
            if (selectedChessman != null)
            {
                MovePiece(xSelection, zSelection);
            }
        }
    }
}
```

2.3.3.5.3. The BoardCursorPosition function

Note: The BoardCursorPosition function is the same of the tutorial developed by Michael Doyon [N3K EN] on the YouTube video Chess Game Tutorial • 1/5 • [Tutorial][C#], in future versions it will be modified to suit the needs of this project.

This function places the mouse inside the scene; it creates a *Raycast Hit* on the *Layer Mask* that will indicate the exact box where the mouse is. The *Raycast* is an invisible vector that connects two points, in this case the mouse and the center of the camera and that extends until it hits an object, in this case the *ChessPlane* layer mask that refers to the chessboard.

The variables used by the function are:

- hit: this Raycast variable will create a Vector3 hit point above the *Layer Mask*.
- xSelection, zSelection: these global variables will place the mouse over the board in the X axis and the Z axis, and will be updated frame by frame.

```
private void BoardCursorPosition()
{
    if (!Camera.main) return;
    RaycastHit hit;

    If(Physics.Rayast(Camera.main.ScreenPointToRay(Input.mousePosition), out hit, 50.0f,
layerMask.GetMask("ChessPlane")))
    {
        xSelection = (int)hit.point.x;
```

```
        zSelection = (int)hit.point.z;
      }
      else
      {
        xSelection = -1;
        zSelection = -1;
      }
    }
```

As we can see in the in the above code, if it does not find the main camera it returns the function without allowing any selection; this control step is because the reference camera is needed to create the *Raycast*.

After the first check, it calculates the possible *Raycast hit* above the *ChessPlane* layer ma*sk*. If the mouse is placed over *ChessPlane* it updates the global variables *xSelection* and *zSelection*, otherwise it leaves the values as -1 in both variables to avoid conflicts with the rest of the code.

The function converts the *Raycast Hit Point* into an int variable. This is because the *ScreenPointToRay* function returns decimals and for the movement of the pieces it is more convenient to use integers, in this way the pieces can only be found between the 0 and 7 position for both the X and Z axis.

2.3.3.5.4. The UpdateChessmanDrag function

<u>Note:</u> The UpdateChessmanDrag function is made from scratch by applying knowledge acquired through the tutorial developed by Michael Doyon [N3K EN], in future versions it will be modified to suit the needs of this project.

This function has a very similar structure to the *BoardCursorPosition* function, but in this case it receives as an input parameter the *c* variable of the *Chessman* type that refers to the selected piece.

This selected piece transforms its position to the current one of the mouse but moving it up one unit in relation to the X-Z plane to avoid collisions with the other figures. In case that the mouse leaves the chessboard, the piece will remain suspended on the edge of the chessboard waiting for the mouse to enter the chessboard again.

```
  private void UpdateChessmanDrag(Chessman c)
  {
    if (!Camera.main) return;
    RaycastHit hit;

     if (Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition), out hit, 50.0f,
LayerMask.GetMask("ChessPlane")))
    {
      c.transform.position = hit.point + Vector3.up;
    }
  }
```

2.3.3.5.5. The LightCursorPosition function

Note: At the beginning the LightCursorPosition function is the same solution developed by Michael Doyon [N3K EN], but later it is updated to provide visual value to the player and added to my custom code.

This function acts as a visual guide to locate objects on top of the *ChessPlane* layer mask, it has not use in the implementation of the project although later it will be recycled for another purpose. It is a function that can be found in the *Update* function and uses these variables:

- xSelection, zSelection: as mention before, these global variables will place the mouse over the board in the X axis and the Z axis.
- widthLine, heightLine: these Vector3 variables have the size and direction of the horizontal and vertical lines that separate the cells of the chessboard.

As we can see in the code below, it places some dividing lines drawing each of the boxes. On the other hand, in the case that the cursor is over the board it draws an X in the square where it indicates.

```
private void LightCursorPosition()
{
    Vector3 widthLine = Vector3.right * 8;
    Vector3 heightLine = Vector3.forward * 8;

    for (int i = 0; i <= 8; i++)
    {
        Vector3 start = Vector3.forward * i;
        Debug.DrawLine(start, start + widthLine);

        for (int j = 0; j <= 8; j++)
        {
            start = Vector3.right * j;
            Debug.DrawLine(start, start + heightLine);
        }
    }

    if(xSelection >= 0 && zSelection >= 0 && xSelection < 8 && zSelection < 8)
    {
        Debug.DrawLine(
            Vector3.forward * zSelection + Vector3.right * xSelection,
            Vector3.forward * (zSelection + 1) + Vector3.right * (xSelection + 1));
        Debug.DrawLine(
            Vector3.forward * (zSelection + 1) + Vector3.right * xSelection,
            Vector3.forward * zSelection + Vector3.right * (xSelection + 1));
    }

}
```

After using the *Debug.DrawLine* functions the lines of the board and the X will not be seen in the final result but they will be seen inside the developer's viewer "Scene", this can be seen in the figure 15.

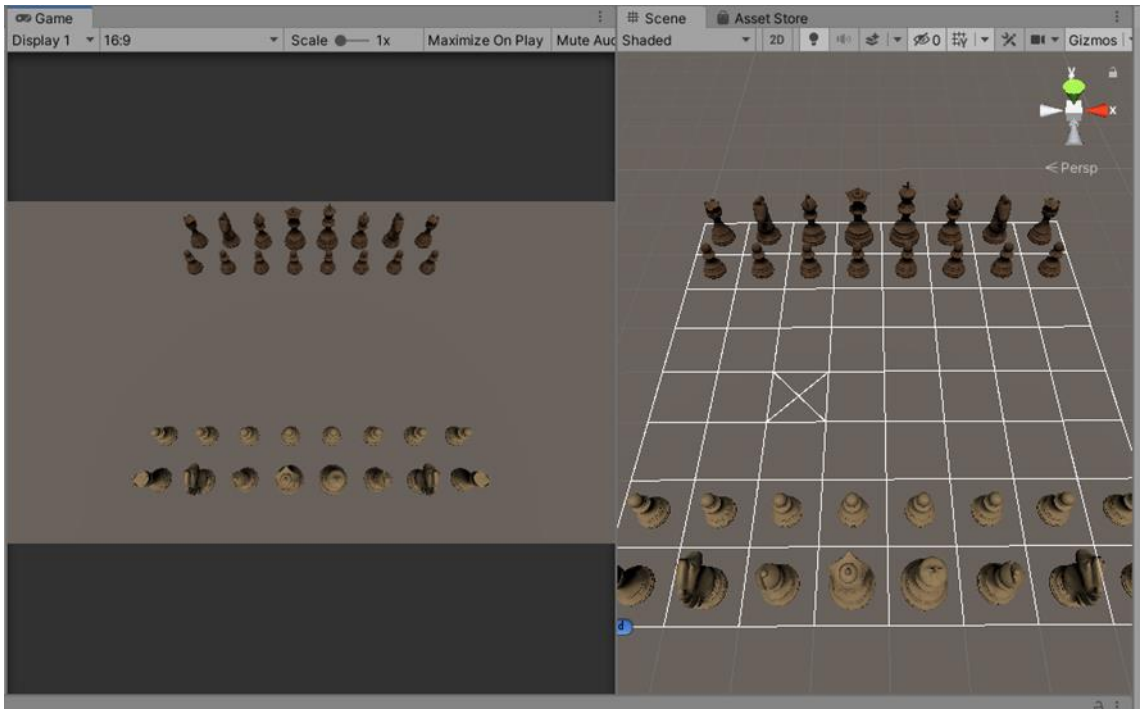

Figure 15. Left: Game view of the Debug.DrawLine. Right: Devekioer's view of th Debug.DrawLine.

At the end of the development of this first version, the function is replaced for something tangible for the player by graphically updating the position of the mouse on the board through the light *selectionLights[0]*.

```
private void LightCursorPosition()
{
    if (xSelection >= 0 && zSelection >= 0 && xSelection < 8 && zSelection < 8)
    {
        selectionLights[0].transform.position = MoveToTheCenter(xSelection, zSelection, -2);
    }
}
```

2.3.3.5.6. The SpawnChessmans function

Note: The SpawnChessmans function is the same one that can be found on the tutorial developed by Michael Doyon [N3K EN] on the YouTube video Chess Game Tutorial • 1/5 • [Tutorial][C#].

The SpawnChessmans function places the pieces at the start of the game; therefore it will only be called at the start of the game from the *Start* function or at the end of the game to play again in the *EndGame* function. It initializes three global variables:

- livingPieces: list of GameObjects that contains all the pieces that keep playing, for Black and for White.

- Chessmans: matrix of the class "Chessman" of 8x8, this matrix will store the position of all the pieces.
- enPassantMove: contains the last move of pawns that we have made to know if we can or not make the En Passant move.

```
private void SpawnChessmans()
{
    livingPieces = new List<GameObject>();
    Chessmans = new Chessman[8, 8];

    enPassantMove = new int[2] { -1, -1};

    SpawnChessman(0, 4, 0);
    SpawnChessman(1, 3, 0);
    SpawnChessman(2, 0, 0);
    SpawnChessman(2, 7, 0);
    SpawnChessman(3, 2, 0);
    SpawnChessman(3, 5, 0);
    SpawnChessman(4, 1, 0);
    SpawnChessman(4, 6, 0);
    for (int i= 0; i < 8; i++)
    {
        SpawnChessman(5, i, 1);
    }

    SpawnChessman(6, 4, 7);
    SpawnChessman(7, 3, 7);
    SpawnChessman(8, 0, 7);
    SpawnChessman(8, 7, 7);
    SpawnChessman(9, 2,7);
    SpawnChessman(9, 5, 7);
    SpawnChessman(10, 1, 7);
    SpawnChessman(10, 6, 7);
    for (int i= 0; i < 8; i++)
    {
        SpawnChessman(11, i, 6);
    }
}
```

To place the piece on the board call the *SpawnChessman* function giving as reference an index, a X position and a Z position. The indexes that we found refer to the global variable *ChessmanPrefabs* where all the pieces are placed as shown in figure 16.

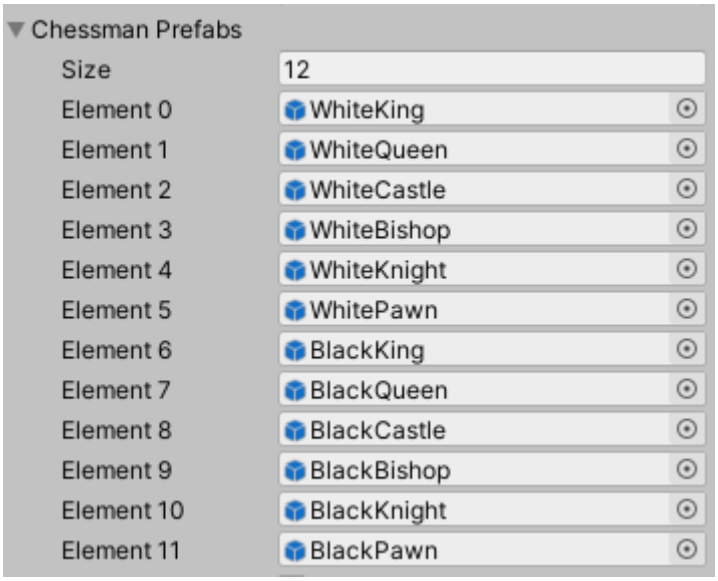

Figure 16. Indexes of the ChessmanPrefabs

2.3.3.5.7. The SpawnChessman function

Note: The SpawnChessman function is based on the solution developed by Michael Doyon [N3K EN] on the YouTube video Chess Game Tutorial • 1/5 • [Tutorial][C#], some adjustments have been made to adapt it to the needs of the project due to the original positions of the different pieces.

The SpawnChessman function initializes the GameObject *go*:

- Assigns it a piece referenced through the index.
- Calls the MoveToTheCenter function by sending the previously received positions (x and z) that will place the piece in the center of the square.
- Orients the figure.

As we can see in the code below, for all the pieces it uses the orientation of its own except for the pieces with the index 4 and 10; these indexes refer to the knights, turning them to face the enemy depending on if they are black (turn the piece 90 degrees) or white (turn the piece 270 degrees).

```
  private void SpawnChessman(int index, int x, int z)
  {
    if (index == 4 || index == 10)
    {
      if (index == 4)
      {
        GameObject go = Instantiate(chessmanPrefabs[index], MoveToTheCenter(x, z), Quaternion.Euler(0, 270, 0)) as GameObject;
        go.transform.SetParent(transform);
        Chessmans[x, z] = go.GetComponent<Chessman> ();
        Chessmans[x, z].SetPosition(x, z);
        livingPieces.Add(go);
      }
```

```
        else
        {
          GameObject go = Instantiate(chessmanPrefabs[index], MoveToTheCenter(x, z), Quaternion.Euler(0, 90, 0)) as GameObject;
          go.transform.SetParent(transform);
          Chessmans[x, z] = go.GetComponent<Chessman>();
          Chessmans[x, z].SetPosition(x, z);
          livingPieces.Add(go);
        }
      }
      else
      {
        GameObject go = Instantiate(chessmanPrefabs[index], MoveToTheCenter(x, z), Quaternion.identity) as GameObject;
        go.transform.SetParent(transform);
        Chessmans[x, z] = go.GetComponent<Chessman>();
        Chessmans[x, z].SetPosition(x, z);
        livingPieces.Add(go);
      }
    }
```

Once the part is initialized, it establishes its parent tree, places the piece inside the *Chessmans* matrix, updates the position for that piece inside the matrix, and finally adds it to the *livingPieces* list.

2.3.3.5.8. The MoveToTheCenter function

Note: The MoveToTheCenter function is based on the solution developed by Michael Doyon [N3K EN] on the YouTube video Chess Game Tutorial • 1/5 • [Tutorial][C#], some adjustments have been made to adapt it to the needs of the project due to the original positions of the different pieces and lights.

The *MoveToTheCenter* function receives the x and z position, places a vector in the center of the square and returns this vector indicating the position of the piece. To achieve this it uses the constants *SQUARE_SIZE* (with a value equal to 1) and *SQUARE_OFFSET* (with a value equal to 0.5); these constants only indicate the width of the square (*SQUARE_SIZE*) and the distance to the center of the square (*SQUARE_OFFSET*), in this way the rest of the program uses integers but actually places the piece 0.5m further away from the reference used by the rest of the code.

```
  private Vector3 MoveToTheCenter(int x, int z, int y)
  {
    Vector3 origin = Vector3.zero;
    origin.x += (SQUARE_SIZE * x) + SQUARE_OFFSET;
    origin.y += (SQUARE_SIZE * y);
    origin.z += (SQUARE_SIZE * z) + SQUARE_OFFSET;
    return origin;
  }
```

Because of this function it was important to place the *object point* of the chessman in the center of the base.

2.3.3.5.9. The SelectPiece function

Note: The SelectPiece function is based on the solution developed by Michael Doyon [N3K EN] on the YouTube video Chess Game Tutorial • 5/5 • [Tutorial][C#], some adjustments have been made to adapt it to the needs of the project and AtLeastOne function is outsourced to be used by the AI in the following version.

The *SelectPiece* function is the function that marks a piece as the one chosen to be moved. The global variables it uses are *Chessmans, possibleMovements* and *selectedChessman*; it also uses the local variable *hasAtleastOneMove* that will be used as an indicator of some possible movement by the selected piece.

As we can see the code below, the function receives a position: *x* and *y*. The first step it takes is to check if there is a piece on that square and if that piece belongs to the moving team.

```
private void SelectPiece(int x, int z)
{
    if (Chessmans[x, z] == null || Chessmans[x, z].isWhite != isWhiteTurn)
    {
        return;
    }

    possibleMovements = Chessmans[x, z].PossibleMove();

    if (!AtLeastOne(possibleMovements))
    {
        return;
    }

    selectedChessman = Chessmans[x, z];

    startDragx = x;
    startDragz = z;

    BoardHighlights.Instance.HighlightAllowedMoves(possibleMovements);
}
```

Once confirmed that a part exists and is from the correct equipment, it calls the *PossibleMove* function to update the global variable *possibleMovements* which will then serve to check if there is any movement available. To do this, calls *AtLeastOne* function to check the *possibleMovements* matrix.

If it does not find any possible movement, it returns and doesn't allow to continue with the function. On the other hand, if it finds a possible movement it updates the global variable *selectedChessman* with the piece stored in *Chessmans[x, y]*; it updates the variables *startDragx*

and *startDragz* indicating where the piece's movement starts; and, finally, it marks with a light all the possible movements that the piece has.

2.3.3.5.10. The MovePiece function

Note: The MovePiece function is based on the solution developed by Michael Doyon [N3K EN] on the YouTube video Chess Game Tutorial • 5/5 • [Tutorial][C#], some adjustments have been made to adapt it to the needs of the project and castling movement has been added to my custom code.

The *MovePiece* function is the function that moves the piece selected in the SelectPiece function to the position that the inputs (*x* and *z*) marks.

The globale variables it uses and updates are: *Chessmans, selectedChessman, enPassantMove, isWhiteTurn, livingPieces* and *onePerTurn*. The global variables it only uses without updating are: *possibleMovements, startDragx* and *startDragz*. It also uses the local variable *c* that is a Chessman type variable that will be initialized with the value of the chess box that the inputs mark.

As you can see on the code piece below, the first step of this function is to check if the position where it wants to move is a valid option between those that have been previously stored in *possibleMovements*. Otherwise, it returns the selected piece to its original position stored in *startDragx* and *startDragz*. Whether it's a permitted move or not, it eliminates, all the lights of the possible movements of the selected card and updates the variable *selectedChessman* leaving it as null.

```
if (possibleMovements[x,z]) [ … ]
else
{
   selectedChessman.transform.position = MoveToTheCenter(startDragx, startDragz);
}
BoardHighlights.Instance.Hidehighlights();
selectedChessman = null;
```

If it is a valid position, it checks if there is a piece in that position and if that piece is from the same colour. In the case a piece is found and is of the opposite team, it checks if it is the king and, if so, call the *EndGame* function; in case it finds a piece that is not the king , remove the piece from the list of active pieces and destroys the object *c*.

```
if(c != null && c.isWhite != isWhiteTurn)
{
   if(c.GetType () == typeof(King))
   {
      EndGame();
      return;
   }
   livingPieces.Remove(c.gameObject);
   Destroy(c.gameObject);
}
```

Then the function checks if the piece to be moved is a Pawn. If it is, check if the move stored *inPassantMove* matches with the move that the Pawn wants to make; if it does, the function locates the piece in the extra square (in the case of black team) or in the lower square (for white team) and remove it. After that, it checks if the pawn has reached the final square (for white team) or the first square (for black team), and if it does, exchanges it for a Queen of its own colour.

```
if (selectedChessman.GetType() == typeof(Pawn))
{
   if (x == enPassantMove[0] && z == enPassantMove[1])
   {
      if (isWhiteTurn)
      {
         c = Chessmans[x, z - 1];
      }
      else
      {
         c = Chessmans[x, z + 1];
      }
      livingPieces.Remove(c.gameObject);
      Destroy(c.gameObject);
   }

   if (z == 7)
   {
      livingPieces.Remove(selectedChessman.gameObject);
      Destroy(selectedChessman.gameObject);
      SpawnChessman(1, x, z);
      selectedChessman = Chessmans[x, z];
   }

   if (z == 0)
   {
      livingPieces.Remove(selectedChessman.gameObject);
      Destroy(selectedChessman.gameObject);
      SpawnChessman(7, x, z);
      selectedChessman = Chessmans[x, z];
   }
}
```

Then, the function updates the variable *inPassantMove* to -1 to prevent errors and checks again if it is a Pawn that is moving. If it is, it checks that the selected Pawn is currently on the 1st row (for white pieces) or the 6th row (for black pieces) and that you want to move it to the 3rd row (for white Pawns) or the 4th row (for black Pawns); if both conditions are correct, the function updates the variable *inPassantMove* with the square skipped.

```
        enPassantMove[0] = -1;
        enPassantMove[1] = -1;
        if (selectedChessman.GetType() == typeof(Pawn))
        {
            if (selectedChessman.CurrentZ == 1 && z == 3)
            {
                enPassantMove[0] = x;
                enPassantMove[1] = z - 1;
            }
            else if (selectedChessman.CurrentZ == 6 && z == 4)
            {
                enPassantMove[0] = x;
                enPassantMove[1] = z + 1;
            }
        }
```

After the initial checks, the function removes the variable stored in *Chessmans* in the position where the selected chessmans is currently located; it places the piece with the new coordinates; it updates the value of the *CurrentX* and *CurrentZ* variables of the selected piece through the *SetPosition* function; and it stores the piece inside *Chessmans* in the new position.

In addition, as can be seen in the following code, it checks if the movement to be done is castling and, if it is, it also moves the Rook to its final position updating *Chessmans* in the process.

```
        if (selectedChessman.GetType() == typeof(King) && selectedChessman.firstMove ==
true)
        {
            if (x == 2)
            {
                Chessmans[0, selectedChessman.CurrentZ].transform.position =
MoveToTheCenter(3, selectedChessman.CurrentZ, 0);
                Chessmans[0, selectedChessman.CurrentZ].SetPosition(3,
selectedChessman.CurrentZ);
                Chessmans[3, selectedChessman.CurrentZ] = Chessmans[0,
selectedChessman.CurrentZ];
                Chessmans[0, selectedChessman.CurrentZ] = null;
            }
            if (x == 6)
            {
                Chessmans[7, selectedChessman.CurrentZ].transform.position =
MoveToTheCenter(5, selectedChessman.CurrentZ, 0);
                Chessmans[7, selectedChessman.CurrentZ].SetPosition(5,
selectedChessman.CurrentZ);
                Chessmans[5, selectedChessman.CurrentZ] = Chessmans[7,
selectedChessman.CurrentZ];
                Chessmans[7, selectedChessman.CurrentZ] = null;
            }
        }
```

Finally, it updates the internal *Chessman* variable *firstMove* through its own function *FirstMoveDone*; changes the colour turn; changes the value of *onePerTurn* to true; and eliminates the possible light that indicates the check to the king.

```
        if (selectedChessman.firstMove == true)
        {
           selectedChessman.FirstMoveDone();
        }

        isWhiteTurn = !isWhiteTurn;

        onePerTurn = true;
        KingLights.Instance.HideKingLights();
```

2.3.3.5.11. The SpawnLight function

Note: The SpawnLight function is made from scratch by applying knowledge acquired through the tutorial developed by Michael Doyon [N3K EN].

This function is called only once from the *Start* function and places the light that symbolizes the mouse in the game (*selectionLightsPrefabs[0]*) in the chess box [4,4]. It also turns the light -90º to orient it correctly in relation to the board because the light is originally parallel to the X-Z plane. Finally, it creates the matching relations and adds it to the GameObject selectionLights.

```
   private void SpawnLight()
   {
      GameObject go = Instantiate(selectionLightsPrefabs[0], MoveToTheCenter(4, 4, -2), Quaternion.Euler(-90, 0, 0)) as GameObject;
      go.transform.SetParent(transform);
      selectionLights.Add(go);
   }
```

2.3.3.5.12. The AtLeastOne function

Note: The AtLeastOnen function is made from scratch by applying knowledge acquired through the tutorial developed by Michael Doyon [N3K EN].

This function is called from *SelectPiece* function and from the AI, it confirms if there is any possible movement within a whole matrix of bools. It initializes the local variable *OneMove* to false and then reviews the whole *movements'* matrix looking for a true; in the case of finding it, it changes the local variable to true. Finally returns OneMove.

```
   private bool AtLeastOne (bool[,] movements)
   {
      bool OneMove = false;

      for (int i = 0; i < 8; i++)
      {
```

```
        for (int j = 0; j < 8; j++)
        {
            if (movements[i, j])
            {
                OneMove = true;
            }
        }
    }

    return OneMove;
}
```

2.3.3.5.13. The EndGame function

Note: The EndGame function is the same as the solution developed by Michael Doyon [N3K EN] on the YouTube video Chess Game Tutorial • 2/5 • [Tutorial][C#].

It also removes all the pieces that still active on the chessboard, removes the lights from the possible movements of the piece, that it is still selected, and restarts the game.

To restart the game it returns the value of the global variable *isWhiteTurn* to true and calls the *SpawnChessmans* function to place the pieces in the original position.

```
private void EndGame()
{
    foreach(GameObject go in livingPieces)
    {
        Destroy(go);
    }
    BoardHighlights.Instance.Hidehighlights();
    isWhiteTurn = true;
    SpawnChessmans();
}
```

2.3.4. Second version of the game

In this part of the project we found an update of the code that includes the implementation of an Artificial Intelligence (AI). This artificial intelligence is based on the MiniMax decision method.

The Minimax algorithm consists of choosing the best movement for the computer, assuming that the user will choose the one that is most damaging to the machine. To choose the best option this algorithm makes a search tree with all the possible moves, looking for that in the turn of the machine its best move is the best scored while if it is the turn of the user its best move is the worst scored. Because of the limitation of the Oculus Go processor, only two levels of depth will be considered.

This development also implements the *InCheck, WhereIsTheKing, InCheck, BlackMoves, BestPossibleScore, AllPieces, AllPossibleMoves, ValueChessmansPosition* and *Minimax*

functions as well as the upgrade of the *Update* function. All the new functions are made from scratch by applying knowledge acquired through the tutorial developed by Michael Doyon [N3K EN].

2.3.4.1. New global variables

To expand the number of game features new global variables have been added, these can be found in table 2.

Name and type of the variable	Definition of the variable
bool onePerTurn = true;	This is a boolean that is used to send the *InCheck* function and the *BlackMoves* function once per round. Its value is updated in the *Update* function and in *MovePiece*.
int kingX, kingZ;	These variables save the position of the King from the colour being played. The values are updated once per round inside the *WhereIsTheKing* function.
int[] bestOption { set; get; }	This variable stores the best play of the Artificial Intelligence (AI). It contains the origin chess box, the final chess box and the score of this move. This variable is updated in the *BlackMoves* function.
int[] auxBestOption { set; get; }	This variable will be used as a support for choosing the best play in the artificial intelligence. This variable is updated in the *MiniMax* function.

Table 2. Global variables added for the second version of the game

2.3.4.2. Upgrade of the Update function

This new version of the *Update* function maintains the format of the first one but adds two new conditions: *onePerTurn* and *isWhiteTurn*.

The *onePerTurn* condition is used to call the *InCheck* function and the *BlackMoves* function only once in the whole turn, preventing that the machine needs to perform the recursive calculations for each frame. In the case of the *BlackMoves* function, it will only be called in the case that it is the turn to move black pieces. On the other hand, if the *InCheck* function returns true it places a light in the position saved in the global variables *kingX* and *kingZ*.

Ones the function enters one time in the *onePerTurn* condition, it becomes false to avoid entering again.

In case it is white team's turn (*if (isWhiteTurn)*) the same functions will be performed as in the previous version, but this time allowing only the movement of the white pieces. With these new conditions, the user can only move the white pieces and the artificial intelligence can only move the black ones.

```
private void Update()
{
   BoardCursorPosition();
   LightCursorPosition();

   if (onePerTurn)
   {
      if (InCheck())
      {
         KingLights.Instance.KingCheckLight(kingX, kingZ);
      }

      onePerTurn = false;

      if (!isWhiteTurn)
      {
         BlackMoves();
      }
   }
   if (isWhiteTurn) [ … ]
}
```

2.3.4.3. The WhereIsTheKing function

The function *WhereIsTheKing* checks box by box the global variable *Chessmans* looking for the location of the King of the team of the turn. Once it found it, updates the global variables *kingX* and *kingZ* keeping the location of the King.

```
private void WhereIsTheKing()
{
   Chessman c;
   for (int i = 0; i < 8; i++)
   {
      for (int j = 0; j < 8; j++)
      {
         c = Chessmans[i, j];
         if(c != null)
         {
            if (c.GetType() == typeof(King) && c.isWhite == isWhiteTurn)
            {
               kingX = i;
               KingZ = j;
               return;
            }
         }
      }
   }
   return;
}
```

2.3.4.4. The InCheck function

This function is used to find out if the King is being threatened by an enemy piece and, therefore, if he has to protect himself in order to not lose the game. This function is called only once per turn from the *Update* function to avoid occupying the machine's processor with a calculation that only needs to be done once.

It starts by calling the function *WhereIsTheKing* to update the global variables *kingX* and *kingZ*, once it gets the current position of the king it checks each of the squares inside *Chessmans* looking if any of the enemy pieces has the king's square as an allowed move. In the case that a single enemy piece has that option it returns true, otherwise it returns false.

```
private bool InCheck()
{

   WhereIsTheKing();

   for(int i = 0; i < 8; i++)
   {
      for (int j = 0; j < 8; j++)
      {
         if (Chessmans[i,j] != null)
         {
            if (Chessmans[i, j].isWhite != isWhiteTurn)
            {
               possibleMovements = Chessmans[i, j].PossibleMove();
               if (possibleMovements[kingX, KingZ])
               {
                  return true;
               }
            }
         }
      }
   }
   return false;
}
```

2.3.4.5. The BlackMoves function

This function is the one that coordinates the movement of the AI, which in this case will be the movement of the black pieces.

It starts by hiding the lights that have been used to indicate the movement of the machine in the previous turn; then it initializes the *bestOption* variable as an array of five integers that updates with the value received by the *MiniMax* function.

Having updated the best movement that is able to calculate the AI, it places the lights marking the origin position and the final position of the machine to then send these parameters to *SelectPiece* and *MovePiece* and perform the movement.

It is necessary to place the lights first and then perform the movement because C# uses a sequential programming structure. This is because the AI lights are only removed within the *BlackMoves* and the *EndGame* functions, so if they were upside down and the AI won after removing them within the *EndGame* it would put them back at the end of the function leaving them visible for the next game.

```
private void BlackMoves()
{
    IALights.Instance.HideIALights();
    bestOption = new int[5] { -1, -1, -1, -1, 0 };
    bestOption = MiniMax(Chessmans, false, 1);
    IALights.Instance.IACheckLight(bestOption[0], bestOption[1]);
    IALights.Instance.IACheckLight(bestOption[2], bestOption[3]);
    SelectPiece(bestOption[0], bestOption[1]);
    MovePiece(bestOption[2], bestOption[3]);
}
```

2.3.4.6. The BestPossibleScore function

The function *BestPossibleScore* is used to calculate the value of the chessboard for each team. To get the calculation of the chessboard, it adds up the existence and the position of each piece for each team and then it makes a difference between them and sends the result as an answer.

This function receives as input parameters:

- board: 8x8 Chessman type matrix that will serve as a reference board.
- ForWhiteTeam: a Boolean indicating whether it is to be calculated for white or black pieces.

The local variables used are:

- c: Chessman type auxiliary variable used to check box by box the entire board.
- scorewhite, scoreblack: int type variable that keeps the scores of each team.
- positionList: int type list that keeps the response of the *ValueChessmansPosition* function although we will only be interested on the first value.

The operation of this function is based on the code that can be seen bellow. We can observe two loops that go through all the squares individually in the whole board checking if in that square (*c = board[i, j]*) a piece is positioned. If there is one, check the type of piece and the team that it belongs to, so that it can be stored in the corresponding score. The scores that have been given to each piece have been:

- King = 9000
- Queen = 90
- Castle = 50
- Knight = Bishop = 30
- Pawn = 10

These values are the ones supported by the grandmasters, multiplying them by 10 to avoid conflict with the score obtained when evaluating the position. There are other possible values depending on the school the master belongs to, but this version is the most common.

```
for (int i = 0; i < 8; i++)
{
   for (int j = 0; j < 8; j++)
   {
      c = board[i, j];
      if (c != null)
      {
         if (c.GetType() == typeof(King))
         {
            if (c.isWhite)
            {
               scorewhite = scorewhite + 90000;
            }
            else
            {
               scoreblack = scoreblack + 90000;
            }
         }
         if (c.GetType() == typeof(Queen)) [ … ]
         if (c.GetType() == typeof(Castle)) [ … ]
         if (c.GetType() == typeof(Knight)) [ … ]
         if (c.GetType() == typeof(Bishop)) [ … ]
         if (c.GetType() == typeof(Pawn)) [ … ]

         positionList = ValueChessmansPosition(board, i, j, i, j, c.isWhite);

         if (c.isWhite)
         {
            scorewhite = scorewhite + positionList[0];
         }
         else
         {
            scoreblack = scoreblack + positionList[0];
         }
      }
   }
}
```

Once the score has been added according to its existence, it is valued according to its positioning through the *ValueChessmansPosition* function sending as data the board, the position of the piece doubled and the colour of the piece. We send twice the position of the piece because this function is implemented to calculate two positions even if we only need one. When we get the position value, we add the score to the corresponding team.

When the whole board has been checked, a difference is made depending if it is to evaluate the score for white or for black and the result is sent.

```
    if (ForWhiteTeam)
    {
       return (scorewhite - scoreblack);
    }
    else
    {
       return (scoreblack - scorewhite);
    }
```

2.3.4.7. The AllPieces function

This function checks the board it receives as an input parameter for all pieces of one colour, when it finds a piece of that colour it saves its position on a list indicating first the column and then the row. When it finishes, it returns the list *iaPieces* with all the values.

```
private List<int> AllPieces(Chessman[,] board, bool ForWhiteTeam)
{
   List<int> iaPieces = new List<int>();

   for (int i = 0; i < 8; i++)
   {
      for (int j = 0; j < 8; j++)
      {
         if (board[i, j] != null)
         {
            if (board[i, j].isWhite == ForWhiteTeam)
            {
               iaPieces.Add(i);
               iaPieces.Add(j);
            }
         }
      }
   }

   return iaPieces;

}
```

2.3.4.8. The AllPossibleMoves function

It creates an integer list that stores all possible movements of a colour. The input parameters it receives are:

- Board: Chessman type matrix that contains a board.
- iaPieces: int list that contains the positioning of all the pieces of one colour.

To do this, it has a first loop that goes through the entire *iaPieces* list and in each iteration looks if that piece has any movement available. In the case of having a movement available, it keeps 4 values in a very specific order:

1- The initial X position of the piece, which is obtained from the *iaPieces* list.
2- The initial Z position of the piece, which is obtained from the *iaPieces* list.
3- The final X position of the piece, which is obtained from *i*.
4- The final Z position of the piece, which is obtained from *j*.

It creates a list with all the possible movements of the pieces for a colour, saving its original position and its possible final position.

```
private List<int> AllPossibleMoves(Chessman[,] board, List<int> iaPieces)
{
   List<int> iaOptions = new List<int>();

   for (int m = 0; m < iaPieces.Count; m = m + 2)
   {
      possibleMovements = board[iaPieces[m], iaPieces[m + 1]].PossibleMove();
      if(AtLeastOne(possibleMovements)){
         for (int i = 0; i < 8; i++)
         {
            for (int j = 0; j < 8; j++)
            {
               if (possibleMovements[i, j])
               {
                  iaOptions.Add(iaPieces[m]);
                  iaOptions.Add(iaPieces[m + 1]);
                  iaOptions.Add(i);
                  iaOptions.Add(j);
               }
            }
         }
      }
   }
   return iaOptions;
}
```

2.3.4.9. The ValueChessmansPosition function

This function is used to evaluate two positions for the same piece. The parameters it receives as input are:

- board: board where we will take the position of the piece.
- x0 and z0: indicate the original position of the piece.
- x1 and z1: indicate which position it wants to move.
- ForWhiteTeam: a bool that indicates if it has to be evaluated for white or for black pieces.

The local variables used are:

- kingMatrix, queenMatrix, castleMatrix, bishopMatrix, knightMatrix, pawnMatrix: 8x8 int matrixes that contains the positioning value for each piece evaluated as if it were for black team.

- c: auxiliary variable of *Chessman* type.
- valuePosition: int list that collects and returns the value of the first position and the second position.

The values that have been taken to rate the positioning of each piece are the ones supported by the grandmasters in the case of fixed values being used. Apart from these, there are other possible fixed values according to the school that the master belongs to, but this version is the most common. On the other hand, there are some matrixes that change their values as the game progresses.

As you can see in the next code, the first step is to register the piece to be evaluated from the board it receives. Then, the function checks if it is evaluating for White and, if it is, exchange the value of both *x0* and *x1* so that it fits the matrixes from where we will get the information. Finally, it checks the type of piece to be evaluated, searches for its value in the corresponding matrix and add it to the *valuePosition* variable that it will later return.

```
c = board[x0, z0];

if (ForWhiteTeam)
{
   if (x0 == 0)
   {
      x0 = 7;
   } [ … ]
   if (x1 == 0)
   {
      x1 = 7;
   } [ … ]
}

if (c.GetType() == typeof(King))
{
   valuePosition.Add(kingMatrix[x0, z0]);
   valuePosition.Add(kingMatrix[x1, z1]);
}
if (c.GetType() == typeof(Queen)) [ … ]
if (c.GetType() == typeof(Castle)) [ … ]
if (c.GetType() == typeof(Knight)) [ … ]
if (c.GetType() == typeof(Bishop)) [ … ]
if (c.GetType() == typeof(Pawn)) [ … ]

return valuePosition;
```

2.3.4.10. The Minimax function

This function calculates the best possible move for the machine trying to give to the player the worst for the next move. The parameters we receive as input are:

- board: initial board from where the best move will be calculated.

- ForWhiteTeam: indicates if it has to value for black or for white.
- iteration: number of iterations it takes.

Note: due the lack of time for the development of the project, this function only contemplates one iteration although it is planned to allow it doing multiple ones. For this reason the variable iteration has not been eliminated although it is not used in the function.

The local variables used are:

- auxBestOption: auxiliary list of int type that stores 5 values (*x0, z0, x1, z1* and *score*).
- AuxChessmans and AuxChessmans2: 8x8 *Chessman* type auxiliary matrixes.
- iaAuxList, iaAuxList2, iaAuxList3, iaAuxList4: int type auxiliary lists.
- score1, score2: int variables.

As we can see in the code below, this function starts initiating the *AuxChessmans* variable and *AuxChessmans2* copying the exact *board* values. Once we have the copy done, it looks for all the possible plays that a team has and stores them in *iaAuxList2*. To be able to use later the *iaAuxList*, where all the pieces of a team had been previously stored, it is cleaned so that no residual value remains. Then the iaAuxList4 is initialized to store the possible scores.

```
for (int i = 0; i < 8; i++)
{
   for (int j = 0; j < 8; j++)
   {
      if (board[i, j] != null)
      {
         AuxChessmans[i, j] = board[i, j];
         AuxChessmans2[i, j] = board[i, j];
      }
   }
}

iaAuxList = AllPieces(AuxChessmans, ForWhiteTeam);

iaAuxList2 = AllPossibleMoves(AuxChessmans, iaAuxList);

iaAuxList.Clear();

if (!ForWhiteTeam)
{
   iaAuxList4.Add(-1);
   iaAuxList4.Add(-1);
   iaAuxList4.Add(-1);
   iaAuxList4.Add(-1);
   iaAuxList4.Add(90000);
}
else
{
   iaAuxList4.Add(-1);
```

```
        iaAuxList4.Add(-1);
        iaAuxList4.Add(-1);
        iaAuxList4.Add(-1);
        iaAuxList4.Add(-90000);
    }
```

Then the function goes through all the possible moves stored in *iaAuxList2* by updating *Auxchessmans* and *Auxchessmans2* according to the possible move stored in that list. Afterwards, it stores all the possible moves of the opponent team in *iaAuxList3* for that particular move in *iaAuxList2*, and finally it initializes the score1 value considering that the opponent will always choose the best option of that possible move.

```
        AuxChessmans[iaAuxList2[a + 2], iaAuxList2[a + 3]] = AuxChessmans[iaAuxList2[a + 0], iaAuxList2[a + 1]];
        AuxChessmans[iaAuxList2[a + 0], iaAuxList2[a + 1]] = null;
        AuxChessmans2[iaAuxList2[a + 2], iaAuxList2[a + 3]] = AuxChessmans2[iaAuxList2[a + 0], iaAuxList2[a + 1]];
        AuxChessmans2[iaAuxList2[a + 0], iaAuxList2[a + 1]] = null;
        iaAuxList = AllPieces(AuxChessmans, !ForWhiteTeam);
        iaAuxList3 = AllPossibleMoves(AuxChessmans, iaAuxList);
        score1 = -900000;
```

For each move stored in *iaAuxList3*, it checks the score of each possible move for the opponent and stores the best one. As you can see in the code bellow, at the beginning *AuxChessmans2* is modified as it will be the board that we will send to evaluate and at the end it takes back the original values for the next iteration.

```
        for (int b = 0; b < iaAuxList3.Count; b = b + 4)
        {
            AuxChessmans2[iaAuxList3[b + 2], iaAuxList3[b + 3]] = AuxChessmans2[iaAuxList3[b + 0], iaAuxList3[b + 1]];
            AuxChessmans2[iaAuxList3[b + 0], iaAuxList3[b + 1]] = null;

            score2 = (BestPosibleScore(AuxChessmans2, !ForWhiteTeam));
            if (score2 > score1)
            {
                score1 = score2;
            }
            AuxChessmans2[iaAuxList3[b + 0], iaAuxList3[b + 1]] = AuxChessmans[iaAuxList3[b + 0], iaAuxList3[b + 1]];
            AuxChessmans2[iaAuxList3[b + 2], iaAuxList3[b + 3]] = AuxChessmans[iaAuxList3[b + 2], iaAuxList3[b + 3]];
        }
```

Once finished the *iaAuxList3* loop, the function checks if the score registered in *score1* is higher than the previous ones saved in *iaAuxList4*. If score1 is lower, the function cleans the *iaAuxList4* and adds the play where it is in the loop of *iaAuxList2* with the score of *score1*; if it

is the same, then the function adds that play with the score of *score1* to the *iaAuxList4* list. This way we can store all the worst possible user's moves in iaAuxList4.

```
        if (iaAuxList4[4] > score1)
        {
           iaAuxList4.Clear();
           iaAuxList4.Add(iaAuxList2[a + 0]);
           iaAuxList4.Add(iaAuxList2[a + 1]);
           iaAuxList4.Add(iaAuxList2[a + 2]);
           iaAuxList4.Add(iaAuxList2[a + 3]);
           iaAuxList4.Add(score1);
        }
        else if (iaAuxList4[4] == score1)
        {
           iaAuxList4.Add(iaAuxList2[a + 0]);
           iaAuxList4.Add(iaAuxList2[a + 1]);
           iaAuxList4.Add(iaAuxList2[a + 2]);
           iaAuxList4.Add(iaAuxList2[a + 3]);
           iaAuxList4.Add(score1);
        }
```

To finish the iteration of the iaAuxList2 loop, the function returns *AuxChessmans* and *AuxChessmans2* to the original board values and clean *iaAuxList* and *iaAuxList3* in order to be used at the next iteration.

```
        AuxChessmans[iaAuxList2[a + 0], iaAuxList2[a + 1]] = board[iaAuxList2[a + 0], iaAuxList2[a + 1]];
        AuxChessmans[iaAuxList2[a + 2], iaAuxList2[a + 3]] = board[iaAuxList2[a + 2], iaAuxList2[a + 3]];
        AuxChessmans2[iaAuxList2[a + 0], iaAuxList2[a + 1]] = board[iaAuxList2[a + 0], iaAuxList2[a + 1]];
        AuxChessmans2[iaAuxList2[a + 2], iaAuxList2[a + 3]] = board[iaAuxList2[a + 2], iaAuxList2[a + 3]];

        iaAuxList.Clear();
        iaAuxList3.Clear();
```

In order to finish with the selection of the best possible move, giving to the worst possible move for the opponent, all possible moves stored in *iaAuxList4* are evaluated and the best one is saved. In the case that there are two possible best moves, the IA chooses the one with the higher final position or, in the case that the final position is also the same, choose the move with the lower initial position.

```
    score1 = -900000;
    for (int c = 0; c < iaAuxList4.Count; c = c + 5)
    {
       AuxChessmans[iaAuxList4[c + 2], iaAuxList4[c + 3]] = AuxChessmans[iaAuxList4[c + 0], iaAuxList4[c + 1]];
        AuxChessmans[iaAuxList4[c + 0], iaAuxList4[c + 1]] = null;
```

```
        score2 = BestPosibleScore(AuxChessmans, ForWhiteTeam);

        if (score2 > score1)
        {
           auxBestOption[0] = iaAuxList4[c];
           auxBestOption[1] = iaAuxList4[c + 1];
           auxBestOption[2] = iaAuxList4[c + 2];
           auxBestOption[3] = iaAuxList4[c + 3];
           score1 = score2;
        }

        if (score2 == score1)
        {
           iaAuxList = ValueChessmansPosition(board, auxBestOption[0], auxBestOption[1], auxBestOption[2], auxBestOption[3], ForWhiteTeam);
           iaAuxList2 = ValueChessmansPosition(board, iaAuxList4[c], iaAuxList4[c + 1], iaAuxList4[c + 2], iaAuxList4[c + 3], ForWhiteTeam);

           if (iaAuxList2[1] > iaAuxList[1])
           {
              auxBestOption[0] = iaAuxList4[c];
              auxBestOption[1] = iaAuxList4[c + 1];
              auxBestOption[2] = iaAuxList4[c + 2];
              auxBestOption[3] = iaAuxList4[c + 3];
              score1 = score2;
           }
           if (iaAuxList2[1] == iaAuxList[1] && iaAuxList2[0] < iaAuxList[0])
           {
              auxBestOption[0] = iaAuxList4[c];
              auxBestOption[1] = iaAuxList4[c + 1];
              auxBestOption[2] = iaAuxList4[c + 2];
              auxBestOption[3] = iaAuxList4[c + 3];
              score1 = score2;
           }
        }

        AuxChessmans[iaAuxList4[c + 0], iaAuxList4[c + 1]] = board[iaAuxList4[c + 0], iaAuxList4[c + 1]];
        AuxChessmans[iaAuxList4[c + 2], iaAuxList4[c + 3]] = board[iaAuxList4[c + 2], iaAuxList4[c + 3]];
     }
```

2.3.5. Final version of the game

In this third version of the project we find the integration of Virtual Reality (VR) in the game, to achieve this objective the camera and the inputs that the code receives must change.

On the other hand, to facilitate a more immersive experience a 3D decoration called Skybox is added. A Skybox is a wrapper around the scene that connects the 6 in-side walls of a cube between them leaving the connections almost invisible. An example is found in figure 17.

Figure 17. Skybox example

2.3.5.1. Project imports

In order to get the code for the VR viewer and the new inputs, it is necessary to download and import the OCULUS Integrates package, which is available for free in the Unity's Asset Store. This package includes all the necessary codes in order to connect any Oculus viewer to the project.

On the other hand, it has also been downloaded the G.E. TEAM Fantasy Skybox FREE package which is also available for free in the Asset Store. This package contains the textures of two possible Skyboxs.

2.3.5.2. VR Camera

One of the most significant changes in this final phase of the project is the change of camera. To obtain a 3 DoF freedom, the existing camera has to be removed and replaced by a structure called *VR Rig* (figure 18).

Figure 18. VR Rig structure

Inside the *VR Rig* we find a GameObject called *CameraOffset* that joins the *VR Camera* with the *VR Controller*. As you can see in this project only one controller is added because it doesn't require the second one to play.

There are also two codes added from the oculus package: *XR Rig* and *OVR Manager* (figure 19). The *OVR Manager* script is used to index the entire Oculus package and make possible the use of its functions within the code; in the XR Rig, we find:

- The Rig Base Game Object is used to indicate which game object acts as the transform from tracking space into world space. In the recommended hierarchy this is the "XR Rig" game object.

- The Camera Floor Offset Object is used to set which object will have a vertical offset applied if the device tracking origin does not contain the users height.
- The Camera Game Object field is used to indicate which game object holds the player's camera. This is important as the user's camera may not be at the origin of the tracking volume. In the suggested hierarchy this is the "Camera" game object.
- The Tracking Space field is used to set the desired tracking space used by the application
- The Camera Y Offset is the number of world space units that the Game Object specified by the Camera Floor Offset Object will be moved up vertically if the device tracking origin does not contain the user's height.

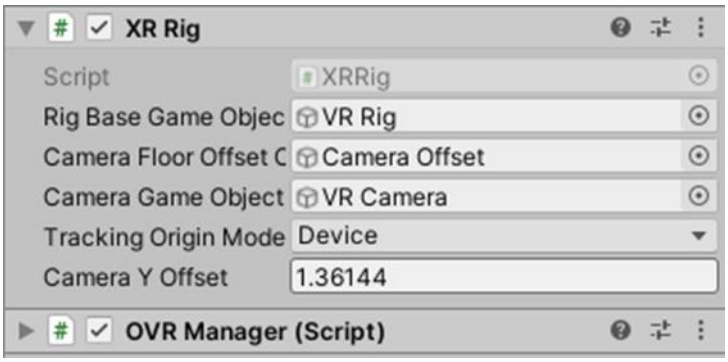

Figure 19. XR Rig and OVR Manager scripts

2.3.5.3. New global variables

In order to use the new command it is necessary to implement the following global variables:

Name and type of the variable	Definition of the variable
float HandRight;	This variable is used to obtain the input from the control Trigger. It is updated in each frame.
Vector2 touchpad;	This variable will receive input from the Touchpad of the controller. It is updated in each frame.

Table 3. Global variables added for the final version of the game

2.3.5.4. Upgrade of the Update function

The change in this function has been minor. We found that the global variable *HandRight* is updated in each frame waiting for an input from the *Trigger*; when the *Trigger* is pressed the function enters to the selection phase of the piece and when the *Trigger* is released tries to move it.

```
private void Update()
{
    BoardCursorPosition();
    LightCursorPosition();

    HandRight = OVRInput.Get(OVRInput.Axis1D.PrimaryHandTrigger);
```

```
    if (onePerTurn) [ … ]
    if (isWhiteTurn)
    {
      if (selectedChessman != null) [ … ]

      if (OVRInput.GetDown(OVRInput.Button.PrimaryIndexTrigger))
      {
        if (xSelection >= 0 && zSelection >= 0)
        {
          if (selectedChessman == null)
          {
            SelectPiece(xSelection, zSelection);
          }
        }

      }

      if (OVRInput.GetUp(OVRInput.Button.PrimaryIndexTrigger))
      {
        if (xSelection >= 0 && zSelection >= 0)
        {
          if (selectedChessman != null)
          {
            MovePiece(xSelection, zSelection);
          }
        }
      }
    }
  }
```

2.3.5.5. Upgrade of the BoardCursorPosition function

As we can see, the global variable *touchpad* is updated in each frame waiting for an input. After obtaining the data and transforming it to the values we need, we confirm that these values are inside the board and then the code updates the global variables *xSelection* and *zSelection*.

It transforms the values received by the controller because they go from -1 to 1 in the X and Y axis, while the code expects values between 0 and 7. Initially it would seem to be enough if it is added one and multiplied by 4, but the touchpad is a rounded area and has problems with the corners of the board. So the solution pass through apply a small offset multiplying the input by 1.25 which makes it easier to reach all the area of the board.

```
  private void BoardCursorPosition()
  {
    int auxx, auxz;

    touchpad = OVRInput.Get(OVRInput.RawAxis2D.RTouchpad);

    auxx = (int)(((touchpad[0] * 1.25) + 1) * 4);
```

```
    auxz = (int)(((touchpad[1] * 1.25) + 1) * 4);

    if (auxx >= 0 && auxx < 8 && auxz >=0 && auxz < 8)
    {
      xSelection = auxx;
      zSelection = auxz;
    }
  }
```

2.3.5.6. Upgrade the UpdateChessmanDrag function

The *UpdateChessmanDrag* function has similar changes to the *BoardCursorPosition* function taking into account that it has to change the position of a piece when it is selected. To achieve this objective the function uses these local variables:

- origindrag: this Vector3 type variable will be the position where the code will send the piece if it is selected.
- auxdragx, auxdragy: are two float type variables that allow an assignment to origindrag.

Even if the variables *auxdragx* and *auxdragy* seem to be an intermediate step that can be removed, the assignment of Vector3 does not allow a direct assignment through a type transformation, so it becomes totally necessary.

```
  private void UpdateChessmanDrag(Chessman c)
  {
    Vector3 origindrag = Vector3.zero;

    float auxdragx, auxdragz;

    touchpad = OVRInput.Get(OVRInput.RawAxis2D.RTouchpad);

    auxdragx = (float)(((touchpad[0] * 1.25) + 1) * 4);
    auxdragy = (float)(((touchpad[1] * 1.25) + 1) * 4);

    origindrag.x += auxdragx;
    origindrag.y += 1;
    origindrag.z += auxdragz;

    c.transform.position = origindrag;
  }
```

2.3.5.7. Lights that we find in the project

We found 4 lights that come out repeatedly throughout the project with the intention of improving the player's experience:

- The light that indicates all the possible movements of the selected piece (figure 20)
- The light that indicates that a king is being threatened (figure 21)

- Mouse / VR Controller pointer light (figure 22)
- The light that indicates the AI movement (figure 23)

All these lights are created through an effect integrated into Unity itself: Particle System. This effect allows having a loop animation of a *particle jet*, this animation is alterable through a series of modules that contains groups of properties. By modifying these properties you can achieve a big variety of effects, also it can be added a texture to give more body to the animation.

The codes that allow and change the position of the lights on the board are *BoardHighlights* class, *IALights* class, *KingLights class* and the *LightCursorPosition* function. The LightCursorPosition fuction has already been shown in the first version and the other three are based on the solution developed by Michael Doyon [N3K EN] and can be found in the Annex.

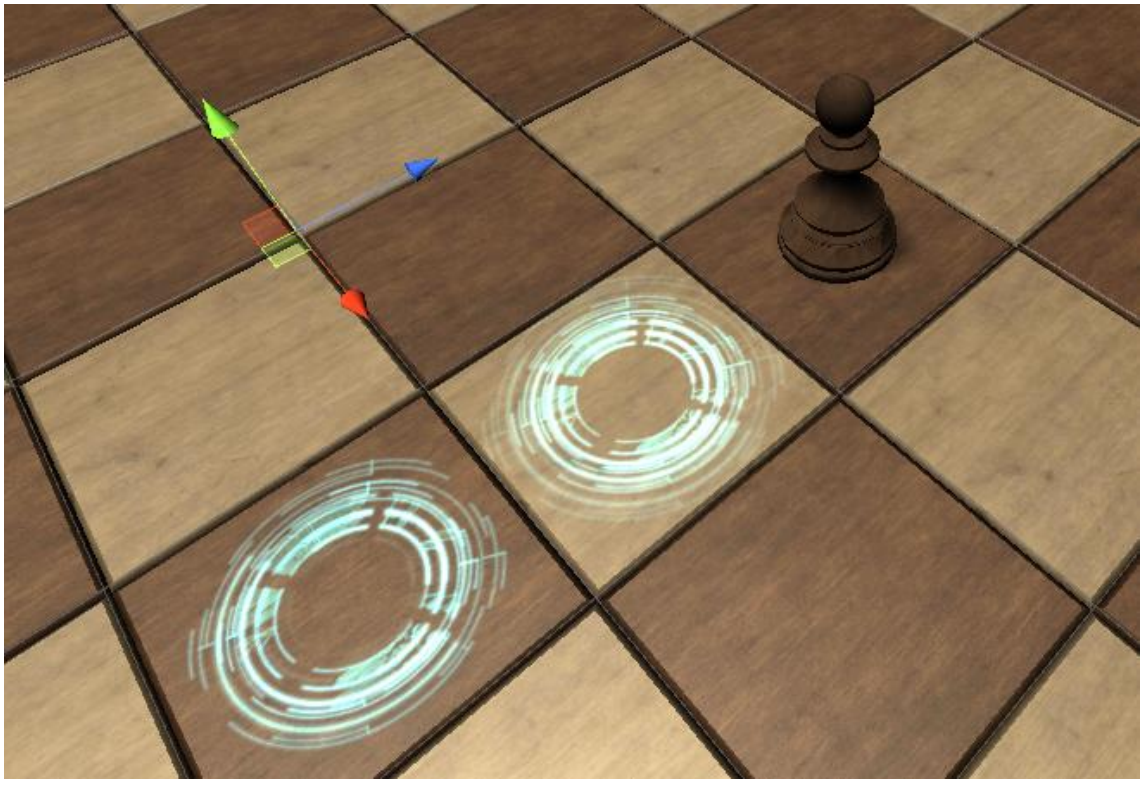

Figure 20. Light of possible movements

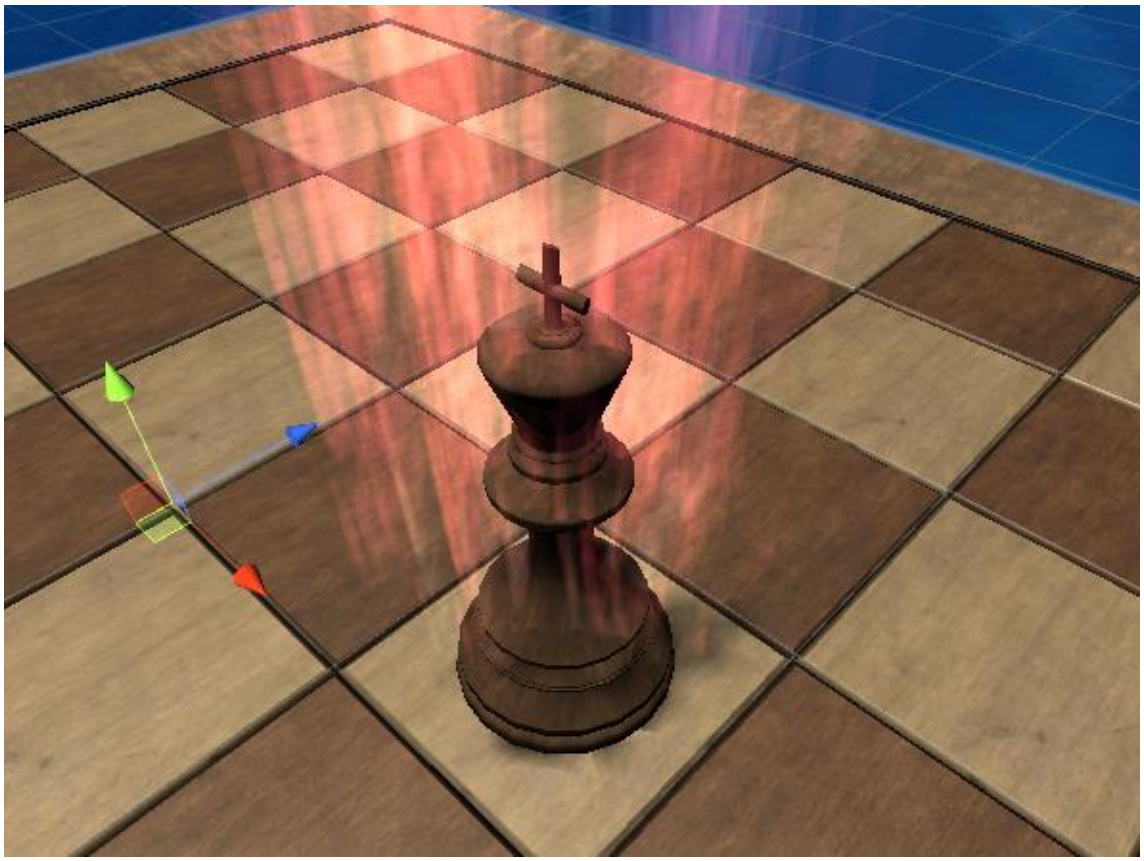

Figure 21. Ligth of be in check

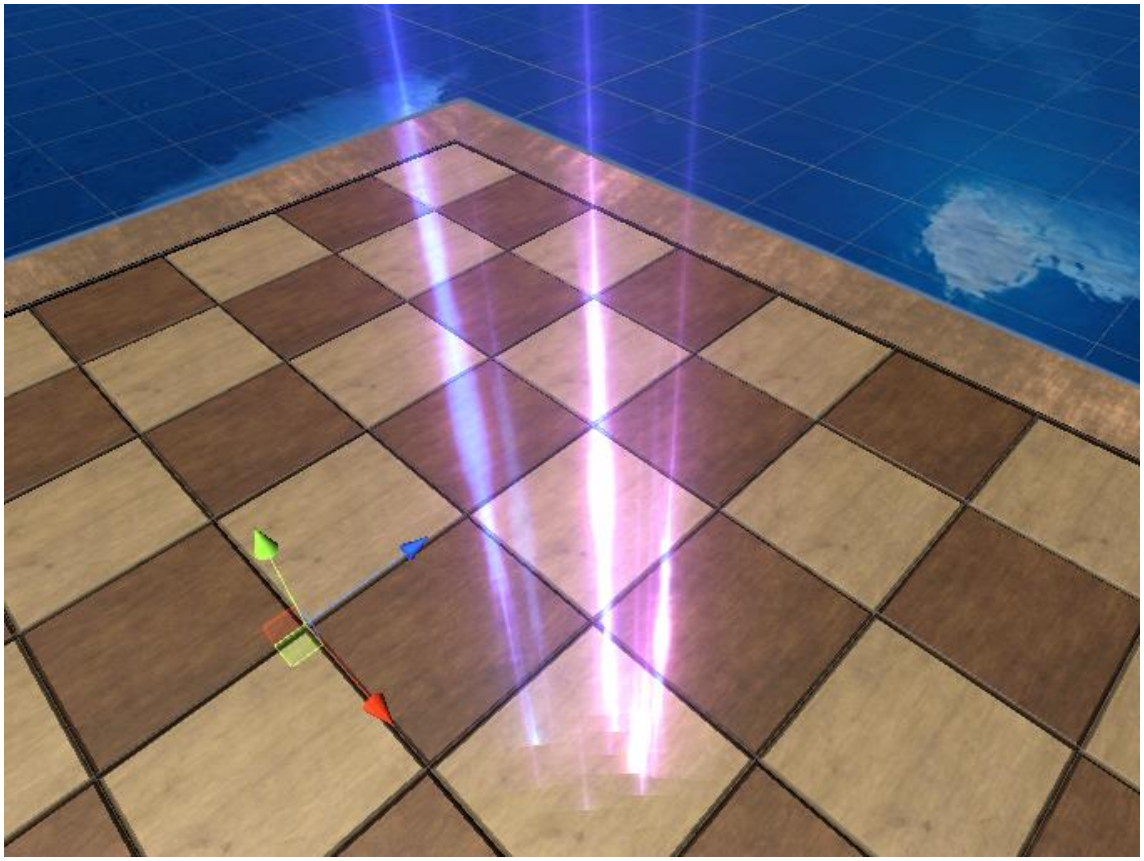

Figure 22. Mouse / VR Controller pointer light

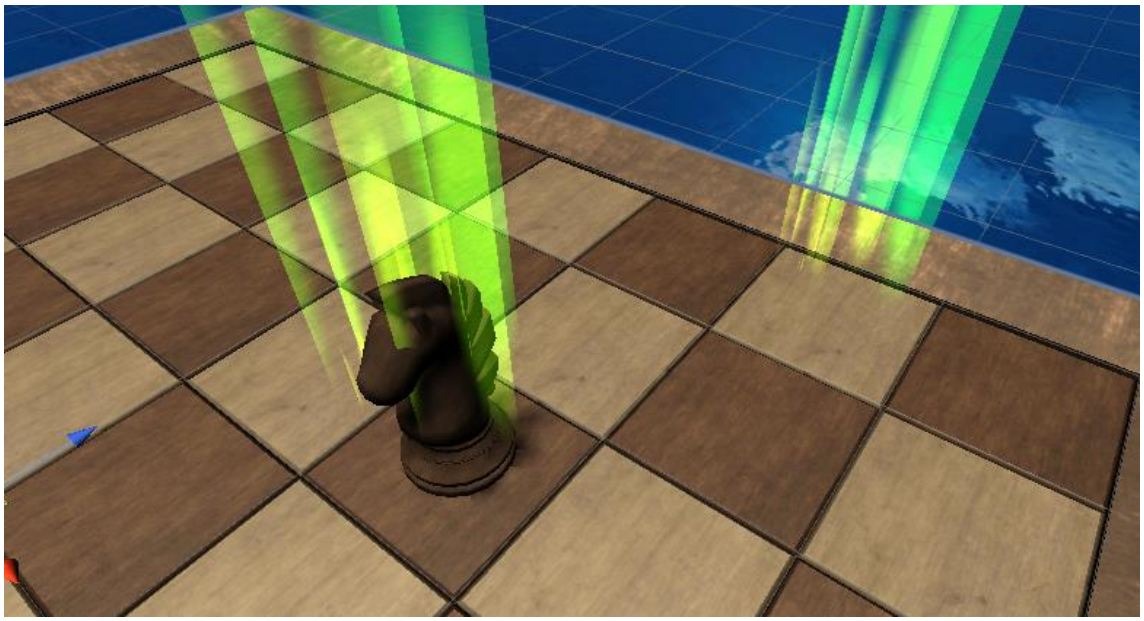

Figure 23. Light of the AI movement

2.3.6. Final view of the game

Once all the development has been completed, all that remains is to install the APK in the Oculus Go and enjoy the experience. Below you can see some pictures taken directly from the device.

Figure 24. Final view 1

Figure 25. Final view 2

Figure 26. Final view 3

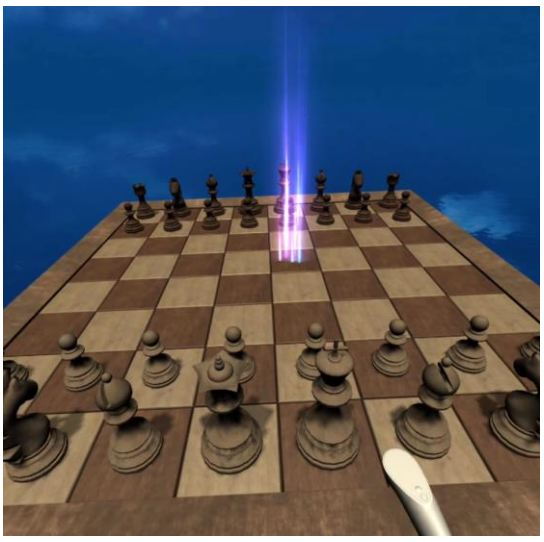

Figure 27. Final view 4

3. SUMMARY OF RESULTS

3.1. Budget

The budget for the creation of this project is presented in table 4 below. It includes both the price of the software and the device used for the tests; we also find the amount of hours invested in the modelling and texturing of the pieces, as well as in the programming of the code.

Budget table			
Software		Price	
Autodesk Maya (annual license)		2.136 €	
Substance Painter		0 €	
Unity (annual license)		150 €	
Hardware		Price	
Oculus Go		275 €	
Computer		1.100 €	
Hours invested			
Worker	Total hours	Price per hour	Total price
Graphic designer	40 h	18 €/h	720 €
Engineer	650 h	30 €/h	19.500 €
Total cost of the project		23.881 €	

Table 4. Budget Table

3.2. Gantt

GANTT DIAGRAM			
Activity	Start date	Duration (days)	Finish date
Research of the state of art	24-feb.	7	2-mar.
Search and installation of the necessary resources	2-mar.	4	6-mar.
Modeling the pieces and the board	6-mar.	7	13-mar.
Texturization of the chess pieces and the chessboard	13-mar.	7	20-mar.
First version of the game	13-mar.	35	17-abr.
Tests of the first version	17-abr.	2	19-abr.
Second version of the game	19-abr.	20	9-may.
Tests of the second version	9-may.	2	11-may.
Final version of the game	11-may.	30	10-jun.
Tests of the final version	10-jun.	5	15-jun.
Collection and review of all information	15-jun.	2	17-jun.
Memo writing	19-abr.	65	23-jun.

Start date	24-feb.
End date	23-jun.

Table 5. Gantt description table

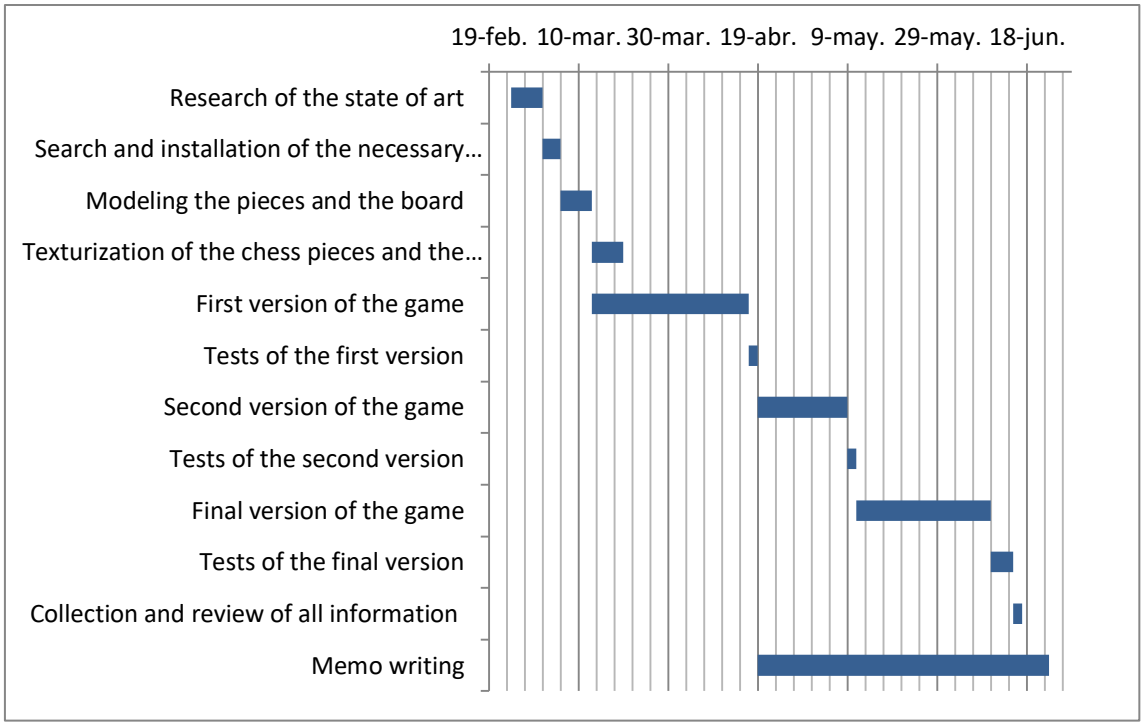

Figure 28. Gantt diagram

3.3. Problems found

At the beginning of the project I didn't have any experience with Unity or any other game engine, this caused a series of errors that I couldn't correct until I really learned how Unity works. Once I learned the basics, I was able to develop my own code without any problems.

On the other hand, during the development of this project we found ourselves in a situation that nobody could expect: the virus COVID-19. This unexpected situation has directly affected the development of this project.

The first problem I encountered from this situation was that at the beginning of the pandemic my computer broke down. As the repair shops were closed, I had to find the problem myself without having too much technical knowledge about it. Once I could fix it, I had already lost a lot of time and this affected the deadlines that I originally defined.

Another problem was that the COVID-19 affected my family financially, making it impossible for me to buy a VR Viewer until almost the end of the project itself.

When I had a device to do the tests, I discovered that the integration of the VR into a project designed for another platform was not as immediate as it might seem at first. Then I discovered the huge community of developers that both Unity and Oculus had who helped me achieve the initial goal of my project.

3.4. Conclusions

Initially I had the impression that virtual reality was not used by small developers and that it was used much more by companies than by individuals, I couldn't be more wrong.

Although big companies are bringing big projects to market, the community of small developers that exists under both Unity and Oculus is not only large but also very active. We find all sorts of forums, video tutorials, news and project displays that help you to solve the problems you encounter in your own development and in case you don't find a direct solution, they offer you tips that can help you to find your own solution.

Thanks to these two great communities, I have managed to achieve the main objective of this project: to complete the development of a chess video game in a virtual reality environment by carrying out each of the phases during the creation of the game.

I am aware that the final result is not entirely commercial, but it offers a very solid base to which can be added possible improvements. If there were the option to dedicate more time to the development of this application, I would achieve a fully marketable version without any doubt.

3.5. Possible future work

There are mainly seven upgradeable points to enrich the user experience when playing this game:

1- Difficulty of the artificial intelligence

The artificial intelligence implemented in this project only takes into account 2 iterations of the Minimax function: a white movement and a black movement. If the AI looks more turns forward, it would be much harder to beat.

On the other hand, this project uses fixed positioning matrices that do not offer all the information that could be useful to the AI. If the AI used matrices with variable values according to the number of turns played, the AI would be able to respond better as the game progresses.

If the difficulty of the AI was improved, levels of difficulty could also be included within the game itself, offering more difficulty to more experienced people and opening the market to more possible users.

2- Artificial intelligence animation

In this project the movement of the artificial intelligence is not seen, it makes an instantaneous transfer of pieces and positions some lights in the origin and in the end to clarify to the user what play has been made. If, in addition to these lights, some kind of animation were implemented for the movement of these pieces, it would give a much more real sensation to the player.

3- Improving the input of the controller

During the development of this project, I have encountered many problems when implementing virtual reality in the game, one of the most difficult to solve was the insertion of inputs by the controller. The solution chosen was to introduce the selection/movement of the piece by the touchpad, but a possibility that would have to be considered would be to perform this action through a Raycast pointer from the same controller.

4- Ambient music and sounds

Unity offers the possibility of introducing both ambient music and sounds to the objects when they perform some action. Ambient music and sounds were not in the initial scope, and the difficulties faced along the project did not allow me to develop it. Creating the sounds for both the ambient and the pieces would give more quality to the final result.

5- Victory or defeat screen

At the end of the game, it does not indicate who has won or offers any kind of message to the user; a possible improvement would be to include a screen for both defeat and victory.

6- Online multiplayer connection

Due to the difficulties of connecting the virtual reality devices, a single-player design was chosen, but it is not impossible to create multiplayer games by connecting two VR headset. If more time were available, the implementation of a multiplayer version would greatly enrich the game.

7- Main menu

To finalize and merge all the different improvement possibilities, develop and implement a menu will improve the user experience.

4. List of bibliographical references

Las 7 fases más importantes en el desarrollo de juegos | Escuela de Videojuegos | Hektor Profe. (n.d.). Retrieved June 28, 2020, from https://docs.hektorprofe.net/escueladevideojuegos/articulos/fases-del-desarrollo-de-videojuegos/

Oculus Quest & Go // Sideloading Made EASY with SideQuest (Windows, Mac and Linux) - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=HspVa4i9rPg

Unity o Unreal ¿Como elegir? | Avante Digital Institute. (n.d.). Retrieved June 28, 2020, from https://www.avantedigitalinstitute.es/2019/05/13/unity-o-unreal-por-donde-empiezo/

Cómo se hace un juego de realidad virtual. (n.d.). Retrieved June 28, 2020, from https://www.xataka.com/realidad-virtual-aumentada/como-se-hace-un-juego-de-realidad-virtual

Los motores gráficos más usados. (n.d.). Retrieved June 28, 2020, from https://www.elobservador.com.uy/nota/los-motores-graficos-mas-usados-2017819500

Top VR Game Engines -2019 Update. (n.d.). Retrieved June 28, 2020, from https://filmora.wondershare.com/virtual-reality/state-of-vr-games-the-game-engines-and-present-convention.html

Si quieres hacer tus propios juegos, estos son los mejores motores que vas a encontrar. (n.d.). Retrieved June 28, 2020, from https://www.vidaextra.com/listas/si-quieres-hacer-tus-propios-juegos-estos-son-los-mejores-motores-que-vas-a-encontrar

Los mejores programas de diseño 3D/modelado 3D | All3DP. (n.d.). Retrieved June 28, 2020, from https://all3dp.com/es/1/mejores-programas-diseno-3d-software-modelado-3d-gratis/

10 engines y motores para crear tus videojuegos | Escuela de Videojuegos | Hektor Profe. (n.d.). Retrieved June 28, 2020, from https://docs.hektorprofe.net/escueladevideojuegos/articulos/engines-motores-recopilacion-programas/

Unity - Manual: System requirements for Unity 2019.4. (n.d.). Retrieved June 28, 2020, from https://docs.unity3d.com/Manual/system-requirements.html

Degrees of Freedom (DoF): 3-DoF vs 6-DoF for VR Headset Selection. (n.d.). Retrieved June 28, 2020, from https://virtualspeech.com/blog/degrees-of-freedom-vr

Como exportar animaciones de Maya a Unity - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=pOsKIwMpr10

Exportar modelo de maya a unity - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=WzTi78q4bsM

COMO EXPORTAR de MAYA a UNITY/ How to export Maya to Unity - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=7kAm9ar9O6M

UV mapping pokemon en autodesk maya 2019 - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=4iUslz1Ygjk&t=80s

TEXTURIZAR UN POKEMÓN EN SUBSTANCE PAINTER y MAYA 2019 - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=c4dn6jdpXFg

Curso basico programacion C# - 2.Variables,errores y breakpoints - Visual Studio 2017 - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=MlrUFwHwrUA&list=PLLJJqiFt6VPoUO_mM5y6VSwGX-B7v__tu&index=2

How to get started with Unity3D - For Beginners - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=XDAYS-qYe6Y

How to Make a Chess Game with Unity | raywenderlich.com. (n.d.). Retrieved June 28, 2020, from https://www.raywenderlich.com/5441-how-to-make-a-chess-game-with-unity

Introduction to Unity Scripting | raywenderlich.com. (n.d.). Retrieved June 28, 2020, from https://www.raywenderlich.com/980-introduction-to-unity-scripting

Chess Game Tutorial • 1/5 • [Tutorial][C#] - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=CzImJk7ZesI

Chess Game Tutorial • 2/5 • [Tutorial][C#] - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=on_J2IFoPME&t=847s

Chess Game Tutorial • 3/5 • [Tutorial][C#] - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=sN_3fMo-dZg&t=590s

Chess Game Tutorial • 4/5 • [Tutorial][C#] - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=DP6I1owXI64&t=1s

Chess Game Tutorial • 5/5 • [Tutorial][C#] - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=q0v53MNLPcM&t=792s

Constructores de instancias: Guía de programación de C# | Microsoft Docs. (n.d.). Retrieved June 28, 2020, from https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/instance-constructors

Curso programación C# - 72. Inteligencia artificial (1)- Visual Studio 2017 - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=fa0K8HcaDCk&list=PLLJJqiFt6VPruMw8E-O37V9bL7STOvC9D&index=1

Coding Challenge 154: Tic Tac Toe AI with Minimax Algorithm - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=trKjYdBASyQ

Let's Create a Chess AI That Can Beat You (probably): Part 1 - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=RbV3R6JrQWo

AI Chess Project. (n.d.). Retrieved June 28, 2020, from https://www.cs.cornell.edu/boom/2004sp/ProjectArch/Chess/chessmain.html

Building a Simple Chess AI – Brandon Yanofsky. (n.d.). Retrieved June 28, 2020, from https://byanofsky.com/2017/07/06/building-a-simple-chess-ai/

Test Driven Chess Artificial Intelligence - CodeProject. (n.d.). Retrieved June 28, 2020, from https://www.codeproject.com/Articles/1168892/Test-Driven-Chess-Artificial-Intelligence

A step-by-step guide to building a simple chess AI. (n.d.). Retrieved June 28, 2020, from https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/

GitHub - lhartikk/simple-chess-ai: A simple chess AI. (n.d.). Retrieved June 28, 2020, from https://github.com/lhartikk/simple-chess-ai

Unity - Manual: Skybox. (n.d.). Retrieved June 28, 2020, from https://docs.unity3d.com/es/530/Manual/class-Skybox.html

Multiplayer Checkers Tutorial #8 - Lobby / Menu - Unity 3D[Tutorial][C#] - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=YyEHu3CpSTM

Multiplayer Checkers Tutorial #11 - Rail, Alert - Unity 3D[Tutorial][C#] - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=yuBB0222ZoY

How To Install Unity Game Engine (Getting Started) - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=KMuMhA6Lk0I

How To Create Your First Simple VR Project In Unity! - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=YpY3cLtOyBk

How to make a VR game in Unity - Part 1 - Setup, Hand presence, Grabbing object - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=sKQOlqNe_WY

Unity VR Tutorial: How to Build a Robin Hood VR Game From Scratch - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=Dh7Wwqs-s2c

Cómo crear e instalar aplicaciones de Unity en Oculus Go - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=8XhAEzBC-3o

Oculus - Unity. (n.d.). Retrieved June 28, 2020, from https://unity3d.com/es/partners/oculus

Oculus - Unity Manual. (n.d.). Retrieved June 28, 2020, from https://docs.unity3d.com/es/2018.4/Manual/VRDevices-Oculus.html

Object Reference Not Set to an Instance of an Object: How To Solve It. (n.d.). Retrieved June 28, 2020, from https://stackify.com/nullreferenceexception-object-reference-not-set/

How to fix adb devices shows unauthorized device. (n.d.). Retrieved June 28, 2020, from https://support.honeywellaidc.com/s/article/How-to-fix-adb-devices-shows-unauthorized-device

GUIDE: ADB Devices not displaying- FIXED : OculusGo. (n.d.). Retrieved June 28, 2020, from https://www.reddit.com/r/OculusGo/comments/8ucrr9/guide_adb_devices_not_displaying_fixed/

I'm on PC and there is no "allow usb debugging" option thing showing up. : OculusQuest. (n.d.). Retrieved June 28, 2020, from https://www.reddit.com/r/OculusQuest/comments/bu4946/im_on_pc_and_there_is_no_allow_usb_debugging/

Quest won't connect to PC on developer mode since last update — Oculus. (n.d.). Retrieved June 28, 2020, from https://forums.oculusvr.com/community/discussion/81191/quest-wont-connect-to-pc-on-developer-mode-since-last-update

Update: Found the Fix and possible reasons Oculus Go's will not connect to PCs for file transfer — Oculus. (n.d.). Retrieved June 28, 2020, from https://forums.oculusvr.com/community/discussion/64628/update-found-the-fix-and-possible-reasons-oculus-gos-will-not-connect-to-pcs-for-file-transfer

Anyone have trouble with Oculus Go disappearing from "adb devices"? — Oculus. (n.d.). Retrieved June 28, 2020, from https://forums.oculusvr.com/developer/discussion/64603/anyone-have-trouble-with-oculus-go-disappearing-from-adb-devices

Oculus Go not showing up in adb :(- Unity Forum. (n.d.). Retrieved June 28, 2020, from https://forum.unity.com/threads/oculus-go-not-showing-up-in-adb.530466/

OculusGo Kiosk Mode. (n.d.). Retrieved June 28, 2020, from https://oculusgokioskmode.tweaklab.org/

Oculus Go Unity Setup (Quick Start) — Scriptable. (n.d.). Retrieved June 28, 2020, from https://scriptable.com/blog/oculus-go-unity-setup-quick-start

Installation Instructions for Oculus Go (Windows 10). (n.d.). Retrieved from https://developer.oculus.com/documentation/mobilesdk/latest/concepts/mobile-device-

Oculus ADB Drivers | Developer Center | Oculus. (n.d.). Retrieved June 28, 2020, from https://developer.oculus.com/downloads/package/oculus-adb-drivers/

Locomotion | XR Interaction Toolkit | 0.0.6-preview. (n.d.). Retrieved June 28, 2020, from https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@0.0/manual/locomotion.html

Easy Controller Selection | Oculus. (n.d.). Retrieved June 28, 2020, from https://developer.oculus.com/blog/easy-controller-selection/

Add ray cast from the hand in Unity — Oculus. (n.d.). Retrieved June 28, 2020, from https://forums.oculusvr.com/developer/discussion/63970/add-ray-cast-from-the-hand-in-unity

Unity's UI System in VR | Oculus. (n.d.). Retrieved June 28, 2020, from https://developer.oculus.com/blog/unitys-ui-system-in-vr/

Unity - Manual: Rays from the Camera. (n.d.). Retrieved June 28, 2020, from https://docs.unity3d.com/Manual/CameraRays.html

Unity - Scripting API: Physics.Raycast. (n.d.). Retrieved June 28, 2020, from https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

Unity 5.2 VR Raycast? - solved - Unity Forum. (n.d.). Retrieved June 28, 2020, from https://forum.unity.com/threads/unity-5-2-vr-raycast-solved.361211/

Unity - Scripting API: Physics.Raycast. (n.d.). Retrieved June 28, 2020, from https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

[TIP] Detecting where the player looks at — Oculus. (n.d.). Retrieved June 28, 2020, from https://forums.oculusvr.com/developer/discussion/4198/tip-detecting-where-the-player-looks-at

Introduction to VR in Unity - PART 3 : TELEPORTATION - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=fZXKGJYri1Y&t=358s

Unity VR Tutorial - Basics of Oculus OVR Tracking - WITHOUT PREFABS - - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=sNgAX0hws7Y

Oculus Quest Unity Tutorial - using raycasts to create hand pointers - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=F60UIo7Y1YY

Unity VR: Oculus Touch Input Sample - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=ozTDt0SPKjg&t=226s

How to Pass Variables Between Scripts in C# - YouTube. (n.d.). Retrieved June 28, 2020, from https://www.youtube.com/watch?v=ck6OyNBC95c

VR Headsets Are Dying A Lonely Death. (n.d.). Retrieved June 28, 2020, from https://www.forbes.com/sites/barrycollins/2020/05/04/vr-headsets-are-dying-a-lonely-death/#643660ea4d95

Stop Saying Virtual Reality Is Dying. (n.d.). Retrieved June 28, 2020, from https://www.forbes.com/sites/joeparlock/2020/05/05/stop-saying-virtual-reality-is-dying/#817989e646e1