Quan Dao

Multiplayer Game Development with Unity and Photon PUN
A Case Study: Magic Maze

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

18 October 2021

Abstract

Author:	Quan Dao
Title:	Multiplayer game development with Unity and Photon PUN A Case Study: Magic Maze
Number of Pages:	40 pages + 0 appendices
Date:	18 October 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Mobile Solutions
Supervisors:	Toni Spännäri

This study demonstrates the development process of a multiplayer game using the Unity game engine with the Photon PUN plugin. The goal of the project was to develop a multiplayer game with sole focus on the technicalities as a fun and realistic way of teaching and learning. The case of the project was digitizing Magic Maze. The reasons behind the choice of developing a digital version of a board game were to avoid all details for game rules and balancing, and instead concentrate on working on a multiplayer game.

The project was carried out by using an iterative approach due to continuous learning and improvement on the understanding of the PUN plugin and multiplayer game development concepts.

The study aims at enthusiastic game developers who are seeking initial guidelines on developing a multiplayer game with Unity engine.

Keywords: Game development, Unity, Photon PUN, C#, 3D games

Contents

List of Abbreviations

UI:	User-interface
PUN:	Photon Unit Networking
API:	Application Programming Interface
OS:	Operation System
OOP:	Object-Oriented Programming
FOV:	Field of View
UNet:	Unity Networking
TCP:	Transmission Control Protocol
UDP:	User Datagram Protocol
rUDP:	Reliable User Datagram Protocol
FPS:	First-person games

1 Introduction

This project covers the development process of digitizing a multiplayer board game using Unity game engine and Photon PUN. Even though the game is originally built for Windows OS, there are plans to expand the platform coverage of the game to port to as many platforms as possible, from mobile platforms, like Android and iOS, to consoles, such as Nintendo Switch.

Board games have existed for decades, entertaining people from all ages, colours, and genders. Not only are they used at home or bars, but they are also endorsed by companies for team activities. Co-op board games can be seen in every company's game night. The main reason for this is in addition to the relaxing times they can easily provide, they also are the perfect tool for improving teamwork and communications between colleagues. However, since board games require every player to all gather at the same place and at the same time, it can prevent games from being played at all. Therefore, digitizing a board game provides opportunity to everyone to be able to enjoy the games with their friends or colleagues.

Among various board games available on the market, Magic Maze stands out thanks to its unique silent cooperation mechanic. In addition, it features a different multiplayer mechanic comparing to most games since each players have the option to control every characters. Often times, players can only control their own character, but this is not the case in Magic Maze. This alone creates interesting challenges when it comes to making a digital version due to the lack of tutorials comparing to traditional multiplayer games. Therefore, it is chosen as the subject for this study.

The report intent is to provide a starting guide for people who already have some experience with the Unity game engine and looking to expand their knowledge by building a multiplayer game with Photon PUN. The goal is to walk through the process of digitizing the Magic Maze board game and provide tips

or guidance on solving potential challenges. Following the Introduction, Chapter 2 introduces the core concepts and components of the Unity game engine and Photon PUN multiplayer plugin. Chapter 3 covers the game designs aspect of digitizing an existing board game with predefined rules. In Chapter 4, the actual development process of the game is covered in detail. Finally, Chapter 5 focuses on the possible improvements provided during the playtesting process. The project's intention is to gain the knowledge of developing a multiplayer game, and not in any way to generate incomes.

2 Tabletop Games Design

Even though it can seem unbelievable, but board games may have existed for thousands of years. A series of carved stones were found by archaeologists in a burial site in southeast Turkey. They were sculpted in various shapes, such as pigs, dogs. There were also tokens that represent dice and other game-like tokens [1].

Many of the knowledge and expertise humanity nowadays have been the result of a long and tiring process of trial and error. It began as soon as humans populated earth, and has been going on since then, sometimes even without any concept of it. Similarly, knowledge on game design and development has developed and passed through generations. Though having been in existence for an enormously long time, the definition of game design is so broad that it is challenging for games to guarantee their success, even ones that come from well-known game designers. At its simplest form, it is the act of deciding what a game should be about. However, it often includes making thousands of decisions. Unlike chemistry, where there is a periodic table for the known elements, no such thing exists in the game design world. Even though there can be talks about best practices, they cannot be used as solutions to every single design question [2].

As more and more games succeed, they act as study subjects for game designers around the world. Such knowledge, accumulating over the years,

together they form a concept called game literacy. Many of the design and decision on what makes a good game is based on passed exposure to either tabletop or video games. For trading or bidding games, inspiration can be made from Monopoly, as it is often considered as one of the brands of games. For strategy, Chess is one of the best and most known tabletop game in the world. It is recommended for any game designer to learn from it to construct their game literacy [3, p. 3]. For example, during the making of the gun upgrade system for “The Last of Us”, Alexandria Neonakis, a UI designer at Naughty Dog, iterated through various options. While maintaining the main idea of it being accessible directly from the weapon swapping menu, there are various styles of user-interface that she tested. Based on her own game literacy, the standalone upgrade system tends to be forgotten easily by players, and only accessed when it is absolutely needed. However, no matter what version she chose for the integrated upgrade system, players tend to perform upgrade as soon as possible, without much planning and thoughts into the best option for their playstyle since it is always available. In addition, drawing from her experience, people who complain about the user-interface, often is not about how it looks, but rather the clutter it causes on the screen.

As a result, Alexandria Neonakis decided to move it completely out of the weapon swapping radial UI. Instead, players would have access to an upgrade table in carefully placed locations. Not only does this allow the designers to control where the most logical position for the table would be, but also encourages players to spend upgrade points on what they actually want or need since it would be a while until the next one [4].

Tabletop games, and even video games, have various core elements that contribute to an engaging entertainment tool. First of all, even though it might not be explicitly defined in the rule, most, if not all, games have states in some form. A game state is usually a snapshot of the situation in the game at an exact moment, including all elements, such as characters position, current score, enemies count. It contains all the necessary information to recreate the game at that point. For example, a particular position in chess is a game state

of its own. Secondly, to allow players to alter the game state, mechanics are available. They are the set of actions or processes that can be used to update the game. A perfect example in chess is moving a piece to a new square. Such movement has created an entirely new game state where new moves appear or get blocked. When mechanics exist that do not require player control, they are system mechanics. Often times, they are due to the physical limitations or the constantly moving clock in the real world. Combining the components above in different ways that reinforce or encourage players to play the way the game is intended is called dynamics [3].

3 Technologies Used

To develop any game with the purpose to learn the process and challenges one might come across, the choice of game engine is one of the most important decisions to make. Without extensive experience with the subject, any help from the seniors or the community is much needed. Therefore, the Unity game engine is the suitable choice for students or indies thanks to the massive number of active users.

On top of the Unity game engine, the fastest way to develop a multiplayer game would be with the help of a prebuilt tool. One of the most powerful, and the most popular plugin for such need is called Photon PUN. It contains ready-made features for sending and receiving messages between clients, matchmaking, friend list, etc.

3.1 Unity Game Engine

3.1.1 History

Unity, developed by Unity Technologies, is a cross-platform 3D game engine initially launched on June 6, 2005. It is built using C++, but supports game development with C#, UnityScript (a JavaScript-like scripting language), and Boo. Starting from the year of 2014, Unity was gradually removing supports for

UnityScript and Boo due to inferior performance, poor documentation, and lack of usage outside of the Unity game engine community. With the 2018.2 version of the engine released, support for both UnityScript and Boo was officially removed. Even though existing games built using mentioned programming languages can still be compiled, documentation and built-in options to create new scripts in such languages is discontinued [3, p. xix].

Thanks to being built with intention to be a cross-platform engine, the Unity game engine contains full support for porting a game to all types of platforms, from mobile platforms, like Android and iOS, to desktops, and to all consoles, such as PlayStation, Xbox, Nintendo Switch. However, due to the optimization limit nature of the only language available to users, being C# instead of C++, the engine is mainly used for developing mobile games and small to medium-sized console games [3, p. xix].

One of the most beloved “features” of the Unity game engine is the pricing and financial eligibility of the available plans.

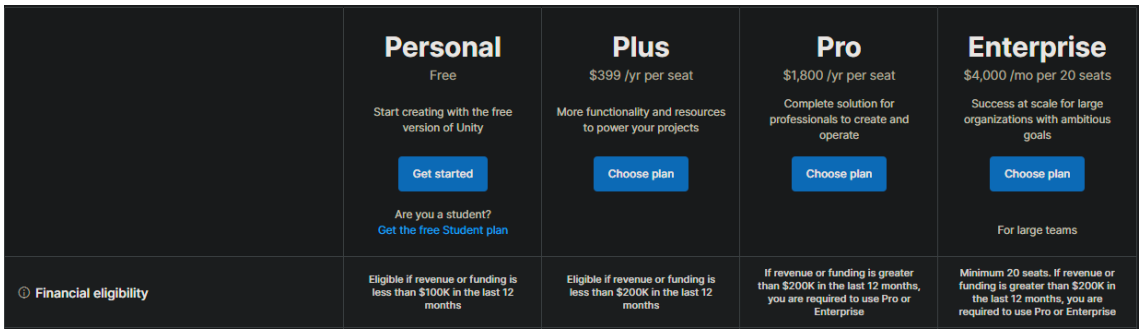


Figure 1. Available pricing plans for the Unity game engine

As demonstrated in Figure 1, since many of the game engine users are students, indies, or small studios, a free plan until generating over $100K in revenues a year is one of the best options available while not hindering the development process of the engine. As the game studios grow, gradually increasing price plan is available to meet the grow rate of the organizations [4, p. 1].

3.1.2 Concepts and Terminologies

Even though in Unity, there are various terminologies used only within the engine, the same concepts can be found in other game engines available in the market. This is primarily because within the Object-Oriented Programming (OOP) world, one of the best ways to build a scalable application is through composition instead of inheritance.

The most basic and founding concept of the engine is that every single object in a scene is called a Game Object. Essentially, a Game Object in the Unity game engine is simply a “thing”. At the core, it provides little to no functionalities without the additional components.

Components are the most important constructing elements to building games or applications with Unity engine. For example, looking at any books, it consists of the book cover, and the pages. They act as the components creating an object called a book.

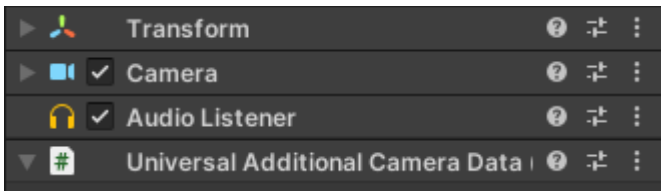


Figure 2. Example of the component list of a camera game object

As illustrated in Figure 2, all the components, in combination, fills the requirement of a camera object. The Transform component contains all the information about the game object’s three-dimensional position, three-dimensional rotation, and scale. Every object in the game should, at minimum, includes the Transform component. The next important component, and the founding component for a camera game object, is the Camera component. It holds all the necessary information for capturing frames, such as projection, and field of view (FOV). The other two components, Audio Listener and Universal Additional Camera Data, are supportive components that every camera game object can have.

The tool utilized for viewing the game object components in Figure 2 is called the Inspector window.

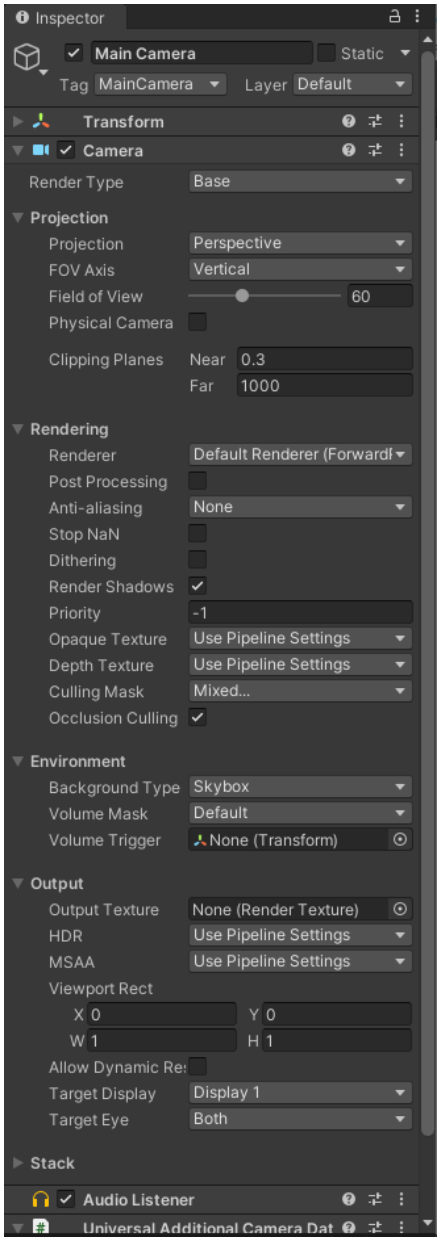


Figure 3. The Inspector window with the Camera selected

In addition to being able to view the game object's components, the inspector displays information for any objects selected, such as game assets, like a sprite or a sound file. However, the most important feature of the Inspector window, is the ability to edit the settings for the selected object, like names or components' settings. All of the info shown on the Camera component above are easily

configurable with only a couple clicks. Not only does this work with built-in components provided by Unity such as the Camera or Transform components, but also self-developed components. This allows for much simpler and user-friendly ways to update game settings in just a few clicks and without the need of a code editor, significantly reducing the game-balance duration.

Since the Unity game engine, by default, supports three-dimensional game development, being able to view every object from all angles would help tremendously in the development process. Built directly into the engine is a window with the only purpose to let the users traverse the entire game easily from any angles, called Scene, as shown in Figure 4.

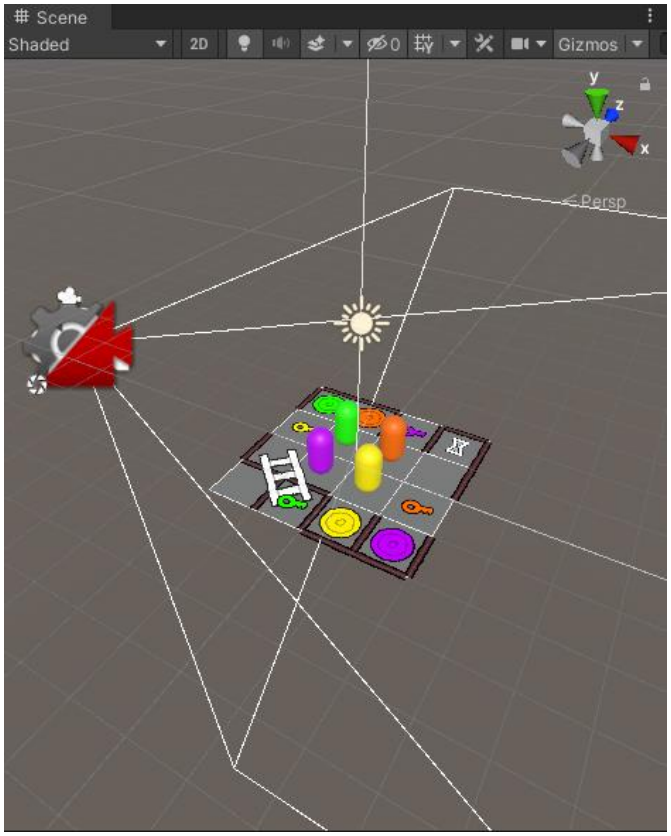


Figure 4. Example view of the Scene window

In addition to the mentioned features, the window also displays a grid, allowing easy placement and alignment of objects' positions.

It is impossible to work on any applications without being able to preview the outcome of the work being done. Available to all users, the Game window, demonstrated in Figure 5, acts as a sample of how the game would look for actual players.

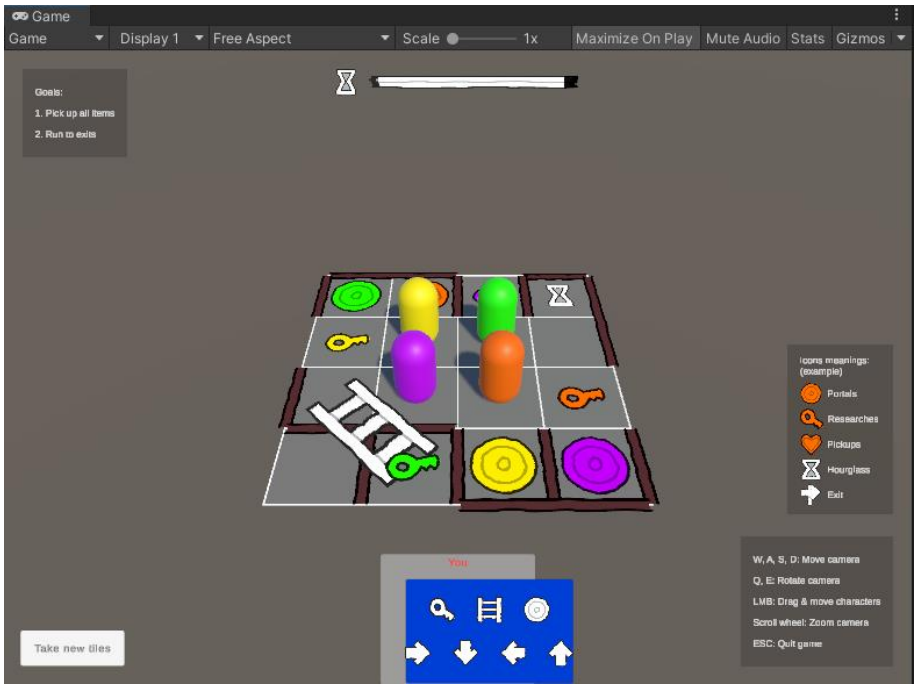


Figure 5. Example view of the Game window

Due to the complexity of developing a game, various options for playtesting are required, such as Mute Audio, or Stats to show the game's performance.

3.2 Photon PUN

Photon Unity Networking (PUN), developed by Exit Games, is a Unity plugin for building multiplayer games. Originally, when it was first released, it's primary goal was to provide an Application Programming Interface (API) that is consistent with the existing networking solution built by the Unity team, while solving some of its issues. Fortunately, ever since the deprecation of the Unity Networking (UNet) due to failures in meeting the needs of the multiplayer games' creators, PUN has quickly risen through the ranks to take the place as one of the most powerful tools for building multiplayer games [4, p. 75]. It has

been used in many of the successful games, like Prison Architect, PixelGun3D, GolfClash, and The Forest [5, p. 1].

3.2.1 Networking Terminologies

The fundamental of any networking solutions is the communication protocols. In general, there are two types of protocol, the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). TCP is built to be as reliable as possible. Before any TCP connection can begin, a “handshake” message chain is required to ensure that the communication between clients is reliable [5]. One of the main advantages of the protocol is that the received order of the messages is guaranteed to be exactly as they were sent. Therefore, in the scenario where some data is lost in transmission, the stream would be interrupted so that the lost message can be re-transmitted. Consequentially, the latency is significantly increased [4, p. 26].

On the other hand, UDP is the target protocol when speed is highly preferred, i.e., multiplayer games. However, such speed comes at a cost of reliability. In UDP, messages are not guaranteed to arrive, nor are they guaranteed to arrive only once. Therefore, a per-message reliability layer is built on top of UDP to allow developers to have flexibility on speed and reliability. This technique is known as rUDP [4, p. 26].

To utilize the protocols, applications either make use of APIs to communicate with the server or a direct socket into the server. An API is a command that allows communication between devices over the network without any user interaction. It is always a one-time action. Once the communication is complete, meaning the command’s execution is done, there is no ongoing connection between those devices. As a result, even though there might be thousands of devices communicating with the server at the same time, none of them know about the others [7, p. 1].

On the other hand, a direct socket connection is, in a way, having the devices plug a cable directly to the server. Therefore, similar to a LAN network, all connected clients can easily communicate with each other as though they are under the same network. In order to build a real-time multiplayer game with constant actions and communications between clients, this is the only option suitable for the purpose and requirements [8, p. 5].

PUN utilizes rUDP and direct socket to a cloud server to allow interaction between players with maximize speed while keeping reliability in-check [4, p. 75].

3.2.2 PUN vs UNet

The key difference between PUN and UNet is in the hosting machine. In UNet, the players are the ones creating and maintaining the room. One player, known as the host, would be the one to create the room, which lives on the local machine. The host would then have to do workarounds to public their self-hosted server. The game session can then start and continue to operate normally. That is, until the host disconnects. Since the server is technically the host's machine, as soon as the host leaves the session, either voluntarily or forcefully, e.g., due to network issues, the game is ended instantly. Even though the self-hosted server structure has its own benefits, such as low latency, the mentioned issue can be devastating [4, p. 75].

Another available structure for multiplayer game is the cloud server structure, which PUN employs. PUN works with another service, also built by Exit Games, called Photon Cloud. It is simply a cluster of servers with sole purpose to host rooms. Anytime players require a game session, they can request a room from the Photon Cloud. The room, publicly available to every player unless otherwise configured, is then constructed, and resided on the server cluster. Therefore, there is no host in a room, but rather a "master client", which by default is the player who requested the room. Although this comes with the disadvantage of requiring all players to have internet connection even though under the same

LAN network, when the master client leaves the room, the game can continue seamlessly [4, p. 76].

3.3 Integrated development environment

When it comes to C# application development, there are various integrated development environment (IDE) that programmers use. Since each has their own advantages and disadvantages, the decision on which is the suitable one to use often depend on the team or personal preferences.

For more conventional type of application built with C#, most teams prefer to use a famous IDE from Microsoft, called Visual Studio. It is a productive, modern, and innovative tool [9]. It contains many features, such as AI-powered code completions, hot-reload, and automatic continuous integration and continuous deployment (CI/CD) to Azure, a cloud platform maintained by Microsoft [10]. When it comes to Unity support, the only “meaningful” feature that is available is the auto complete for lifecycle methods [11].

However, for Unity projects specifically, many teams and studios prefer the new tool provided by the JetBrains team called Rider. In addition to the features that Visual Studio has, Rider has some of the best integration with the Unity game engine. In 2019, it even got recommended by the Unity team as a “fast C# scripting” tool instead of the more popular Visual Studio [12]. Not only do Rider has the lifecycle auto complete that Visual Studio does, in includes the ability to run tests, highlights on code lines with potentially mediocre performance, usage search in Unity files, and even support for shader language [13].

4 Game Designs of Digitizing Magic Maze

4.1 Magic Maze the Board Game

Magic Maze is a game that is set in a mall, where all the players are controlling four thieves trying to steal the four magical items hidden somewhere within. The

game starts out with every character on the same tile. As the thieves move and the game progress, the players open up the map by placing down additional tiles in search for the items and exits. Even though there are various pre-set scenarios with increasing difficulty, the object of the game is always to steal all magical items and exit the mall. The only loosing condition of the game is if the time runs out. There is an hourglass in the game, that players can potentially turn over to gain more time when any characters are on an hourglass grid cell.

Figure 6. Example game of Magic Maze

Different from common co-op board games, Magic Maze allows each individual players in a group of two to eight to move all four characters. However, to prevent a single player from doing everything and hinder the co-op aspect, each player receives a card that limits the movement direction or action that they can perform, as illustrated in Figure 6 [14]. On some movement cards, not only the direction is shown, but other symbols are displayed, signalling the players that they have additional power. There are escalators on some tiles, which only the player with the escalator icon on their movement card can use. Another skill that

allows players to skip certain parts of the game map is the ability to teleport characters from anywhere on the map to the same-coloured portal symbol. However, players can only make use of the portal before the phase two of the game, which happens when all four magical items are stolen. Quick traversing skills are only useful if the map is big enough to require such actions. To expand the game map and search for the necessary icons, when the characters are placed on cells with a magnifying glass sign, only the player with a similar magnifying glass icon on their movement card can take a new tile, place it down and open up the map. An important note to this is that only characters with the same colour as the magnifying glass can be used to explore more parts of the map.

Finally, the rule that makes Magic Maze a unique board game experience is that during gameplay, unless a specific rule allows, the game must be played in silence. The players, when no voiceful communication is permitted, need to cooperate in harmony, understand one another without talking. Fortunately, since everyone in the game shares a common goal, and every character can be seen and controlled by every player, less discussing is required. In simple terms, the game is designed carefully to not force constant communication between players to be able to work together. However, if at any time any players feel the need to remind another participant that there are actions available for them to take, a hammer-like tool can be used to tap on the table.

4.2 Magic Maze Implementation

4.2.1 Grid implementation

With only a quick glance at the game, one can recognize that the game functions on a grid-based logic instead of a more freedom style that many games follow. Most, if not all, of the game mechanics interact directly with the grid and its cells. For example, the characters only move from cells to cells, items are placed on cells, new tiles placement is cell dependent, and both portals and ladders take characters from one cell to another. Therefore, a solid

and scalable grid logic implementation plays a key role in the ease of future development.

Different from various games with grid-based logic, where only a single grid is needed for entire game since it can expand forever, such approach comes with many downsides if utilized for Magic Maze. First of all, as the playing area expands, there are no finite starting point for the entire grid since new tiles can be placed in a place where the cell position would be lower than zero, which shouldn’t be possible. One can potentially change the starting point as new tiles spawn in, but such computation can be counter-intuitive and affect performance unnecessarily. Besides, making sure the grid state stays updated correctly across all players over the internet can introduce additional complication. Secondly, since tiles placement can be placed diagonally, such grid would contain many cells that are inaccessible, while still being kept in memory. This would, once again, result in unused resources. Finally, as the game contains a finite set of tiles and settings, one of the best ways to set everything up is by having a prefab for each tile, with their respective settings for portal, researches, walls, and ladders placement. Consequently, having a separate grid for each tile is more manageable and scalable. This solution would also come with a benefit of being able to easily configure the two ends of a ladder, or any other tools available on the tile, as demonstrated in Figure 7.

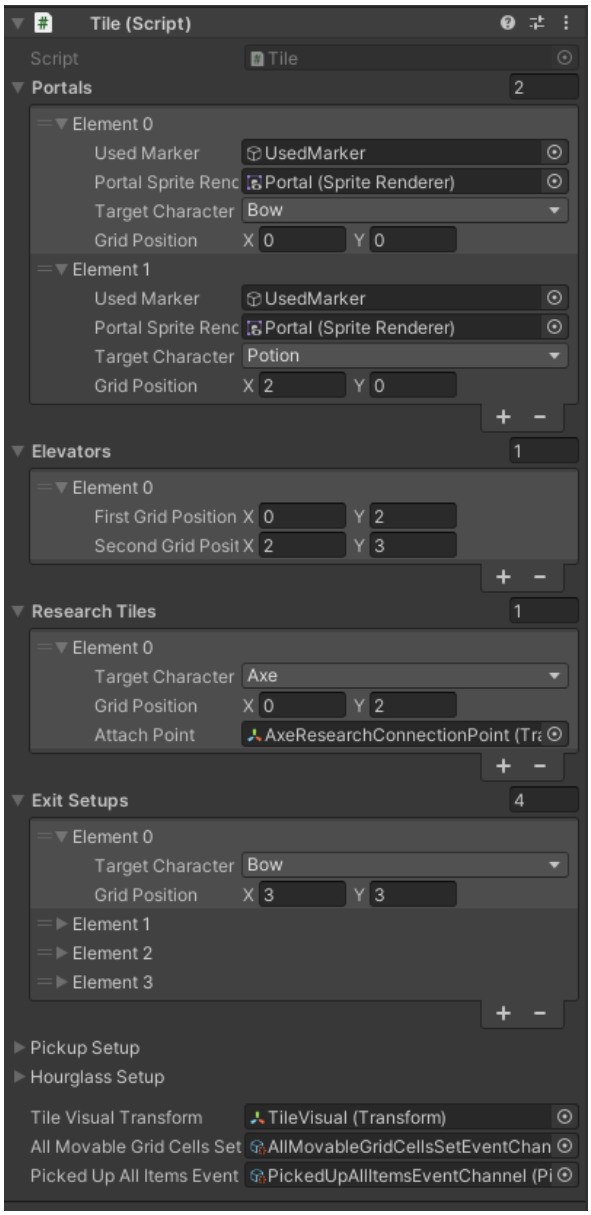


Figure 7. Example settings of a tile prefab

As with anything in software development, there is no solution that comes without any disadvantages. When new tiles are placed, it can potentially be rotated from its default rotation. Therefore, the internal grid would also need to be translated accordingly to ensure the correct placement of aforementioned tile setups. By default, in order to get the world position of a grid cell, one would only need to calculate the cell position compared to the grid starting point. However, with the additional requirement of rotation, every time such information is requested, one more step is needed to factor in the grid rotation, as shown in Figure 8.

```
private Vector3 GetCellWorldPosition(int x, int y)
{
    return FormatWorldPosition(
        _containerTransform.rotation * _visualTransform.localRotation * (new Vector3(x, y: 0, z: y) * _cellSize) +
        _originPosition);
}
```

Figure 8. Calculation of cell world position given a grid position

On the other hand, as demonstrated in Figure 9, the same calculation would need to be done when converting world position to a grid position.

```
private (int x, int y) GetGridPosition(Vector3 worldPosition)
{
    if (!_bounds.Contains(worldPosition)) return (-1, -1);

    var containerRotation :Quaternion = _containerTransform.rotation;
    var visualLocalRotation :Quaternion = _visualTransform.localRotation;
    var (x :int, y :int) = (
        Mathf.FloorToInt(
            f: Mathf.Abs((containerRotation * visualLocalRotation * (worldPosition - _originPosition)).x) /
            _cellSize),
        Mathf.FloorToInt(
            f: Mathf.Abs((containerRotation * visualLocalRotation * (worldPosition - _originPosition)).z) /
            _cellSize));
    if (x < 0 || y < 0 || x >= _width || y >= _height) return (-1, -1);

    return (x, y);
}
```

Figure 9. Calculation of grid position from a given world position

One can question the reasons behind having references to both a container and a visual transform in both Figure 8 and Figure 9. This is solely because of the unique new tiles' placement rule of Magic Maze. Every single tile in the game, except for the starting tile, contains a designated entry point when placing. This means that there is only one possible way for any two tiles to be connected. For example, in Figure 10, this is the one and only method of linking up those exact tiles. Due to this specific requirement, to allow for accurate and simple instantiation of new tiles, a separate visual transform is needed to guarantee the correct visual of the tile. When placing new tiles, one would only need to rotate the game object's forward direction to match the attach point's forward direction. As a result, the calculation in both Figure 8 and Figure 9 would need to factor in both the rotation of both the prefab parent and the visual wrapper.

Figure 10. Example of new tiles alignment

4.2.2 Movement card settings implementation

Every player in the game only has a limited set of possible moves, as indicated on their personal movement card. Since the game contains specific rules for movement cards allowed for different player count, the best way to configure all of the action sets is through the help of Scriptable Objects, shown in Figure 11.

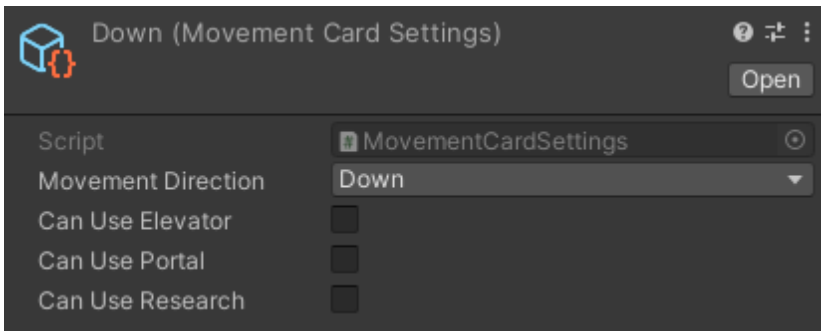


Figure 11. Example of movement card settings scriptable objects

As there are only four possible basic movement directions available, an Enum is the most suitable data type. However, Unity's default serialization does not have support for multiple choice selection for Enums, while a card can allow movement in more than one direction. Therefore, a custom property drawer,

which is basically a custom serialization, is needed. While this may appear as a simple task, there are a few technical details that one would need to know. Firstly, a property drawer only works with property attributes. Secondly, in order to develop an Enum dropdown with multiple choice functionality, knowledge of how Enum and bit operation works is the most importance. By default, each type in an Enum has an integer value. When multiple options are selected, it can be interpreted as an addition of each value. Therefore, in order to translate a total value to the options selected, one would need to use factors of two for each of the Enum type value, as illustrated in Figure 12.

```
public enum MovementDirection
{
    Right = 1,
    Down = 2,
    Left = 4,
    Up = 8
}
```

Figure 12. Example of integer values of factor of two

In addition to the movement direction settings, the card settings would also need Boolean values on whether elevator, portal, or research can be done.

4.2.3 Character controller implementation

Due to the nature of board game that Magic Maze is, a close representation of the real-life experience playing the board game is one of the key parts of creating a fun, engaging, and cooperative environment since numerous of the game's attraction is in the physical aspect of it.

One of the actions that the players are required to perform constantly is the movement of picking up the character, moving it to the target cell, and placing it down. Since the players can choose to either pick up from the bottom or the top of the yellow character, the game would need to realistically simulate the drag and drop movement based on the grab position. For example, if the player were

to pick up from the bottom of the yellow character, the player would need to move it until the hand, or the cursor in this case, to somewhere near bottom of the target cell. On the other hand, if the top of the character was grabbed, the cursor would need to move to the top area of the next cell. To implement such feature, a state for keeping track of the initial mouse position is needed. A ray could be cast from the middle of the camera to the tile, returning the 3D hit position. Afterwards, after the mouse has moved to a new location, a ray casted from the camera would returns a different hit position on the tile. Simply by subtracting the difference in the hit points, then utilizing that as the character move direction and magnitude, one could validate whether the movement distance is sufficient, having accounted for the pickup position of the character, to realistically move to a new cell.

```
var planeHitPoint:Vector3 = ray.GetPoint(enter);
var mouseMoveDirection:Vector3 = planeHitPoint - _mouseStartPosition;
var placementMouseMoveDirection:Vector3 = planeHitPoint - _placementMouseStartPosition;
var targetPlacementPosition:Vector3 = _targetGridCell.CenterWorldPosition + MathUtils.FloorOrCeilToIntVector3(placementMouseMoveDirection);
var targetPosition:Vector3 = _startGridCell.CenterWorldPosition + mouseMoveDirection;
```

Figure 13. Realistic calculation of character move distance

However, due to the nature of video games, it can potentially be difficult for players to confirm whether a character is moved to the correct position or not during the drag to move motion. Therefore, a mildly transparent character, shown in Figure 13, is shown once the drag starts as an indication of the target cell of the movement.

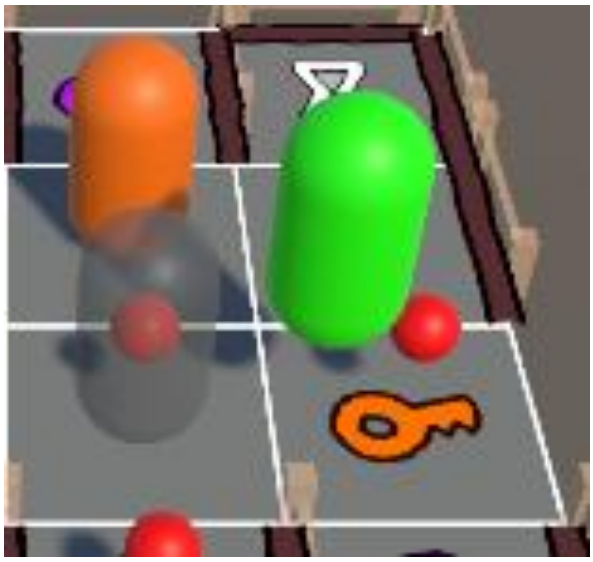


Figure 14. Transparent character as indication of target cell placement

In addition to the transparent character, certain highlights, illustrated as small red spheres in Figure 14, were found to be extremely valuable of possible target cells when moving characters during testing. Even though each player would have a HUD for their own movement card, showing directly on the tiles which kind of movement is available to them using a simple and subtle indication could significantly improve the user experience while not disrupting any gameplay.

One of the simplest ways of achieving the highlights feature is through having a list of all possible target cells. Anytime a character is picked up, such list is updated by looping through all placed tiles and all four horizontal and vertical directions, checking the player's movement card abilities with all the related actions on the tiles to get all possible cells around the map. For portals, since they could only work with characters of the same colour or type, filtering out unfitting portals is crucial. On the other hand, when it comes to elevators, the character type does not play any role. The cells on either of the two ends of an elevator would only be available if the character is already on one of them. This is demonstrated in Figure 15.

```
if (MovementCardSettings.canUsePortal)
{
    var portalGridCell =
        placedTile.GetTargetCharacterPortalGridCell(_selectedCharacter.Type);
    if (portalGridCell != null && portalGridCell.CharacterOnTop == null &&
        !_gameHandler.HasCharactersBeenOnPickupCells &&
        !_allMovableGridCells.Contains(portalGridCell))
    {
        _allMovableGridCells.Add(portalGridCell);
    }
}

if (MovementCardSettings.canUseElevator)
{
    if (_startGridCell.Elevator != null &&
        placedTile == _gameHandler.CharacterOnTileDictionary[_selectedCharacter])
    {
        var otherEndElevatorGridCell = placedTile.GetOtherElevatorEndGridCell(_startGridCell);
        if (otherEndElevatorGridCell != null &&
            !_allMovableGridCells.Contains(otherEndElevatorGridCell))
        {
            _allMovableGridCells.Add(otherEndElevatorGridCell);
        }
    }
}
```

Figure 15. Logic for filtering out non-available portals and elevator cells

In contrast to both the portals and elevators, the logic needed for simple directional movement is slightly more complicated. This is mainly due to the blockage of other characters and walls. Since a character can move in a straight line onto other tiles, and each tile contains its own grid, the only possible approach would be to check cell by cell whether it is traversable. As shown in Figure 16, only cells that contain neither walls nor characters are added to the movable grid cells list.

```
do
{
    var nextGridCell = placedTile.Grid.GetGridCellObject(nextGridCellPosition);
    if (nextGridCell == null) break;

    if (!Physics.Raycast( origin: startGridCell.CenterWorldPosition, movementDirection, maxDistance: TilePlacer.CellSize,
            (int) wallLayerMask) && nextGridCell.CharacterOnTop == null)
    {
        _allMovableGridCells.Add(nextGridCell);
        startGridCell = nextGridCell;
        nextGridCellPosition = startGridCell.CenterWorldPosition + movementDirection;
        continue;
    }

    break;
} while (true);
```

Figure 16. Logic for checking whether cells in a direction are traversable

4.2.4 Camera controller implementation

One other important feature to be able to “sell” the game is the camera controller. Since the goal is to allow the players to enjoy Magic Maze how it can be felt when playing the board game in real life, and the camera is essentially the eyes of the players, a smooth and intuitive camera controller plays a key role in making the game fun.

Nowadays, whenever a unity project is created, one of the default assets that any developers should be using is Cinemachine. It is a built-in package for more complex camera behaviours, such as spanning or subtle camera shaking. Additionally, one feature, which is referred by many as the most useful tool in Cinemachine, is the readily made feature of camera follow and look at. These are some of the most commonly needed behaviours on a camera that the default Camera component within Unity does not provide.

```
_newPosition.x = Mathf.Clamp( value: _newPosition.x, ColliderBounds.min.x, ColliderBounds.max.x);
_newPosition.z = Mathf.Clamp( value: _newPosition.z, ColliderBounds.min.z, ColliderBounds.max.z);
```

Figure 17. Example of camera position clamping

There are three basic camera actions available for any board game digitalization, the movement, the rotation, and the zoom. Beginning with the movement, there are a few things that needs to be considered. Firstly, the camera would need to be confined to certain area. Even though there are several ways to accomplish the feature, the path chosen in this study is through using a 3D box collider, since one can utilize the bounds of it to restrict the possible position of the camera, as demonstrated in Figure 17.

As for camera rotation, since most of the game interaction is through the mouse, the most intuitive control of camera would also be through the same device. After some investigation of how other digital board game handles camera rotation, as well as through some actual AB testing, comparing the use

of keyboard versus mouse, using the right mouse button for the rotation of the camera is by far the best option. Another important aspect to consider is with default rotation, the game object would rotate around itself. Even though this effect is perfect in some situations, such as first-person games, the players in digital board games would most likely need the camera to rotate around a certain point to see things from a new perspective. One of the simplest methods of achieving this is by docking the camera game object under another game object as parent. Then, one would need to keep the parent position at the world's zero point, and simply position and rotate the camera view so that it points directly at the parent position, as demonstrated on Figure 18.

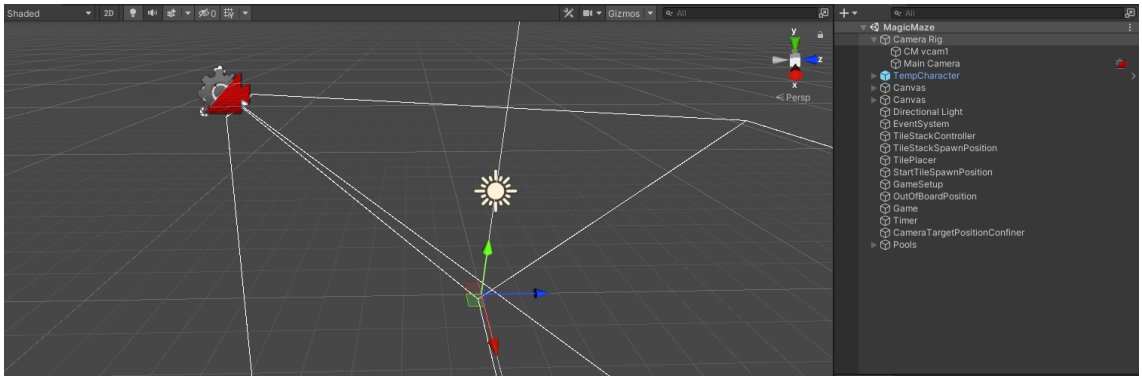


Figure 18. Example camera setup for camera rotation around one point

Afterwards, instead of updating the rotation of the game object's transform, it is the local rotation that should be renewed, since it is the rotation related to the parent.

Finally, the camera zoom was initially configured to slowly go up towards a ninety-degree until the view is directly on top of the board. However, after several tests with other users and normal playtesting, the behaviour was uniformly agreed upon as counter-intuitive and not useful in most cases. It was found that the best option would be to zoom in and out on the camera's forward direction. One thing to note is that similarly with the movement, the camera would also need to have a restriction on how high and low it can go, which could be accomplish using the same collider bounds mechanic.

5 Multiplayer implementation

Even though it is playable, Magic Maze cannot reach its fullest potential when playing without friends.

5.1 General setups

When it comes to integrating multiplayer mechanics into Magic Maze, it is in general more complexed comparing to many other games, be it digital board games or not. This is mostly due to the fact that the players share the control of all characters within the game, instead of the more convention one-to-one relationship.

5.1.1 Matchmaking implementation

Photon PUN has one of the simplest setups for multiplayer in Unity. There are only two configuration steps needed one can dive into the script. Firstly, since PUN is not a self-hosted multiplayer solution, a server would need to be created to store all of the running game sessions. This, which should be created for each game through their website, is fully maintained by Photon.

Figure 19. Example of PUN server created on Photon service

As shown in Figure 19, the free-tier server of Photon would only allow for 20 concurrent users (CCU). In addition, there is a graph for the CCU over time in case one would need to upgrade the server to keep it up and running.

One other important detail to note from Figure 19 is the App ID line. In order to connect the game in Unity to the correct server on Photon's cloud, the value there is needed. Within Unity, provided by Photon PUN with the unity package is a ServerSettings scriptable objects, which is used to config everything PUN related.

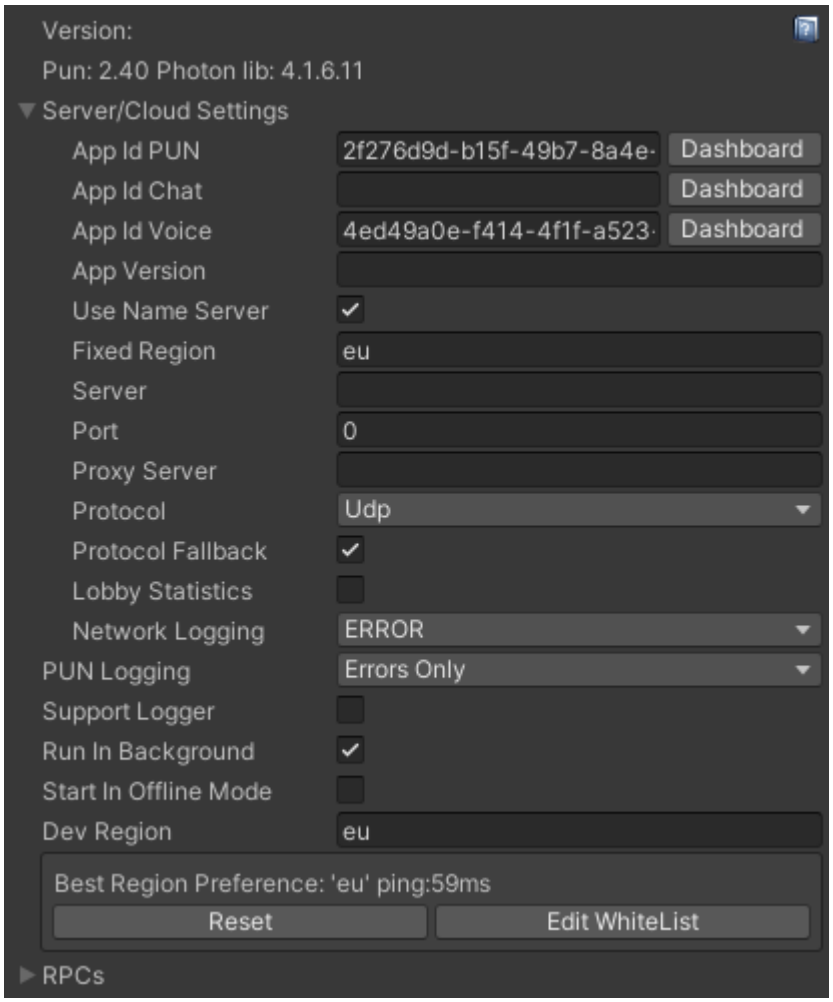


Figure 20. Photon PUN ServerSettings scriptable objects

Additionally, there are various other settings that can be changed depending on the game and how the developers want to integrate the multiplayer into it. For example, by default Photon would pick the best region for the server depending on where the player is. Though this behaviour is desirable in most cases, the team can decide only limit to the EU region, either for development or legal purposes.

Even though there are different ways of connecting the game to the server, the default method recommended by Photon is by having a script inheriting from a custom MonoBehaviour made by the team, called MonoBehaviourPunCallbacks. Its purpose is to allow developers to make custom game logic when the server connectivity state changes. In most cases,

once the game successfully connects to the master server, the next step would be to join a lobby. Unless customized, every single player would be connected to the same one. Within the lobby, there can be an infinite number of rooms. The inherited class contains built-in method for custom logic handling when new rooms are added, updated, or removed.

```
public override void OnRoomListUpdate(List<RoomInfo> roomList)
{
    foreach (var info in (IEnumerable<RoomInfo>)roomList)
    {
        if (info.RemovedFromList || !info.IsOpen)
        {
            if (!_cachedRoomList.ContainsKey(info.Name)) return;

            Destroy(_cachedRoomList[info.Name].gameObject);
            _cachedRoomList.Remove(info.Name);
        }
        else
        {
            var spawnedRoomListItem = Instantiate(roomListItemPrefab, roomInfosContainer);
            spawnedRoomListItem.Setup(info);
            _cachedRoomList[info.Name] = spawnedRoomListItem;
        }
    }
}
```

Figure 21. Custom logic for UI update when lobby room list is updated

As shown in Figure 21, whenever the room list of the lobby is updated, a scrolling list of all items will either be stripped off some items or added with some new ones. There are two occasions when a room list item should be removed from the open list, either when a room is completely deleted from the lobby, or when a room is closed. Since this logic varies game by game, the method provided by PUN allowed for simple customization. For example, in this digital version of Magic Maze, once a group has decided to start a game, the room would be closed from the public, preventing new players from joining in. This is mostly due to the complication it can cause to existing players since the movement cards setup is player-count dependent. In other games, such as a first-person shooter (FPS) game, a room is usually open for more players since the experience could only improve with more players in such genre.

A room list item is an UI representation of an available room in the lobby. This is one of the simplest matchmaking methods to implement using PUN. The players can then pick the correct room to join in with their friends based on the room name. It should also show how many players are already in the room, as shown in Figure 22. Once the players are in the room, the players can start the game session once the sufficient number of players is met. In most of the cases, only the room owner should be able to do that.

Figure 22. Example of room list item on the UI

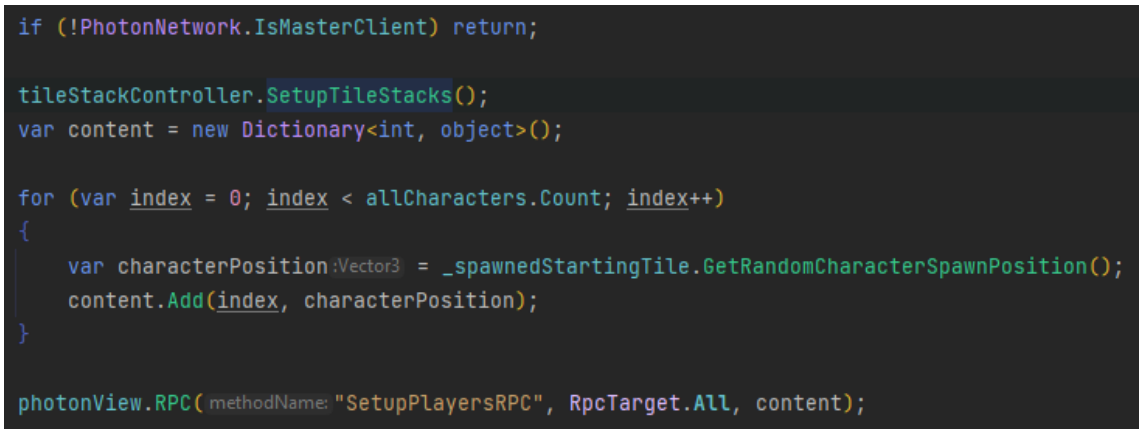
5.1.2 Game initialization

Once the room starts, when it comes to initializing the game, Magic Maze differs vastly from common online multiplayer games, due to the fact that all players control every character available in the game. Since Photon PUN provides many different methods of syncing the game states between clients, choosing the most suitable one for the situation at hand can be challenging. For the initialization of the board, the approach decided on is doing most things on the master client, since it is the client with most reliability and lowest latency. In the Awake method of a GameSetup script, the most important step to take on the master client is to get the correct movement card setups for the number of players present in the room, as shown in Figure 23.

```
private void Awake()
{
    if (Instance != this)
    {
        Destroy(Instance);
        Instance = this;
    }

    if (!PhotonNetwork.IsMasterClient) return;

    for (var index = 0; index < movementCardsSetupCollection.allSetups.Count; index++)
    {
        var movementCardsSetup = movementCardsSetupCollection.allSetups[index];
        if (movementCardsSetup.playerCount != PhotonNetwork.CurrentRoom.PlayerCount) continue;

        _cardSetupIndex = index;
        var random = new SystemRandom();
        var randomMovementCards :IOrderedEnumerable<MovementCardSettings> = movementCardsSetup.cardSet.OrderBy(item :MovementCardSettings => random.Next());
        _runtimeMovementCardSettingsList = new List<MovementCardSettings>(randomMovementCards);
    }
}
```

Figure 23. Awake method on GameSetup script to initialize movement card

Afterwards, each game client would need to instantiate its own starting tile, instead of spawning one first in the master client and syncing it. This is mainly due to the fact that nothing is needed to be synced other than the spawned tile transform. Other than spawning in the first tile, each client would also need to request a random movement card settings from the main client. This can be done through RPC, which is essentially a remote method caller on another client, with the ability to send custom data if needed.



```
if (!PhotonNetwork.IsMasterClient) return;

tileStackController.SetupTileStacks();
var content = new Dictionary<int, object>();

for (var index = 0; index < allCharacters.Count; index++)
{
    var characterPosition :Vector3 = _spawnedStartingTile.GetRandomCharacterSpawnPosition();
    content.Add(index, characterPosition);
}

photonView.RPC( methodName: "SetupPlayersRPC", RpcTarget.All, content);
```

Figure 24. Master client game setup for tile stacks and characters position

All of the other steps needed should be done on the master client, as shown in Figure 24. Firstly, the client would need to setup the tile stacks randomly to be

drawn later on. The master client should be the one spawning in new tiles since it is the most trustworthy one. Secondly, to make sure the character position stays the same across clients, the characters are initiated with random starting cells, then the data could then be broadcasted using RPC to all other clients to update correspondingly.

5.2 Syncing the game states

When it comes to syncing the various game states, the development of the game went through different methods and approaches in order to find the best option.

Within PUN, there is an important concept known as ownership. In any multiplayer game, there are multiple types of control rights of objects. First of all, any game objects that exist in the scenes on load that do not have a PhotonView component are available on all clients. In addition, any updates made on one client do not get replicated onto the others. For example, in most cases, the camera game object should not be harmonized since each players need to have their own perspective when playing the game. For Magic Maze, there are several objects, such as the tile stack position or the starting tile spawn position, are network independent since its position and state does not get changed during the game. They are simply used as references for correct placement. Secondly, in order to make game object states synchronize between clients, the method recommended by Photon is through a ready-made component called PhotonView. Any objects containing such component is designated an identification number. This id is then used by Photon during gameplay to find and update the same object on all other clients, as shown in the Figure 25.

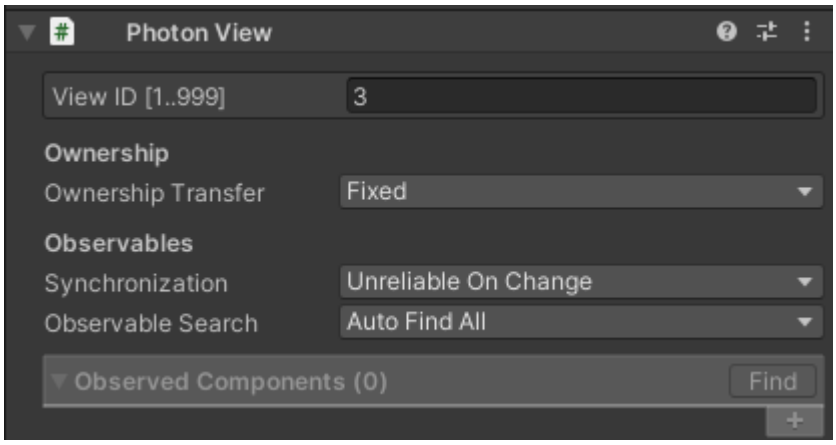


Figure 25. A PhotonView component in the Inspector

Underneath the View ID variable is the ownership configuration. By default, if the game object with the component exists on scene start, it is owned by the master client. However, when any players instantiate a game object over the network, the object is then owned by the creator. Such objects cannot be controlled or updated by any clients other than the owner. In addition, unless configured, the ownership of a synced object does not get changed during the game. In order to perform such action, the other two available options for the ownership transfer field are “Takeover” and “Request”. If takeover is picked, as soon as an ownership change request is received, the game object is instantly no longer owned by the same player. Request, on the other hand, is a less invasive and slower process of ownership transfer. In this case, a request would need to be received and accepted by the controller before the sender could gain full rights to the game object. The final and most important section of the PhotonView component is the Observables. It allows developers to add any custom synchronization logic to a component list. For example, in a game where players would share a high score, such data could be easily updated correctly across all clients.

Finally, a game object could be instantiated as a room object. It is a special type of networked object where no players have ownership of it. This is mostly used for things that are level dependent. For example, in an FPS game, there could be logic to spawn an airdrop every few minutes. Such object should not be

owned by any players, but rather the entire room. A normal scene object would not work, as it needs position synchronization across all clients so that all players have a chance at picking it up.

5.2.1 Character movement controller script ownership

Among the different scripts to synchronize between clients, the character movement controller script is arguable the most challenging one since it plays the core role in the gameplay. Due to the complexity, it underwent various methods of synchronization.

One might think that since each player should have their own character controller to be able to independently move the figures around, there should be one game object within the scene that contains the script component. This was the original approach taken in this study due to the intuitiveness. However, as soon as the implementation is done, a challenging problem arose. Even though the network logic of the script makes sense, the actual thing that gets its position update are the characters on the board, rather than the script component. The simplest method to update the state of the figures correctly is through the use of PhotonView. At first glance, this might seem simple and obvious, but one key component of the tool is the ownership. If PhotonView component is used, there is no reasonable target as the owner for the characters. Even though the transfer configuration can be marked as “Takeover”, there would still be delay between handover, affecting the gameplay.

As a result, the approach taken is to let each client locally instantiate their own characters. At game start, the master client would broadcast the spawn location of each type of character to all others, in order to make sure that they are all placed at the same position. Afterwards, for every player, a new instance of separately owned movement controller is instantiated over the network.

5.2.2 Characters movement synchronization

Using the ability to add custom synchronization logic to the PhotonView component, whenever a character is picked up and moved by a player, such info would be broadcasted to every other player so that they can update their own instance of the figures. Even though this method cannot guarantee a lag-free position update of characters during the drag motion, it is negligible since only the final position of a character plays an important role. On placing a figure to a target cell, a couple different RPC calls would need to be made to ensure a smooth and plausible character placement. Firstly, since the master client is the most trustworthy client, a validity check would be requested from it. While the check is being handled, another RPC call is sent to all clients, including the master one, to place the character at the target cell. Even though this essentially bypass the check, it would not force the players to wait for the validation result as due to the network latency, it can take several milliseconds before it reaches back to the initiator. However, if the validation failed, the master client could send a cancel signal to all other players to basically perform an undo action on the character.

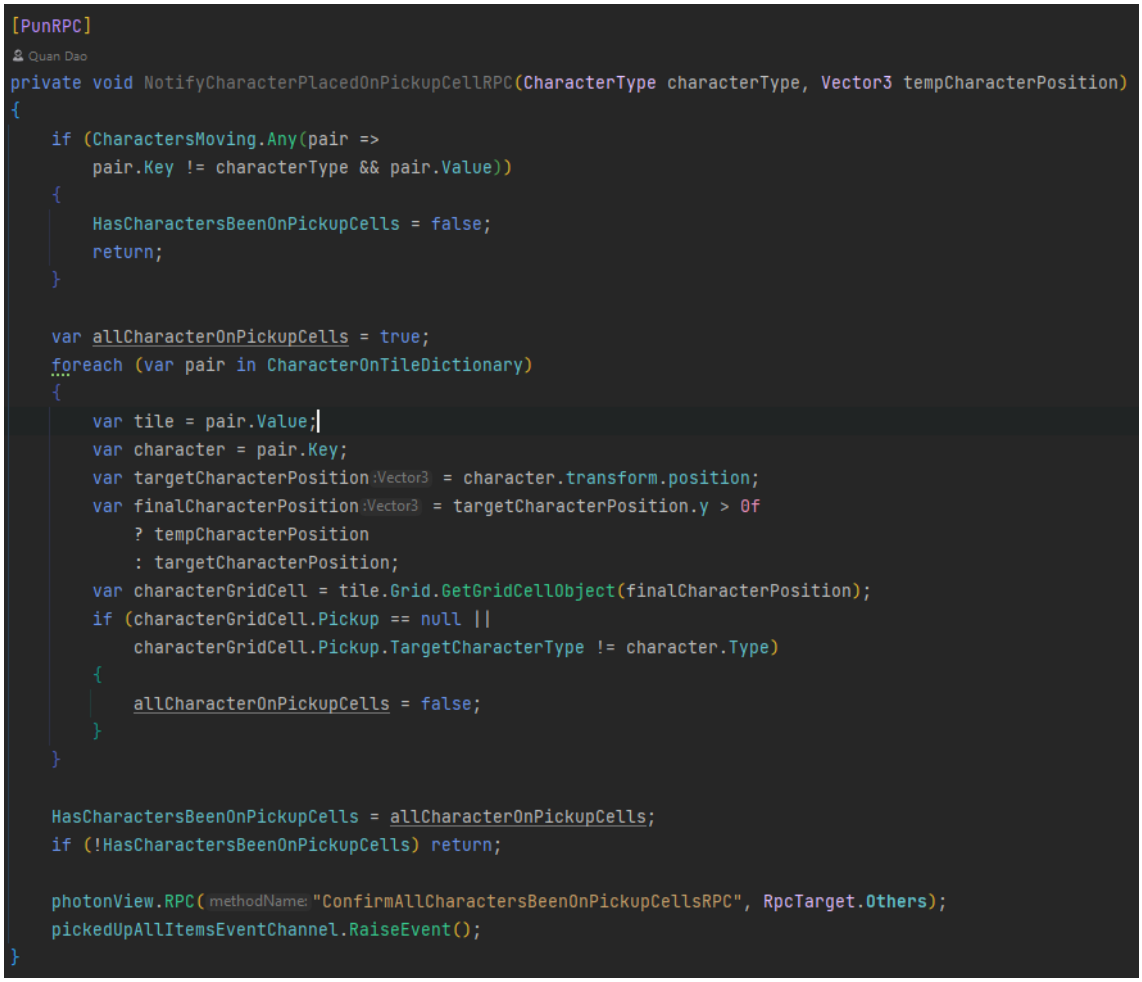
5.2.3 Exploring the map

Similar to the character movement, every time tile research is requested, an RPC would be sent to the master client for validity check since the tile research relies heavily on the correct character placement. If a client has managed to cheat the characters position, it would be unsafe to allow any player to handle the tile exploring on their own. In addition, in order to reduce the wait time, the tiles would be placed one by one instead of all at once.

5.2.4 Picking up objectives

In Magic Maze, as soon as all characters are on all of the objectives, the portals are no longer usable by any players. In addition, only then can the figures enter the exit cell and win the game. Once again, the concept and the importance of

the master client plays an important role in implementing a safer and more reliable game state synchronization across all players. Whenever a character is recognized as placed on an objective cell on any client, an RPC would be sent to the server to check whether it is true or not, and whether all of the characters are on all of the cells with objective.



```
[PunRPC]
Quan Dao
private void NotifyCharacterPlacedOnPickupCellRPC(CharacterType characterType, Vector3 tempCharacterPosition)
{
    if (CharactersMoving.Any(pair =>
        pair.Key != characterType && pair.Value))
    {
        HasCharactersBeenOnPickupCells = false;
        return;
    }

    var allCharacterOnPickupCells = true;
    foreach (var pair in CharacterOnTileDictionary)
    {
        var tile = pair.Value;
        var character = pair.Key;
        var targetCharacterPosition :Vector3 = character.transform.position;
        var finalCharacterPosition :Vector3 = targetCharacterPosition.y > 0f
            ? tempCharacterPosition
            : targetCharacterPosition;
        var characterGridCell = tile.Grid.GetGridCellObject(finalCharacterPosition);
        if (characterGridCell.Pickup == null ||
            characterGridCell.Pickup.TargetCharacterType != character.Type)
        {
            allCharacterOnPickupCells = false;
        }
    }

    HasCharactersBeenOnPickupCells = allCharacterOnPickupCells;
    if (!HasCharactersBeenOnPickupCells) return;

    photonView.RPC( methodName: "ConfirmAllCharactersBeenOnPickupCellsRPC", RpcTarget.Others);
    pickedUpAllItemsEventChannel.RaiseEvent();
}
```

Figure 26. RPC method on master client to validify the objective cells placement

As shown in Figure 26, if all characters are on all of the cells with the items, a different RPC would be sent to all other clients to notify them of the game state update.

5.2.5 Syncing the game timer

Magic Maze is a very time sensitive game. As soon as the time runs out, all of the players are instantly lost. Therefore, making sure that the timer matches across all clients, whenever a change occurred, plays a crucial role in the fairness and quality of the gameplay. Some methods were considered for this study, but due to the complexity of any time related operation in a multiplayer game, a fail-proof result was never achieved. However, a “good enough” technique is implemented.

```
private void Update()
{
    CurrentTime = Mathf.Clamp(value: CurrentTime - Time.deltaTime, min: 0f, maxTime);
    timerUI.UpdateTimer(CurrentTime / maxTime);

    if (CurrentTime == 0)
    {
        gameEndedEventChannel.RaiseEvent();
    }
}
```

Figure 27. Timer’s Update method to achieve a timer feature

First of all, creating a perfect timer, simulating a real-world clock, in C# is nowhere near a simple feat. Within Unity, even though there would still be difference or delay comparing to an actual clock, a timer could be achieved through the Update method and built-in delta time variable. Update is a method that is called every frame on any component. In live game situations, a system can run the game at very different frames per second. Consequently, Unity provides a variable that represents the time taken between frames, called “deltaTime”. Combining both concepts, a frame-independent timer could be accomplished in Unity, as demonstrated in Figure 27.

```
var content = new object[]
{
    _tilePlacer.AllPlacedTiles.IndexOf(_targetGridCell.Tile),
    _targetGridCell.CenterWorldPosition, _timer.CurrentTime
};
photonView.RPC( methodName: "UseHourglassRPC", RpcTarget.Others, content);
_targetGridCell.UseHourglass();
```

Figure 28. Example of timer synchronization using custom parameters

In a multiplayer environment, a timer synchronization could be reached using the custom parameters that can be sent alongside with any RPC call. The master client could send along the current time on the client whenever the hourglass is flipped, as shown in Figure 28. On receiving the data from master player, the client could then use the time to have a pseudo time synchronization. This method is not taking into use the time taken for the network package to be sent from one player to another.

6 Conclusion

The study demonstrated the full process of developing a multiplayer game using Unity engine and Photon PUN plugin. In addition to the walkthrough of the technicalities, the study also covers the design aspect of digitizing a board game. Even though Magic Maze already contains its own rules and designs, due to the difference in nature for a board game and its digital version, many decisions were made to provide a fun and engaging gameplay. Everything needs to be displayed and simulated clearly, concisely, and realistically, such as the rules instruction, the status of different elements of the game, the game phase, and the characters movement.

Different challenges and misunderstanding during development are explained clearly and concisely to provide an effective guide. The concepts of RPC, Owner, Controller, and the techniques of using Transform, and calculating

quaternions relative to world and relative to parent were discussed and described in detail to offer the chosen solution to solve such obstacles.

Finally, as a crucial part of any application development, playtesting resulted in various possible improvements, especially the ones that were not considered during development.

References

1. Rossella Lorenzi. Oldest known gaming tokens dug up in Bronze Age Turkish graves. Available from: https://www.nbcnews.com/sciencemain/oldest-known-gaming-tokens-dug-bronze-age-turkish-graves-6C10920354. [Accessed 9 November 2022]

2. Jesse Schell. The Art of Game Design, 3rd Edition. A K Peters/CRC Press; 2019.

3. Ethan Ham. Tabletop Game Design for Video Game Designers. Routledge; 2015.

4. Kotaku.com. How We Made *The Last of Us*'s Interface Work So Well. Available from: https://kotaku.com/how-we-made-the-last-of-uss-interface-work-so-well-1571841317. [Accessed 9 November 2022]

5. Sue Blackman. Beginning 3D Game Development with Unity: The World's Most Widely Used Multiplatform Game Engine. New York, USA: Springer Science + Business Media, LLC.; 2011.

6. Store.unity.com. Compare Plans. Available from: https://store.unity.com/compare-plans. [Accessed 2 November 2022]

7. Alan Stagner. Unity Multiplayer Games. Packt Publishing; 2013

8. Photonengine.com. Showcase. Available from: https://www.photonengine.com/en-US/PUN/Showcase. [Accessed 2 November 2022]

9. Lisa Bock. Learn Wireshark – Fundamentals of Wireshark. Packt Publishing; 2019.

10. Brajesh De. API Management: An Architect's Guide to Developing and Managing APIs for Your Organization. Apress; 2017.

11. David B. Makofske, Michael J. Donahoo, Kenneth L. Calvert. TCP/IP Sockets in C#. Morgan Kaufmann; 2004.

12. Visualstudio.microsoft.com. Vs. Available from: https://visualstudio.microsoft.com/vs/. [Accessed 7 November 2022]

13. Azure.microsoft.com. Home. Available from: https://azure.microsoft.com/en-us/. [Accessed 7 November 2022]

14. Visualstudio.microsoft.com. Game Development. Available from: https://azure.microsoft.com/en-us/. [Accessed 7 November 2022]

15. Youtube.com. Fast C# Scripting in Unity with Jetbrains Rider! – Overview. Available from: https://www.youtube.com/watch?v=2CvSoo0hlWI. [Accessed 7 November 2022]

16. Jetbrains.com. Rider for Unity. Available from: https://www.jetbrains.com/lp/dotnet-unity/. [Accessed 7 November 2022]