172114203 & 152113001

Strategy game built on Unity3D Lightweight Render Pipeline that supports mobile and computer platforms with Mirror Networking Asset based multiplayer and an Artificial Intelligence with MiniMax Algorithm.

Duzce University 2020 Summer Thesis Project

Özgür Özbek & Atilla Çoruhlu

20.08.2020

Abstract

Purpose of this article is to provide an insight on both common and uncommon problems encountered by the engineering end with LWRP, Mirror and MiniMax algorithm and how to approach the problem to solve it efficiently. This also mentions some optimizations regarding those topics. Since this is a thesis project, detailed instructions on how to achieve these problems are mentioned. Topics reviewed will have a clear indication of separation with titles and subtitles. Sources can be found under references. They mention how to use or achieve a state while this article points out how we used them and what we encountered. Every code block shown is not intended to be used as is for you, and the software is licensed under the MIT License.

Keywords: Unity, Render Pipeline, MiniMax, Artificial Intelligence, Mirror, Tic Tac Toe

Explaining how to solve various problems encountered by using the current game development technologies mentioned in the thesis project to make a multiplayer game.

T.C. DUZCE UNIVERSITY FACULTY OF TECHNOLOGY COMPUTER ENGINEERING DEPARTMENT

Strategy game built on Unity3D Lightweight Render Pipeline that supports mobile and computer platforms with Mirror Networking Asset based multiplayer and an Artificial Intelligence with MiniMax Algorithm.

This project prepared by Ozgur Ozbek and Atilla Coruhlu has been accepted by the following jury as the **Bachelor's Degree Graduate Thesis** in Duzce University Faculty of Technology Computer Engineering Department.

Thesis Advisor
Prof. Dr. Ibrahim Yucedag
Duzce University

Jury Members

Duzce University _____

Duzce University _____

Duzce University _____

Thesis Defense Date: 14.09.2020

Statement

This thesis study is our own work, we did not behave unethically in any stages from the planning stage to the release. We have obtained all information in this thesis within academic and ethical rules. We have cited all the information and comments obtained from outside this thesis. Their sources on the time of writing are included in the list of references. We did not violate patent and copyright rights during the study and writing of this thesis.

September 14th, 2020

Ozgur Ozbek

Atilla Coruhlu

Thanks

Many thanks to our professors Prof. Dr. Ibrahim Yucedag, Dr. Metin Toz, AP. Ali Calhan and AP. Serdar Birogul. Thanks to their expertise on artificial intelligence, optimization, networking and programming skills, we think we came up with the best approaches to certain problems.

Thanks to the people who have always supported us during these times, our families and friends, Emine & Faruk Ozbek, Ayse & Kemal Coruhlu, Aleyna Selale Seker, Kubra Nur Cin, Umut Demiray, Efe Bozoklu, Arda Arinkal and Mustafa Meral.

September 14th, 2020

Ozgur Ozbek

Atilla Coruhlu

Table Of Contents

Tools Used And Their Explanations

We mention what we used and how we used them here. Tools that are not required but used like Unity Hub, Github Desktop and Trello are not mentioned.

Unity3D

Unity3D is a game engine. It includes things that are required to make a game such as audio drivers, multiplatform compilers, rendering etc.

Visual Studio Code & Monospace

It is used to integrate our code to Unity3D while including quality of life tools for programming like Linter, Parser and, Error Handling methods.

Adobe Illustrator

It is a designer software we used to make almost every two dimensional asset such as buttons and icons or texture assets like tiles and buttons that are used in the game.

Lightweight Render Pipeline

It is a render pipeline that is already optimized to run better on both CPUs and GPUs. It is integrated on Unity3D. We used it to render the board, particles and any other material used. We have used tools like the PostProcessing Stack, Shaders and Shader Graph. Shader Graph and its nodes allow us to use many rendering effects with customization.

MiniMax Algorithm

This is an algorithm that tries to get the highest possible score within the current node while iterating recursively. This is used when a single player hosts a game to play. When there are no opponents, the artificial intelligence will play to provide a satisfying session. Currently it only thinks ahead for three turns because we wanted it to move fast and be beatable.

Software

We used C# to communicate with Unity3D, Markdown for documentation, C# and Python for the artificial intelligence, C++ for socket programming, NodeJS for server client relations and, Git for version control.

Game Theory

In order to make many competitive decisions during the development phase, mathematical formulas proposed by John F. Nash were used to help make rational decisions.

Unity Asset Store

Used to include packages like Mirror into our project.

Reaper, Kontakt, Audacity & BoscaCeoil

It was used when doing sound engineering for in-game music, orchestration, sound effects and more.

GitHub

Used to version control. We also had to modify how Unity3D stores metadata and how it saves files to be able to track the project with Git.

Being able to pull and push individually let us work on the same project simultaneously.

Scenes And Their Explanations

These are the screens displayed by Unity3D and they show the output or help us navigate through. Every information is provided and game logic runs under these scenes.

Menu Scene

This is the first scene we encounter when we run the game. This scene has the option to mute and unmute the sound, buttons to play, to see the tutorial and to exit. Menu Scene also displays the version and the title.

We can change how the game table looks by clicking the “easter egg” button and see the info scene by clicking the info button.

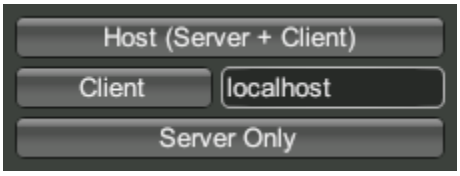
Tutorial Scene

This scene displays the achieved rules mentioned under the “Logic And Rules” section.

Info Scene

This scene displays the credits information. The information includes content under titles developers, testers and information providers. It also has a disclaimer to indicate that this is a thesis project.

Offline Scene



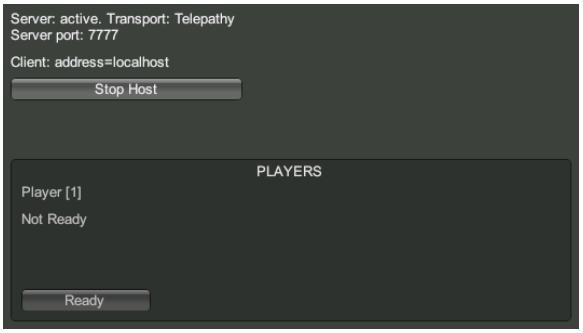
This scene allows us to Host a Server+ Client network connection. We can play with others on localhost if our firewall is not blocking this process or play with others online if our 7777 port is open.

Client button also lets us to join others. Others in this context is indicated by the Input Field. On this picture, it is localhost.

If nobody joins the host, the host can still start the game and play against the artificial intelligence.

We use Mirror to provide networking needs.

Online Scene

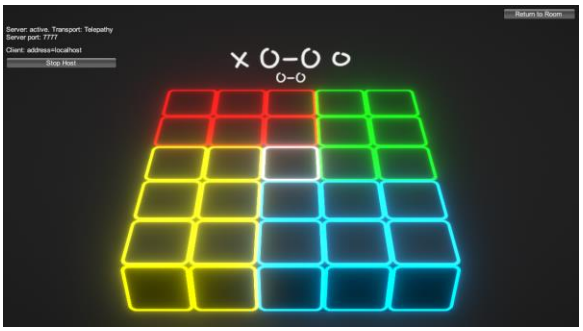


This scene displays the information for the room such as the NAT Port, Client Address and network transport type. Host can choose to terminate this host session aswell.

We also see a lobby. This lobby only takes up to two players since the game can only be played by two players. Players can ready up and the host can start the game is everyone is ready.

We can also kick other players if needed.

Game Scene



This is the game scene. Every game related logic is being run under this scene. Score, Buttons, Moves, Game Commands, Networking Commands, Client Remote Procedure Calls, Hooks and SyncVars run here.

Mainly, two players can play against each other. Host will always go first. Score is indicated aswell.

If there are no players, MiniMax will act as a player and play instead.

Scene Transitions

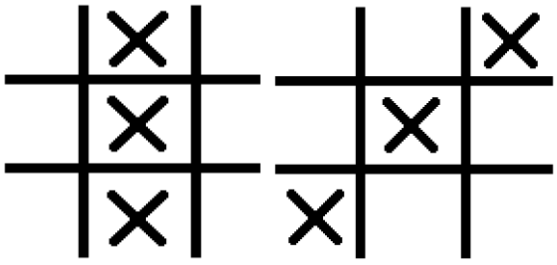
We wanted a visual feedback while changing scenes. Simply, fading in and out solves this problem.

We have used the Animator embedded in Unity3D. In a canvas, we faded by changing an image's alpha value. Due to performance concerns, we instead decided to work with a code. This also helped us to include this transition on multiple instances.

We got rid of the animator and ended up with a one liner that allows us to transition between scenes.

Logic And Rules

Firstly, information about Tic-Tac-Toe has to be given. This game is played on a three by three board, usually with a pen and paper. Player scoring by aligning three of their pieces either diagonally, horizontally or vertically, wins the game.



X goes first, then the game will continue with O. X being the first player, O being the second. Maximum amounts of moves is 9 and is as follows.

X,O,X,O,X,O,X,O,X

Although, the game will end if one of the players score. Unlike the regular Tic-Tac-Toe, on XOOOX, we placed four boards on top of each other. This was inspired from 4D Chess. Thanks to this, there is more replayability and strategy. With the artificial intelligence, even without a friend, you can still play so there are very little problems while looking for companions.

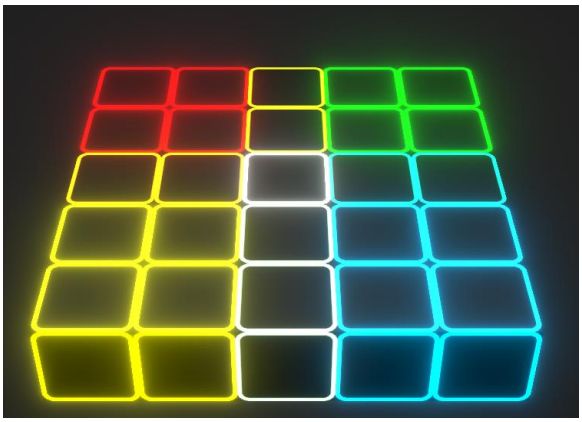
Logic

The game is played on a five by five field. Middle square represents a corner for all four tables. Even though we see a five by five area, it is actually an overlapping 6 by 6 area that has four three by three area.

For user experience, we color-coded the table. For overlapping squares, we have a shader effect that transitions between two adjacent table colors. White representing the middle square is unique, since it is the only tile that every table has access to. That is why it is white.

White is rgb(255,255,255)

Since some versions of Unity3D differ between the methods for their shaders, we kept all effects such as transitions and emissions compact as possible in render pipeline.



Aside from all that, contrary to common Tic-Tac-Toe, our game does not end when a player scores. The player with most combos wins.

Logic Gaps

There are four gaps that need to be addressed immediately.

1) What is the value of a combined square?

2) How do we decrease the amount of ties?

3) How to even out the advantage from going first?

4) We don't want odd combinations like;

red middle + white + blue middle

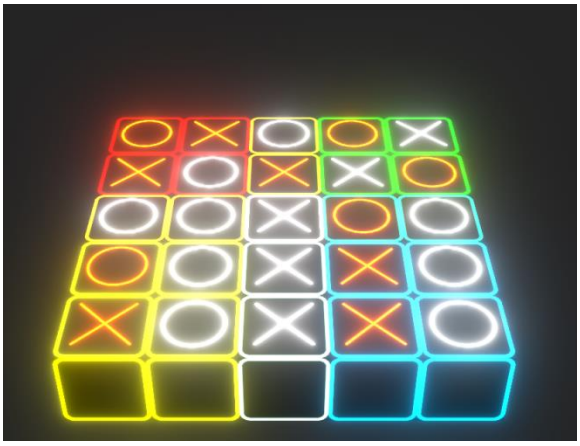
to score, so how do we divide the five by five table to four three by three tables in code?

Answers To These Gaps

These issues need to be answered to as simply and clearly as possible.

We do not want to provide complex solutions and take away from the fun aspect.

1) As you can see in the following image,



even though X's have two scoring combos, since one of them is on a combined area it scores from both the tables. This situation increases the amount of ties.

2) Tic-Tac-Toe has a fame of ending in a tie, and a tie is not as fun as winning -or even losing-. To decrease this amount, we have already got rid of the "score once to win" rule. Now, we are going to check the amount of tiles we gained score from. If the scores for players equals, we are going to count the tiles.

As you saw before, X and O both have the score of 3. But, with this new rule, X has 5 tiles it gained score from while O has 9. Therefore, O is rendered as the winner.

3) Since X has a massive advantage, we need to give a chance to O.

Middle square according to our MiniMax table has the score of 12 while the next highest score is 6. This means that we can just simply give O two turns to equally handle the advantage problem. This way, O can also have the advantage of having played tiles on all tables, also rendering X's advantage of going first to none.

3	2	6	2	3
2	4	4	4	2
6	4	12	4	6
2	4	4	4	2
3	2	6	2	3

MiniMax Table

4) We have already provided a visual solution for the table organisation problem. We calculate every table seperately in code in different arrays. This makes us calculate the combined fields more than one but also prevents code repetition. Calculating combined areas twice also acts as an attention grab, deepening strategy.

Rules

According to everything mentioned above, our rules follows:

- Played with two players.
- X goes first.
- O goes twice on the second round.
- Combined areas of three by three tables score double.
- When a player scores, the game does not end.
- When all twenty-five squares are played, game ends. Player with the higher score wins.
- If scores are equal to eachother, player with the most scoring tiles wins.
- If scores are still tied, it is a tie.

Pre-Development Phase

Every asset shown in earlier screenshots are made by us inside Unity3D or with Adobe Illustrator. For tiles, X and O we used MSPaint and LWRP's Emission settings. These emission settings are derived from a PBRGraph.

After all visuals and drawings are in line with the gameplan, we began programming the main logic. At this state, we used four different multi-dimensional arrays to store the data for all our tables. This way, we could track score easily. If we were to calculate this with a five by five array, it would both be slower and hard-code heavy.

Image you see has additional shadowing because every vector drawing we need to use in game has to be white because we handle all coloring and texturing from our shader graphs. Emission, HDR Coloring, Light Bouncing should be provided by the lightweight render pipeline. This way, every color or hue-saturation adjustment we make is correct.

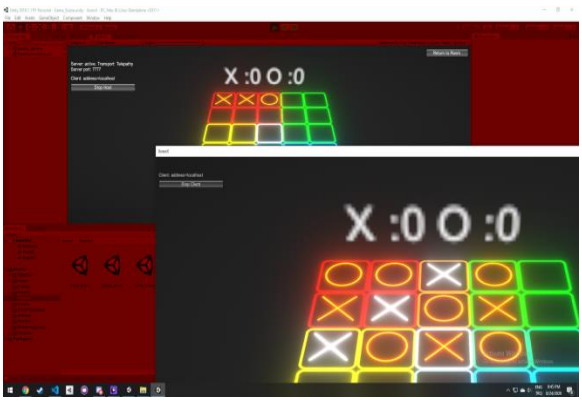
After programming the "Button.cs" script, we attached it to all our squares. In these squares, script runs on click and in an object oriented way transfers data to our "GameMaster.cs" script. This script then handles all array related operations.

Score, Move Number and more is stored under "GameMaster.cs". After calculation, score is then transferred to "Score.cs" where it finally displays the current score.

After handling turns logic, making buttons clickable only once, making opponent marked buttons not clickable and fixing other non-sensical errors, the game is playable on one screen with one mouse for two players.

Multiplayer Development Phase

There are many important variables defined on previous versions. These variables need to be tracked and synced on multiplayer. For transmission to be fast and effective, Mirror only supports primitive data types. These are strings, chars, integers, floats and other similar data types. An integer variable, moveNumber can be synced without any extra effort using SyncVar decorator. Although it is synched, we cant show server side changes on the client side. Here is an example.



Data Types And Synchronization

To accomplish array syncing, we need to define its method for Mirror to use. We added a new class that inherits one of Mirror's syncable classes. Afterwards, we made a struct that holds primitive data types. We used a constructor to get our previous dataset that uses arrays then set it as our new dataset using this struct. We then arranged our previous code to use this struct. This swap to multiplayer data structure cost us valuable time, but thanks to the switch our client and server run synced. They are not visible currently but it is easily fixed by adding scripts client specific command decorators and server specific remote procedure calls.

Authentication

Client → Server data transferring is not possible without an Authenticator. Since Mirror only provides us with simplified authenticators we moved almost every syncable function to server side from client side. Every server side function this way does not require an authenticator.

Since we also never allow the client have an authenticator, server scripts only run on server side, as intended. This also gives us a new layer of security.

By using the Command decorator we let clients call server functions. By using the ClientRPC decorator we let the server update states and run commands on clients. This way, both client and server are able to see activity on themselves made from the other side. Server handled score calculation material update, move increases and more, therefore we had no issues on synchronization. Every client is now able to see every effect and SyncVars.

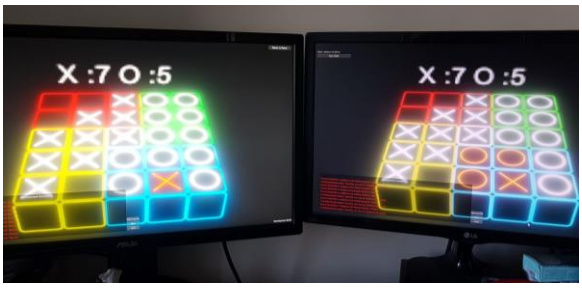
Before doing this with primitive data types but without commands and RPCs, we encountered a computer science problem called race condition.

Race Condition

Race condition occurs when two or more processes try to access a shared data and the change it.

This condition in our case is happening simply because multiplayer networking. Two atomic processes run at the same time to be synchronious and desynchronize because one side is faster or undergoing complex calculations. While this error was difficult to identify and fix, Unity3D's timers and Mirror's Command - RPC system has provided us with a massive advantage. Now, calling the Update in Fixed times with FixedUpdate and then running either commands or RPCs fixes this.

The image below shows serverside effect functions running asynchronously on clients. Everything other than that like scoreboard is correct and still in sync. This proves that there was a race condition.



Right screen as server and left screen as client; image provided can clearly show how client's blue table effects are no longer in sync because of the race condition. We can also see that the score on top is correct and synchronous.

Nagle's Algorithm

This algorithm enhances data transmission between TCP/IP networks by decreasing the amount of packages. Sometimes, objects on the same network needs to receive a call from the server and since this needs to happen at the same time, between ticks, there is conflict. This conflict rendered the game to be unplayable at a quick pace. However this is easily solved by increasing the tickrate to almost double of what was before and then implementing Nagle's Algorithm to gain some network speed back.

Beyond Multiplayer

After answering all immediate networking problems, we now need to implement a turn mechanism to the game. This means we need to wait for the server to play X or wait for the client to play O. Since all commands already run on the server side and especially synchronization is made with primitive data types we can introduce a variable for this.

The game can now be player on network whether LAN or WAN. WAN needs things like port forwarding and a static ip but these things are not problems for this thesis paper.

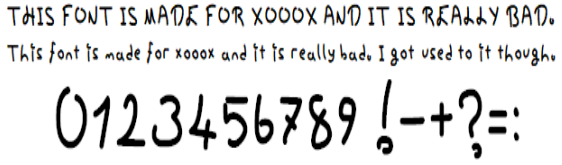
We then added a menu, sound effects, music, skoring trackers, tutorials and many other small things that make the game a game.

Asset Creation

We needed visuals and other enhancements to make the game more like a game, and we did not want to implement unnecessary and non-commercially licensed anything from outside. Since we wanted some character aswell, we created our own.

Font

One of the most important original asset is the font. We created the XOOOX font. First used Adobe Illustrator to make the letters and then used the online tool calligraphr. Using this we first made a template then added the font file to Unity3D. Here is an image for displaying the font.



Music

Using BoscaCeoil, we created a simple midi file. Then using Reaper we recreated a similar tone and with Kontakt, orchestrated it. We liked how it sounded with classical orchestra instruments. Afterwards we adjusted these sound files with Audacity and gained in tune sound effects and a loopable background music. Files were then added to Unity3D project folder.

We have two problems with this. Since the game has multiple scenes, our music should not restart everytime a scene changes or

just stop. It needs to be constant. Using Unity3D's DontDestroyOnLoad method and script instancing this is easily handled.

Other problem is that the game is playable online. We don't want to hear two music on top of eachother or other sound effects that the other side is hearing. We added an instancable audio listener component to all network managed player prefabs and these are either synced or not managed by our AudioManager script. Even though we used more than what we needed in terms of audio channels, this solved our problem.

Sound effects being asynchronous is also a minor issue. We called these effects from our synced GameMaster. This way there is an additional check but no players hear synced audio twice.

Singleplayer Gameplay

Firstly, learning from popular mobile games, adding a mock online player is obviously the way to go. An artificial intelligence in our case that can play a competitive game is good enough. Single player happens when the hosting server+client hits ready on the lobby. GameManager checks if there is one client playing, and calls the MiniMax Artificial Intelligence to play if the check is true.

MiniMax Algorithm

Only using the scores and table states has zero effect. Usually the table state is fed to get the correct assumption from the AI upon player movement. AI then decides the right move based on some score calculation. Since our game does not end with one end game score unlike traditional tic-tac-toe, we need to optimize and customize the AI. There are couple things that the AI should know. There are also more optimizations available like alpha-beta pruning, randomized outputs and more, but our project does not need it yet and therefore they are not mentioned in this paper.

Table Optimization Problem

Firstly we needed a good point system for the table. Every tile is not equal and we already know this so we can simply feed the AI this information.

To calculate these we firstly gave 1 point to every tile. This way common tiles ended up becoming 2 pointers while the middle tile ended up becoming a 4 pointer. When we tested this with Python however, AI did not make sensible moves to win or to block us.

To fix this we are going to give the AI extra points for blocking our combo and scoring combos.

Even though it's movements were sensible, it's moves on board specific tiles seemed random. To fix this we needed to change the board scores.

Previously we gave every tile on the board 1 score. Even though they were not equal, their added bonus would help was what we thought. Now, we removed it and applied a different approach.

We scored the tiles on a simple question.

"How many combos can this be a part of?"

After answering this, and implementing, on a three by three board corner tiles equal 3, center tile equal 4 and the rest is 2 points. When combined, some are doubled and this is even better for us. This removes the need to say that some tiles are combo tiles to the AI.

♥	A	B	C	D	E
1	3	2	6	2	3
2	2	4	4	4	2
3	6	4	12	4	6
4	2	4	4	4	2
5	3	2	6	2	3

This table shows the latest achieved five by five matrix.

Şekilde görülen masa puanlaması en son ulaştığımız 5x5 matris değerlerini gösteriyor. Bu noktada ise şunu düşünmemiz gerekmekte. Kombo yapma veya kombo bozma durumunda karelere kaç puan ekstra verilmeli?

Combo Scoring Problem

These can't be equals however. MiniMax chooses the first best option for it's current iteration. So it would choose the best option closest to the button number. We also don't want to combo on the left side while the AI combos on the right side. If we interrupt it, we would easily win. So this proves that our point of not making them worth the same point.

We can't also just say "Give combo making a 10, and interrupting a 20!". It needs to be sensical with the previously made table scores.

To examplify this, let's say we made a combo at the A2 tile. This could let us make another combo next turn. Therefore, it's combo value should be equal to B2 or C4. Because when you combo, your opponent can combo aswell. 2 points is the best combo value for this example since it is close to the standart derivative of the board. If you were to interrup the combo for the same tile, you would need to get a score higher than 2 but lower than the minimum higher number (which is 6).

$$Tile + Combo = Points\ Lost$$

$$2 + 2 = \mathbf{4}$$

$$Min(Higher\ Tile + 3) - Combo = Points\ Gained$$

$$X > (2 + 3 = 5) - 2 = X - 2$$

3 is added because it is the interruption bonus.

$$X = 6;$$

$$6 - 2 = \mathbf{4}$$

As you can see, both equations equal the same number for the most worthless tile. This balance is good enough for us.

Therefore to fix the issue mentioned before, we gave scoring a +2 bonus and combo interruption a +3 bonus.

Emotional Behaviour Problem

After a while, the AI feels repetitive because how MiniMax behaves on equal outcomes. It takes the first one, we don't want to store and pick a random one aswell because as depth increases, this would ruin the AI's plans.

We have implemented a random bool value that is representing the AI's anger. Unlike traditional tic-tac-toe which ends with one score, we can implement this to our game. Simply;

If angry → 2 more points from blocking

We can also choose to randomize the first move of the AI because our table is symmetrical. This will also give us more replayability but is not needed yet.

Key Scripts

This section will provide you with some parts from the scripts and explain what they do. These are crucial for this project.

GameMaster.cs

Now, lets talk about Mirror SyncVars.

```
[SyncVar(hook=nameof(RpcMoveAdvance))
]
    public int MoveNumber = 0;
```

This piece of code will keep MoveNumber in Sync between all clients, but is not yet returned to the clients. That is why SyncVar decorator needs a hook.

```
[ClientRpc]
public void RpcMoveAdvance
(int _oldint, int _newint)
{ MoveNumber = _newint; }
```

This is the hook we mentioned before. As you see, it replaces the old integer value from clients with the new one from server. This is how we track our move count.

```
[System.Serializable]
    public class SyncListTuple : Sync
List<Item> { }

    [System.Serializable]
    public struct Item {
        public string a;
        public string b;
        public string c;
    }
```

This is how we communicate with the table. As you can see, we can't use arrays but we converted them to structs using primitive data types.

```
SyncListTuple syncRed =
new SyncListTuple ();
SyncListTuple syncBlue =
new SyncListTuple ();
SyncListTuple syncGreen =
new SyncListTuple ();
SyncListTuple syncYellow =
new SyncListTuple ();
```

Every table is created and tracked like this.

```
public override void OnStartServer ()
{ base.OnStartServer (); }
```

This is how we start the Mirror Server.

```
void Awake () {
if (NetworkServer.connections.Count =
= 1) {
MiniMax.board = new int[5, 5] { { 0,
0, 0, 0, 0 }, { 0, 0, 0, 0, 0 }, { 0,
 0, 0, 0, 0 }, { 0, 0, 0, 0, 0 }, { 0
, 0, 0, 0, 0 } }; } }
```

This is how we initiate the MiniMax script if there are no other clients than us.

```
[ServerCallback]
public void _beginFill () {
for (int i = 0; i < Red.Length / 3; i
++) {
syncRed.Add(new Item(){a=Red[i,0],
b=Red[i,1],c=Red[i,2]});
syncGreen.Add(new Item(){a=Green[i,0]
,b=Green[i,1],c=Green[i,2]});
syncYellow.Add(new Item(){a=Yellow[i,
0],b=Yellow[i,1],c=Yellow[i,2]});
syncBlue.Add(new Item(){a=Blue[i,0],
b=Blue[i,1],c=Blue[i,2]});
}

syncRed.Add(new Item(){a="red",b="",
c=""});
syncGreen.Add(new Item(){a="green",
b="",c=""});
syncYellow.Add(new Item(){a="yellow",
b="",c=""});
syncBlue.Add(new Item(){a="blue",
b="",c=""});
}
```

This piece of code here is a good demonstration of a code that clients should not run. This is achieved with the ServerCallback decorator.

```
void FixedUpdate () {
xCount = 0;
xCount += Check (syncRed, "X");
xNumber = ComboCount ("X");
}
```

This is how we keep track of score. xCount is the score, xNumber is the amount of scoring tiles.

This is also a good indicator of how we show scripts in this paper. They are not whole in this state. We need to check syncBlue, syncGreen and syncYellow aswell but this is more understandable and easier to read.

Also in FixedUpdate();

```
if(
NetworkServer.connections.Count==1) {
if(Moves[MoveNumber] == "O") {
int hamle = MiniMax.nextTurn ();
RpcChangeArray (hamle.ToString ());
GameObject.Find (hamle.ToString ()).
GetComponent<Button>().
CheckMaterial = 1;
MoveNumber++;
} }
```

this is called and this is the initiating code for the MiniMax.

```
[ClientRpc]
public void RpcChangeArray (string Bu
ttonName) {
if (Moves[MoveNumber - 1] == "X") {
GameObject.Find (ButtonName).GetCompo
nent<Renderer> ().sharedMaterial = ma
terialXO[1]; } else { GameObject.Find
 (ButtonName).GetComponent<Renderer>
().sharedMaterial = materialXO[2]; }

if (isServer) { switch (ButtonName) {
#region InsertToTables
#endregion } } }
```

This is a good example of how we used RPCs. This piece of code will run on all clients but not on server. It changes the button material to show X and O.

```
public class GameMaster :
NetworkBehaviour { }
```

GameMaster is a public class that inherits Mirror's NetworkBehaviour.

```
[ClientRpc]
public void RpcComboGen (string a, st
ring b, string c, int material_) {
GameObject.Find (a).GetComponent<Rend
erer> ().sharedMaterial = materialXO[
material_];
Destroy (Instantiate (ComboParticle,
GameObject.Find (a).transform.positio
n, GameObject.Find (a).transform.rota
tion), 2f); }
```

This is how we generate combo situations and change their material. This is called from another RPC to change colors and that is called from the Check function that runs on FixedUpdate.

```
public int ComboCount (string Variabl
e) { int temp = 0; for (int i = 0; i
< 25; i++) { if (GameObject.Find ((i
+ 1).ToString ()).GetComponent<Render
er> ().sharedMaterial.name == "Combo"
+Variable) { temp++; }} return temp;}
```

This is how we calculate scoring tiles.

```
[Header("MiniMax AI Specifications")]
    public int minimaxDepth = 3;
    public int minimaxBrain = 3;
```

We also keep the settings for our MiniMax in our GameMaster. Header decorator is for Unity3D.

```
public string[, ] Green = new string[
9, 3] { { "", "3", "" }, { "", "4", "
" }, { "", "5", "" }, { "", "8", "" }
, { "", "9", "" }, { "", "10", "" },
{ "", "13", "" }, { "", "14", "" }, {
 "", "15", "" } };
```

Our old dataset looked like this but now they are inserted into the structs shown before.

Button.cs

```
[SyncVar]
    public int CheckMaterial = 0;
```

Button also has a SyncVar to keep it's material. Material can not be synced because it is not a primitive data type but an integer can.

```
public void ButtonLogic () {
if (CheckMaterial == 1) {
return;
}
CheckMaterial = 1;
}
```

This is how we use the SyncVar. This prevents the players from clicking on top of eachother's tiles.

```
CmdClickIncrease ();
```

Also in ButtonLogic this is called to increase the move number. It runs the following.

```
[Command (ignoreAuthority = true)]
private void CmdClickIncrease () {
GameMaster.instance.MoveNumber++;
}
```

A command decorator is to make Server run commands from clients. Ignore authority tag inside is to accept all client side requests. It is needed because every player can click on an empty tile.

```
Destroy ((GameObject) Instantiate (Ga
meMaster.instance.GetParticle (), tra
nsform.position + offset + new Vector
3 (0f, 1f, 0f), transform.rotation),
2f);
FindObjectOfType<AudioManager> ().Pla
y ("Move");
```

Still in ButtonLogic this piece of code runs on online play and it plays an effect and a sound.

```
CmdSendButtonName ();
```

This is also called in ButtonLogic and it sends the required data to GameMaster to proceed with its functions.

```
[Command (ignoreAuthority = true)]
private void CmdSendButtonName () {
GameMaster.instance.RpcChangeArray
(this.name); }
```

It is used like this. GameMaster then changes the arrays and then displays it in a synced way.

```
MiniMax.Call(
Convert.ToInt32(this.name));
```

MiniMax is called from the button when the button name is needed.

```
public class Button :
NetworkBehaviour { }
```

Button is also a public class that inherits Mirror's NetworkBehaviour.

AudioManager.cs

Audio Manager is a MonoBehaviour class. It also benefits from a sub-class that is called Sound.cs. This file has options like file path, sound and pitch.

```
void Awake()
{ if(instance == null){instance=this;
} else { Destroy(gameObject);
return; }

DontDestroyOnLoad(gameObject);

foreach (Sound s in sounds) {
s.source = gameObject.AddComponent<Au
dioSource>();
s.source.clip = s.clip;
s.source.volume = s.volume;
s.source.pitch = s.pitch;
s.source.loop = s.loop; } }
```

This script is the initiation point for our AudioManager. This has it's own empty gameObject just like the GameMaster.

```
void Start () { Play("Theme"); }
```

On start, this will play the theme song for our game.

```
public void Play (string name) {
Sound s = Array.Find(sounds, sound =>
sound.name == name);
if (s == null) { return; }
s.source.Play(); }
```

This is the script that makes our sound played. It is really easy to be called aswell.

```
FindObjectOfType<AudioManager>().Play
("Move");
```

This is a good example.

```
public class AudioManager :
MonoBehaviour { }
```

AudioManager is a public class that inherits MonoBehaviour from Unity3D.

MiniMax.cs

```
public static int[, ] board = new int
[5, 5] { { 0, 0, 0, 0, 0 }, { 0, 0, 0
, 0, 0 }, { 0, 0, 0, 0, 0 }, { 0, 0,
0, 0, 0 }, { 0, 0, 0, 0, 0 } };
```

This is to let the AI know what a board looks like.

```
public static int countPoints
(int value) {
var score = 0;
if (GameMaster.instance.minimaxBrain
>= 1) {
//Score Declarations
return score; }
```

This is how the AI calculates score.

```
public static int _MiniMax (int[, ] b
oard, int depth, bool isMaxing) {
if (depth == GameMaster.instance.mini
maxDepth) {
return countPoints (1);
}

if (isMaxing) { var bestScore = -
999999; for (int i = 0; i < 5; i++) {
for (int j = 0; j < 5; j++) { if (boa
rd[i, j] == 0) { board[i, j] = 1;
var score = _MiniMax (board, depth +
1, false); board[i, j] = 0;
if (score > bestScore) { bestScore =
score; } } } } return bestScore;
} else { var bestScore = 999999;
for (int i = 0; i < 5; i++) { for (in
t j = 0; j < 5; j++) { if (board[i, j
] == 0) { board[i, j] = -1;
var score = _MiniMax (board, depth +
1, true); board[i, j] = 0;
if (score < bestScore) { bestScore =
score; } } } } return bestScore; } }
```

This is how the AI operates. It first checks if it is the maxing depth, controls all available fields, play to one, calculates score by calling itself recursively, removes the play.

```
public static int nextTurn () { var b
estScore = -999999;
int[] bestMove = new int[2];
for (int i = 0; i < 5; i++) { for (in
t j = 0; j < 5; j++) { if (board[i, j
] == 0) { board[i, j] = 1;
var score = _MiniMax (board, 0, false
); board[i, j] = 0;
if (score > bestScore) { bestScore =
score; bestMove[0] = i;
bestMove[1] = j; } } } }
board[bestMove[0], bestMove[1]] = 1;
return moveBoard[bestMove[0],
bestMove[1]]; }
```

This is what is called from the GameMaster.

TableMaterialController.cs

This is given because it will be a good example for the next section.

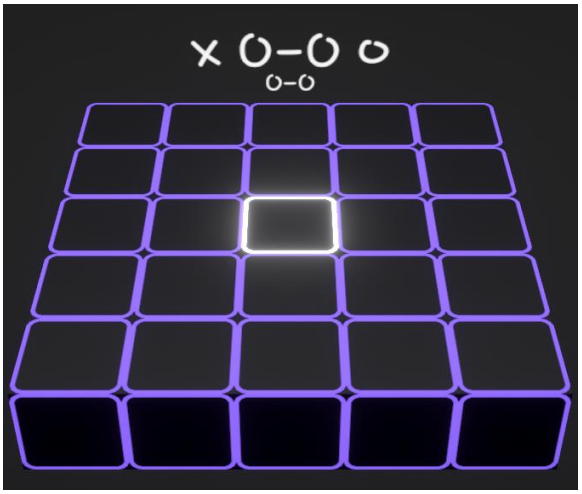
```
using UnityEngine;

public class TableMaterialController:
MonoBehaviour {
public Material eggMaterial;
private void Awake () {
if (PlayerPrefs.GetInt("Egg",1)==0) {
this.GetComponent<Renderer> ().shared
Material = eggMaterial; } } }
```

This calls a PlayerPref then changes the material for the tiles if needed.

PlayerPrefs And Other

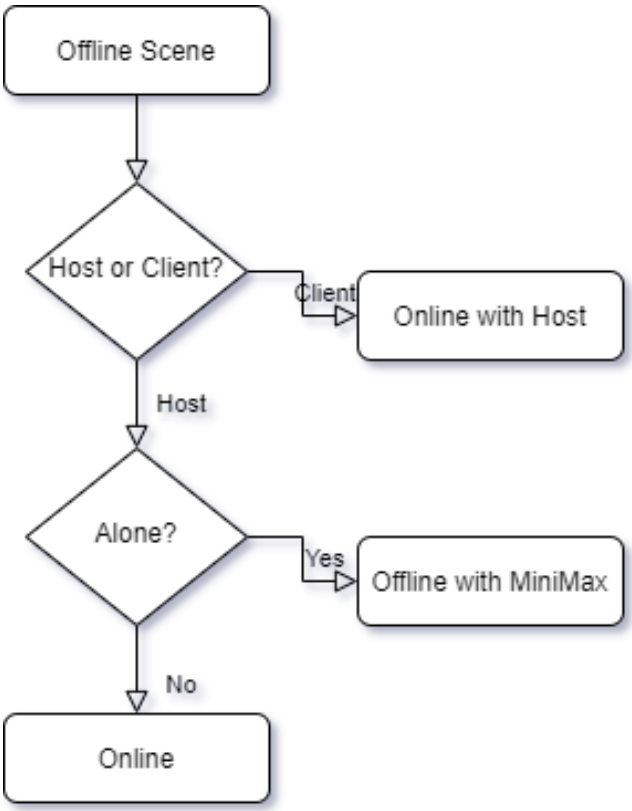
PlayerPrefs are used in two instances. One is used for the Mute-Unmute button for the music, the other is used for the table materials.

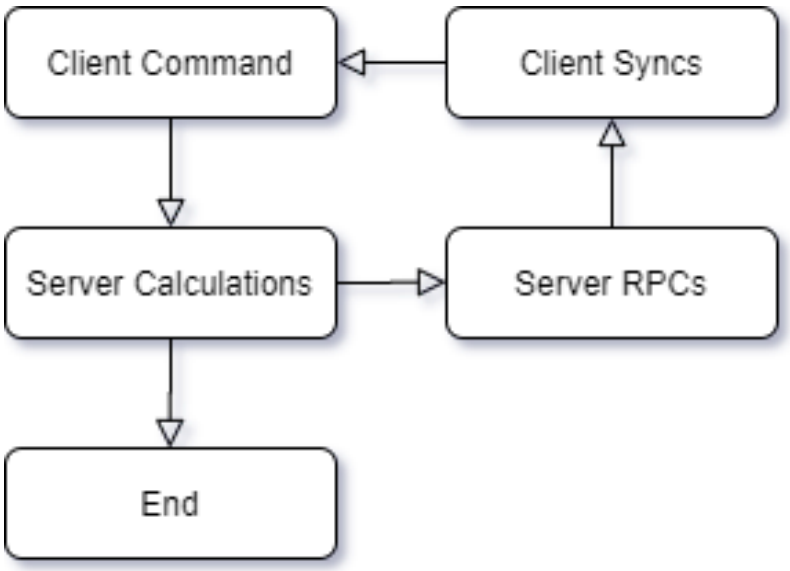


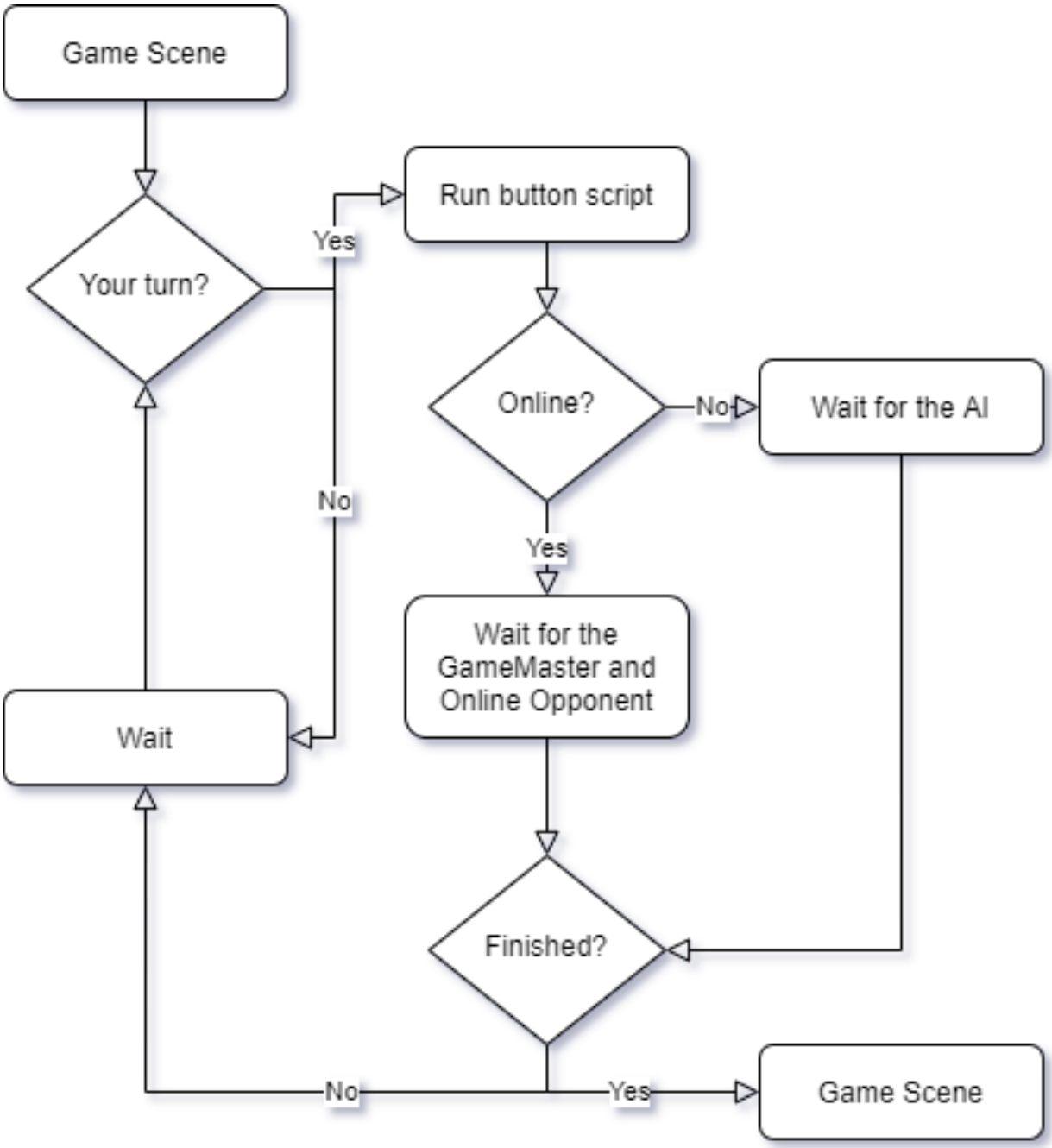
PlayerPrefs are Unity3D's save methods. Our game states can also be saved like this but since there is no way to give the same numbers to every network identity, it is impossible for us to currently save the game.

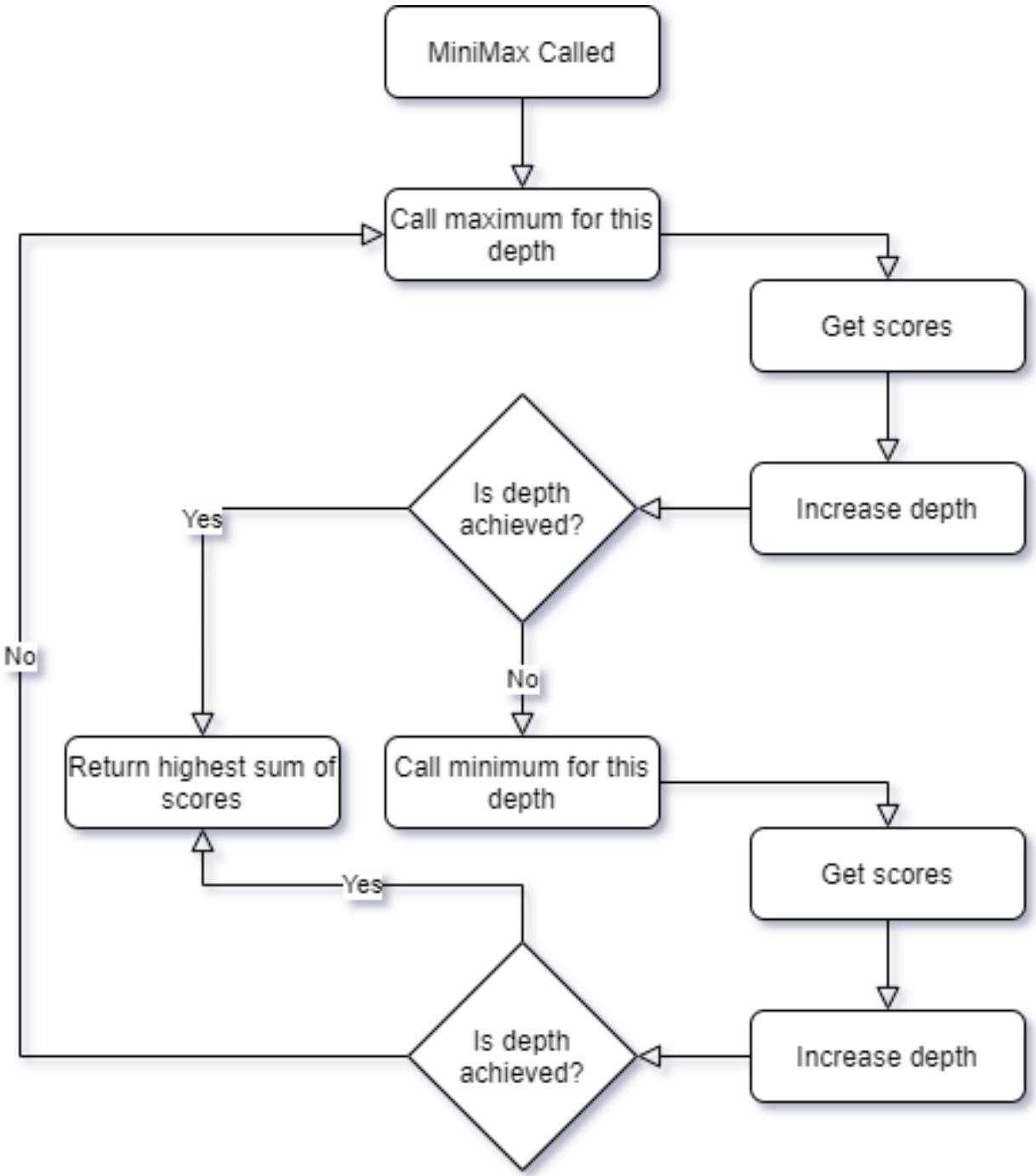
Flow Diagrams

This section shows how some things work graphically.






Game Scene
Your turn?
Yes
Run button script
Online?
No
Wait for the AI
Yes
No
Wait for the GameMaster and Online Opponent
Wait
Finished?
No
Yes
Game Scene


MiniMax Called
Call maximum for this depth
Get scores
Increase depth
Is depth achieved?
Yes
No
Return highest sum of scores
Call minimum for this depth
Get scores
Increase depth
Is depth achieved?
Yes
No

License

Bibliography

This is what we used to get references for Unity3D and read the documentation.

1. https://docs.unity3d.com/Manual/

This is the Mirror documentation.

2. https://mirror-networking.com/docs/

This is how we setup the starting code for MiniMax.

3. https://en.wikipedia.org/wiki/Minimax
4. https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/
5. https://www.youtube.com/channel/UCvjgXvBlbQiydffZU7m1_aw

This was needed for some problem solving in regards to LWRP.

6. https://www.youtube.com/user/Brackeys

This was needed for Mirror and understanding Mirror's structure.

7. https://www.youtube.com/c/FatDino
8. https://www.youtube.com/channel/UCGIF1XekJqHYIafvE7l0c2A

This is where we derived our main game logic from.

9. https://en.wikipedia.org/wiki/Tic-tac-toe

This is the C# documentation.

10. https://docs.microsoft.com/tr-tr/dotnet/csharp/

This is the LWRP documentation.

11. https://docs.unity3d.com/Packages/com.unity.render-pipelines.lightweight@5.10/manual/index.html

Articles about MiniMax that we gained an understanding from.

12. KAAN GÖKCESU(2017), Online minimax optimal density estimation and anomaly detection in nonstationary environments
13. AMIN FARIDYAHYAEI(2017), A multi-level continuous minimax location problem with regional demand
14. Dzhafarov Vakif, Karamançıoğlu A, Çetintaş S (1997),Minimax optimal control for one class of uncertain systems

This is what we used to better integrate the MiniMax algorithm to our game AI.

15. https://www.gamebusiness.jp/article/2019/07/09/15947.html

These are the music related information tabs.

16. https://www.reaper.fm/userguide.php
17. https://www.native-instruments.com/fileadmin/ni_media/downloads/manuals/kontakt/KONTAKT_6.4_Manual_26_08_2020_ENGLISH.pdf

This is the explanation that helped us solve our own race condition problem.

18. https://www.baeldung.com/cs/race-conditions

This is the implemented Nagle's algorithm.

19. https://en.wikipedia.org/wiki/Nagle%27s_algorithm

Nagle's Article on TCP/IP Internetworks

20. https://tools.ietf.org/html/rfc896

This is the font template website.

21. https://www.calligraphr.com/en/

This web application was used to create flowcharts.

22. https://app.diagrams.net/