

Contents

1.	Basic softwares.....	2
1.1	Unity 2020.03.30.....	2
1.2	Microsoft Visual Studio.....	2
1.3	Microsoft Visual Studio Code.....	2
1.4	XAMMP	2
2.	Database.....	2
2.1	Users table	2
2.2	User_settings table	4
3.	Main menu	5
3.1	Intro.....	5
3.2	Login.....	8
3.3	Sign up.....	10
3.4	Change password.....	12
3.5	Options menu	13
3.6	Choose play mode.....	14
4.	Gameplay	14
4.1	Check for win	16
4.2	One player mode.....	17
5.	Camera animations	20

1. Basic softwares

1.1 Unity 2020.03.30

The entire game is written in Unity game engine. I chose Unity because its main programming language is C#, which I've studied for the last 5 years.

1.2 Microsoft Visual Studio

Visual Studio is the main IDE we use in school, so I'm comfortable with it that's why I thought that it would be perfect for this project.

1.3 Microsoft Visual Studio Code

I used Studio Code to write HTML, CSS and PHP files.

1.4 XAMMP

XAMMP is one of the main components of the project. It runs all the main servers needed for the game (Apache, SQL, Mercury).

2. Database

The player has the option to create an account so he can save his options into the database. In order to be able to provide that, I created a database with two tables.

2.1 Users table

Login credentials and information about the account are stored in this table.


	#	Name	Type	Collation	Attributes	Null	Default
<input type="checkbox"/>	1	id 	varchar(50)	utf8_hungarian_ci		No	None
<input type="checkbox"/>	2	email	varchar(50)	utf8_hungarian_ci		No	None
<input type="checkbox"/>	3	username	varchar(20)	utf8_hungarian_ci		No	None
<input type="checkbox"/>	4	password	varchar(255)	utf8_hungarian_ci		No	None
<input type="checkbox"/>	5	active	tinyint(1)			No	0

Figure 1: The structure of the users table

The table contains 5 columns:

- ID: An automatically generated unique string that is used to identify an account. This is the primary key for the table.
- Email: Contains the email address that is used to communicate with the user.
- Username: The username that the user signed up with.
- Password: The password that the user signed up with. It is automatically encrypted using PHP's password_hash() function.
- Active: A boolean value indicating whether the account has been activated or not. The default value is 0 (false).

id	email	username	password	active
d02fR03b8EqIPfE-YsDzjw	example@example.com	example	\$2y\$10\$SqsgiM/yKq59aTirL1UoEujLOktehYzulnnWIWWSk/I...	0

Figure 2: An example of an inactivated account

2.2 User_settings table

In-game settings are stored here.


#	Name	Type	Collation	Attributes	Null	Default
1	user_id 	varchar(50)	utf8_hungarian_ci		No	None
2	resolution	varchar(10)	utf8_hungarian_ci		No	1920x1080
3	quality_index	int(11)			No	3
4	fullscreen	int(1)			No	1
5	music_volume	float			No	0.1
6	effect_volume	float			No	0.5

Figure 3: The structure of the user_settings table

The table contains 6 columns:

- User_id: The same string that was generated at signing up. This is the foreign key for the table. The two tables are joined using this column and the user table's primary key.
- Resolution: The resolution used in the game. Default is 1920x1080. If the user's display doesn't support the resolution saved in the database, the game will fall back to the currently used one.
- Quality_index: The index of the quality settings selected in the dropdown menu. (0: Very Low – 6: Ultra). Default is 3 (Medium).
- Fullscreen: A boolean value storing whether fullscreen should be used or not. Default is 1 (true).
- Music_volume: Float value between 0 and 1, storing the volume for the music in game. Default is 0.1.
- Effect_volume: Float value storing the volume for the effects. Default is 0.5.

user_id	resolution	quality_index	fullscreen	music_volume	effect_volume
d02fR03b8EqIPfE-YSDzjw	1920x1080	3	1	0.1	0.5

Figure 4: An example of the default values in the user_settings table

3. Main menu

3.1 Intro

The game starts with the intro scene. Here, the user can choose the game's language from a dropdown menu. The user can also mute/unmute the effect and music volumes.



Figure 5: The intro scene

The dropdown menu's content is read from .ini files in the Language directory of the game. Before the first frame update the game reads in the language files. The first line of each file, is the string that will be displayed in the dropdown. The rest of the file contains key-value pairs that will be stored in a Dictionary<string,string>. The keys in all language files have to match, because the main game reads in the texts from these dictionaries always asking for the value of one specific key. After all the lines have been read from the file, the dictionary will be added to a list, which contains all dictionaries.

```
private void Read_files()
{
    string[] files = Directory.GetFiles(Application.dataPath + "/Languages", "*.ini");
    foreach (string file in files)
    {
        Read_lang_file(file);
    }
}
```

Figure 6: The Read_files() function

```
private void Read_lang_file(string filepath)
{
    using (StreamReader reader = new StreamReader(filepath))
    {
        Dictionary<string, string> lang = new Dictionary<string, string>();
        options.Add(reader.ReadLine());
        while (!reader.EndOfStream)
        {
            string[] temp = reader.ReadLine().Split('=');
            lang.Add(temp[0], temp[1]);
        }
        languages.Add(lang);
    }
}
```

Figure 8: The Read_lang_files() function

(options is a List<string> which contains all options for the dropdown menu and languages is the list containing all dictionaries)



```
lang_eng.ini - Notepad
File Edit Format View Help
English
Choose_lang=Please choose a language!
Submit=Submit
Register=Sign up
Continue_offline=Continue offline
Email=E-mail
Username=Username
Password=Password
Login_success=Login successfull!
Login_incorrect=Incorrect username/password!
Connection_failed=Can't connect to server!
Welcome=Welcome
Play_game=Play game
Sign_in=Sign in
Sign_out=Sign out
Options=Options
Restart=Restart
Exit=Exit
Back=Back
Save=Save
Fullscreen=Fullscreen
Account_created=Account created successfully!
Account_exists=Account already exists!
```

Figure 7: The content of a language file

The index of the currently selected language is stored in an integer which is changed through a function called `Set_language_index()`. The function is connected to the dropdown menu and the function gets the currently selected index through an integer parameter every time the user changes the value of the dropdown.

```
public void Set_language_index(int index)
{
    current_lang = index;
    button_text.text = languages[current_lang]["Submit"];
    choose_text.text = languages[current_lang]["Choose_lang"];
}
```

*Figure 9: The Set_language_index function
(current_lang is the integer that stores the currently selected index,
the two texts are the main message for the user and the text of the submit button,
both are changed to the now selected language.)*

Other scripts can access the selected dictionary through a function, which accepts a string key as its parameter and returns the value of that key.

```
public string Return_language_string(string key)
{
    return languages[current_lang][key];
}
```

Figure 10: The Return_language_string() function

Unity can determine the language of the OS that is running the game, so I added a feature that, if the OS's language is Hungarian then Hungarian is selected by default in the dropdown menu.

```
private void Set_language_from_OS()
{
    if (Application.systemLanguage == SystemLanguage.Hungarian)
    {
        dropdown.value = options.IndexOf("Magyar");
    }
}
```

Figure 11: The Set_language_from_OS() function

If the user presses the submit button the main scene gets loaded in through a loading screen.

3.2 Login

After the scene has loaded, the login panel welcomes the user. From here, the user can choose to log in, sign up, continue offline or ask for a new password. The user can start the log in sequence if he clicks the submit button. The submit button is disabled by default and only gets enabled once the user has filled out both input fields. Once the user starts typing in the input field, a message appears giving some more information, on how to complete the field.



Figure 12: The login panel

If the user clicks the button or hits enter while the button is enabled, a coroutine will run. The coroutine will send the input fields' value to a specified URL with POST method. On that URL, a PHP file will take the data and check if an account exists in the database. If it can't find an account, it will send back 0 meaning the account couldn't be found. If the account was found, it will send back the account's ID, username, resolution, quality_index, fullscreen, music_volume and effect_volume as a string separated by a "/". If it can't connect to the database, it will send back -1 meaning that the connection failed. The game will show a message / log in according to these codes. If the log in was successful, the game will take the string and split it up by the character "/" and convert the values to the right datatypes, then apply the settings to the game. The settings are also saved in a class.


```
IEnumerator Log_in()
{
    login_feedback.color = Color.white;
    login_feedback.text = langs.Return_language_string("Connecting");
    submit.interactable = false;
    WWWForm form = new WWWForm();
    form.AddField("username", username.text);
    form.AddField("password", password.text);
    UnityWebRequest req = UnityWebRequest.Post("tictactoe3d.ddns.net/login/login.php", form);
    req.certificateHandler = new CertificateHandler_zsivany();
    req.timeout = 10;
    yield return req.SendWebRequest();
    try
    {
        if (req.downloadHandler.text == "-1" || req.downloadHandler.text == "")
        {
            throw new Exception($"{langs.Return_language_string("Connection_failed")}");
        }
        else if (req.downloadHandler.text == "0")
        {
            Debug.Log($"[{Time.frameCount}] Can't find user {username.text}");
            Log_in_failed();
            throw new Exception($"{langs.Return_language_string("Login_incorrect")}");
        }
        else
        {
            StartCoroutine(Log_in_successfull(req.downloadHandler.text));
        }
    }
    catch (Exception ex)
    {
        login_feedback.color = Color.red;
        login_feedback.text = $"{ex.Message}";
        Debug.Log($"[{Time.frameCount}] {login_feedback.text}");
    }
    req.Dispose();
}
```

Figure 13: The coroutine that initiates the login sequence

```

try {
    require("../connect.php");
} catch (\Throwable $th) {
    echo(-1);
    exit();
}

$username = strip_tags(trim($_POST['username']));
$password = strip_tags(trim($_POST['password']));
$sql = "SELECT * FROM users INNER JOIN user_settings on users.id = user_settings.user_id WHERE
users.username = '$username' AND active = '1'";
$result = mysqli_query($conn,$sql);
if ($result -> num_rows == 0) {
    echo(0);
    exit;
}
$row = $result -> fetch_assoc();
if (password_verify($password,$row['password'])) {
    $data = $row['id']."/". $row['username']."/". $row['resolution']."/". $row['quality_index']."/".
    $row['fullscreen']."/". $row['music_volume']."/". $row['effect_volume'];
    echo($data);
    exit();
}
echo(0);
exit;

```

Figure 14: The PHP file that responds to the web request

3.3 Sign up

The user can create a brand-new account using the sign up panel. The panel can be activated by clicking the sign up button on the login panel. The form on the panel works the same as the one on the login panel, meaning, the submit button only gets activated once the form has been filled out correctly. Once the user clicks the submit button, a web request will be sent to a specific URL where a PHP file will check if an account exists with the given credentials, if not, it will create a new inactivated account in the database. In order to log into the account, the user must activate it through a link (that will be accessible through clicking a button), that will be sent in an email to the email address that was used in the sign up form. The email's language depends on the game's language. If the game was started in Hungarian, the email will be Hungarian as well, otherwise the email will be English. The link in the email will take the user to a PHP file that will try to activate the account, based on an ID that is given to the file through the link. If it can't find an account with that ID (the account has been deleted etc.) it will give a message to the user,

saying that the account cannot be found. If an account is found, it will activate it and show a message to let the user know that they can now log in to the account.



Figure 15: The activation email in English

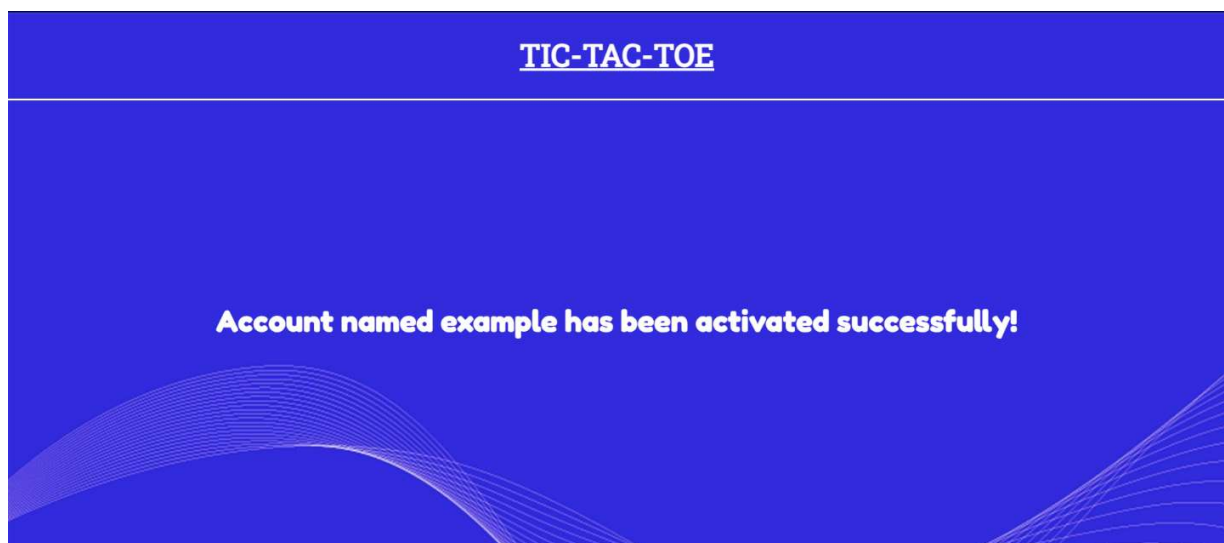


Figure 136: The webpage that activates the user's account

3.4 Change password

In case the user forgets the password to their account, they can ask for a new one, from the change password panel, which is accessible from the login menu. The user only needs to provide the email address linked to the account and an email containing a link will be sent to that address. The link will take the user to a webpage where they can change their password.

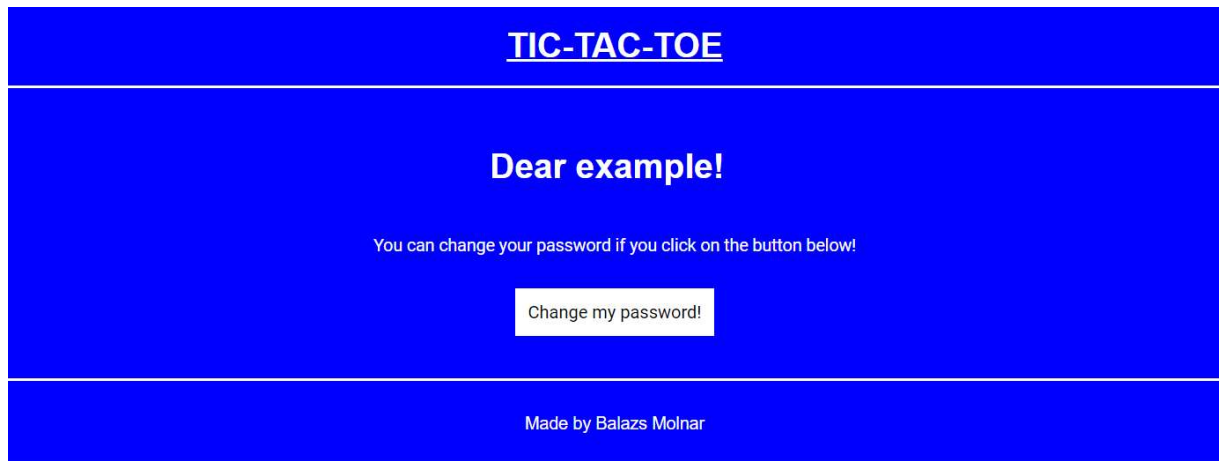


Figure 17: The email that contains the link to change the password

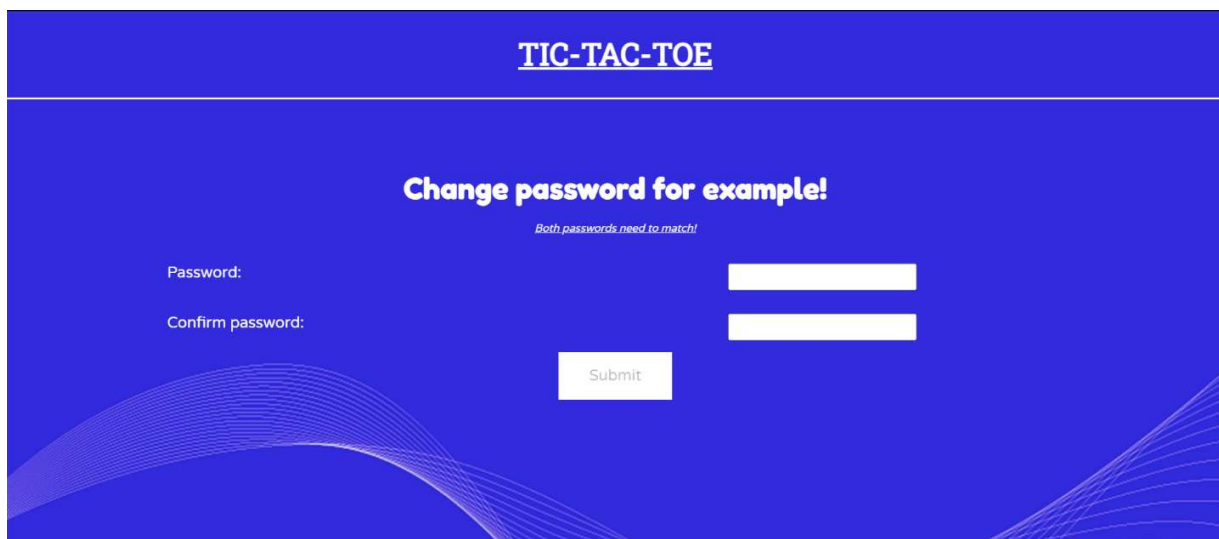
The image shows a blue webpage for changing a password. At the top, the text 'TIC-TAC-TOE' is centered in white, underlined. Below this, the text 'Change password for example!' is centered in white. Below this text is the text 'Both passwords need to match!' in a smaller white font. Below this text are two white input fields for 'Password:' and 'Confirm password:'. Below the input fields is a white button with the text 'Submit' in black. The background of the webpage features a blue gradient with white wavy lines at the bottom.

Figure 148: The webpage where the user can change their password
(The submit button only gets activated, once the user correctly fills out the form)

3.5 Options menu

Once the user decides if they want to log in or continue offline, the main menu will appear, where they can access the options menu. In there, they can change the look of the game (resolution, quality etc.) and they can change the volume of the sound sources. If the user



Figure 19: Options menu without a logged in account

```
private void Resolution_Dropdown_Set()
{
    resolution_dropdown.ClearOptions();
    resolutions = Screen.resolutions;
    int current_resolution = 0;
    for (int i = 0; i < resolutions.Length; i++)
    {
        string option = $"{resolutions[i].width} x {resolutions[i].height}";
        res.Add(option);
        if (resolutions[i].width == Screen.width && resolutions[i].height == Screen.height)
        {
            current_resolution = i;
        }
    }
    resolution_dropdown.AddOptions(res);
    resolution_dropdown.value = current_resolution;
    resolution_dropdown.RefreshShownValue();
}
```

Figure 20: The `Resolution_Dropdown_Set()` function, which fills up the dropdown and sets the current resolution to the currently used one.

(`resolution_dropdown` is the in-game dropdown menu, `resolutions` is a `Resolution[]` that contains all resolutions that are supported by the display, `res` is a `string[]` that contains the resolutions in string format)

changes the value of a setting, it's value will also be changed in the class created at login. The resolution dropdown only contains resolutions that are supported by the user's display. If the user is logged in, the save button will be enabled, so the settings from the class could be saved into the database.

3.6 Choose play mode

The user can choose between 1 player and 2 player mode. In 1 player mode, the CPU will calculate player 2's move so the user can play alone, against a computer.

4. Gameplay

The playable area is based on a 10x10 two dimensional array. This array holds Item classes, which consists of :

- **GameObject** named placeholder: The user clicks this **GameObject** to play the game.
- **Integer** named X: The X coordinate in the array.
- **Integer** named Y: The Y coordinate in the array.
- **Integer** named value: The value of this element (0 – Empty;1 – Player 1; 2 – Player 2)

```
public class Item
{
    1 reference
    public GameObject placeholder { get; set; }
    33 references
    public int value { get; set; }
    70 references
    public int X { get; set; }
    68 references
    public int Y { get; set; }
    14 references
    public Item(GameObject obj, int value = 0, int x = 0, int y = 0)
    {
        placeholder = obj;
        this.value = value;
        X = x;
        Y = y;
    }
    0 references
    public override bool Equals(object obj)
    {
        if (obj == null) return false;
        if (!(obj is Item)) return false;
        return this.X == ((Item)obj).X && this.Y == ((Item)obj).Y;
    }
    2 references
    public override int GetHashCode()
    {
        return this.GetHashCode();
    }
}
```

Figure 21: The Item class

(The Equals function is overridden, so two Item classes can be compared correctly)

After the user clicks one of the GameObjects, a function will run, that will fill up the correct element in the array and spawn the correct animation into the play area. If the step counter is over 8, then the game will start checking for a win. If no one has won, then a player switch will occur, so the next player can take their turn.

The `Grid_Value_Set()` function is the one, that places the correct Items and animations. It takes in the position of the clicked GameObject, and places an Item there if that Item's value is 0 (meaning no one has placed an Item there). It also spawns the animation, checks for win if needed and initiates the CPU's turn in one player mode.

```
public void Grid_Value_Set(int x, int y)
{
    if (grid[x, y].value == 0 && current_game_state == Game_state.Not_Won)
    {
        grid[x, y].X = x;
        grid[x, y].Y = y;
        grid[x, y].value = current_Player;
        placed_Items[current_Player - 1].Add(grid[x, y]);
        step_counter++;
        Transform parent = current_Player == 1 ? X_parent.transform : O_parent.transform;
        Debug.Log($"[Frame:{Time.frameCount}] Placed item for Player {current_Player} at [{x},{y}]");
        GameObject new_item = Instantiate(Item_animations[current_Player - 1], new Vector3(x - .5f + 1, y - .5f, -.5f),
        Quaternion.identity, parent);
        audio_manager.Play_drawing();
        string item_name = current_Player == 1 ? $"X_{placed_Items[0].Count}" : $"O_{placed_Items[1].Count}";
        new_item.name = item_name;
        Check_For_Win();
        if (one_player && current_Player == 1 && current_game_state == Game_state.Not_Won)
        {
            CPU_turn();
        }
    }
}
```

Figure 22: The `Grid_Value_Set()` function which places the correct Item
(`grid` is the `Item[,]`, `placed_Items` is a `List<Item>[]` which holds all the placed Items in the array in two different lists, `new_item` is the animation that will be spawned, `audio_manager` is the script that manages the sounds for the game.)

4.1 Check for win

If the step counter is over 8, then the game will start checking for a win after every item placement. The function will only run until it can decide if there has been a win or not. The function uses other functions to determine a win.

```
public void Check_For_Win()
{
    if (step_counter > 8)
    {
        if (current_game_state == Game_state.Not_Won)
        {
            int counter = 0;
            while (Item_didnt_win(counter))
            {
                counter++;
            }
            if (counter < placed_Items[current_Player - 1].Count)
            {
                playerscores[current_Player - 1]++;
                Game_Won();
            }
        }
        if (step_counter == width * height) Game_draw();
    }
    if (current_game_state == Game_state.Not_Won && !one_player) Change_player();
}
```

Figure 23: The Check_For_Win() function

```
private bool Item_didnt_win(int counter)
{
    return counter < placed_Items[current_Player - 1].Count && (!Horizontal_Check(placed_Items[current_Player - 1][counter]) && !
    Vertical_Check(placed_Items[current_Player - 1][counter]) && !Diagonal_L_R_Check(placed_Items[current_Player - 1][counter]) && !
    Diagonal_R_L_Check(placed_Items[current_Player - 1][counter]));
}
```

Figure 24: The Item_didnt_win() function

The Horizontal_Check() is the function that is used to check for a horizontal win. It returns true, if there are 5 of the same values next to each other and puts the Items into winning_items List, that will be used later for the camera animations. It automatically returns false if there is not enough space for 5 items. (All other “win checki ng” functions are similar to this)


```

private bool Horizontal_Check(Item item)
{
    if (item.X + 4 < width)
    {
        int count = 0;
        int x_Counter = item.X;
        while (count < 5 && grid[x_Counter, item.Y].value == current_Player)
        {
            winning_items.Add(new Item(item_placeholder, current_Player, x_Counter, item.Y));
            x_Counter++;
            count++;
        }
        if (count == 5)
        {
            current_game_state = Game_state.Won_horizontal;
        }
        else
        {
            winning_items.Clear();
        }
        return count == 5;
    }
    else return false;
}

```

Figure 25: The Horizontal_Check() function.

4.2 One player mode

The user can play against the CPU. In this game mode, the CPU automatically calculates the best steps for both players and stores them in two different Lists. It then decides its step based on the lengths of these Lists. If the Player 1's list has fewer elements (meaning that Player 1 needs less steps to win) the CPU will try to block Player 1. In every other scenario, it will try to win.

The Calculate_min_steps() function is responsible for doing this. It returns The List<Item> that will contain the steps needed to win for a player. It takes in two parameters, the List<Item> that contains all the Items placed by a player, and an int value that holds the player's value. It temporarily fills up the temp named List<Item>, that will be returned containing the steps needed to win. It iterates through all the Items placed by the player, and calls 4 different functions that are used to calculate the needed steps in 4 different directions.

```

private List<Item> Calculate_min_steps(List<Item> items, int value)
{
    List<Item> temp = new List<Item>();
    for (int i = 0; i < 5; i++)
    {
        temp.Add(new Item(item_placeholder));
    }
    foreach (Item item in items)
    {
        Debug.Log($"Current item: [{item.X},{item.Y}]");
        Calculate_horizontal_steps(item, value, ref temp);
        Calculate_vertical_steps(item, value, ref temp);
        Calculate_diagonal_l_steps(item, value, ref temp);
        Calculate_diagonal_r_steps(item, value, ref temp);
    }
    return temp;
}

```

Figure 26: The Calculate_min_steps() function

One example of the 4 calculation functions is the Calculate_horizontal_steps(). It takes in 3 parameters: An item, an int value, and a referenced List<Item>, where we store the minimum steps for the player. A List<Item> named temp will be created, where we store the minimum steps needed to win for the Item in the parameter. If it encounters the same value during the search, then it will add it to a new List<Item> named same. At the end, the same list's elements will be removed from the temp list. If the temp's count is less than the main's count (meaning that the currently calculated Item needs less steps to win) then we will copy the steps from temp into main. (All other calculations (vertical and diagonal) are similar to this)

```

private void Calculate_horizontal_steps(Item item, int value, ref List<Item> main)
{
    int x = item.X;
    int y = item.Y;
    if (x + 4 < width)
    {
        if (grid[x + 1, y].value != 3 - value)
        {
            x++;
            int counter = 1;
            List<Item> temp = new List<Item>();
            List<Item> same = new List<Item>();
            while (counter < 5 && grid[x, y].value != 3 - value)
            {
                if (grid[x, y].value == value) same.Add(grid[x, y]);
                temp.Add(new Item(item_placeholder, value, x, y));
                counter++;
                x++;
            }
            if (temp.Count >= 4)
            {
                foreach (Item s in same)
                {
                    temp.Remove(s);
                }
                Debug_steps_to_console("Horizontal", temp, item);
                if (temp.Count < main.Count)
                {
                    main = temp;
                }
            }
        }
    }
}

```

Figure 27: The Calculate_horizontal_steps() function.

After we calculated the steps for both players, the CPU will decide its step based on the count of the two Lists.

```

if (cpu_steps.Count == 1 || player_steps.Count == 1)
{
    if (cpu_steps.Count == 1) Grid_Value_Set(cpu_steps[0].X, cpu_steps[0].Y);
    else Grid_Value_Set(player_steps[0].X, player_steps[0].Y);
}
else if (player_steps.Count < cpu_steps.Count) Grid_Value_Set(player_steps[0].X, player_steps[0].Y);
else Grid_Value_Set(cpu_steps[0].X, cpu_steps[0].Y);

```

Figure 28: At the end, the CPU will decide its step based on the count of the Lists

5. Camera animations

Camera animations are handled by coroutines that are stored in the script called Camera_animations.

The Camera_win_anim coroutine will move the camera along the winning Items. It gets the Items position through its parameter. Along the animation it will move the effect bars (the two black bars on the top and bottom of the screen that are used for an epic effect) to their location, it will slow down and speed up the music and will start spawning in the winning panel.

```
public IEnumerator Camera_win_anim(Vector3 First_Target, Vector3 Second_Target, Vector3 Last_Target, Vector3 Panel_Target, int[] scores)
{
    Debug.Log($"[Frame:{Time.frameCount}] Moving cam to first target");
    while (Vector3.Distance(main_camera.transform.position, First_Target) > 1f)
    {
        main_camera.transform.position = Vector3.Slerp(main_camera.transform.position, First_Target, Cam_Movement_speed * Time.deltaTime);
        yield return null;
    }
    Debug.Log($"[Frame:{Time.frameCount}] Moving cam to second target");
    StartCoroutine(audio_manager.Slow_down_music());
    audio_manager.Play_slowmo();
    StartCoroutine(Move_effect_bars(true, true, false));
    Time.timeScale = .7f;
    Draw_line draw_line = GameObject.Find("GridRunner").GetComponent<Draw_line>();
    StartCoroutine(draw_line.Draw_line());
    while (Vector3.Distance(main_camera.transform.position, Second_Target) > 1f)
    {
        main_camera.transform.position = Vector3.MoveTowards(main_camera.transform.position, Second_Target, Cam_Movement_speed * 1.2f * Time.deltaTime);
        yield return null;
    }
    StartCoroutine(Move_effect_bars(false, false, true));
    StartCoroutine(audio_manager.Speed_up_music());
    Time.timeScale = 1;
    canvas_script.Panel_Set(Panel_Target, scores);
    Debug.Log($"[Frame:{Time.frameCount}] Moving cam to third target");
    while (Vector3.Distance(main_camera.transform.position, Last_Target) > 1f)
    {
        main_camera.transform.position = Vector3.Slerp(main_camera.transform.position, Last_Target, Cam_Movement_speed * Time.deltaTime);
        yield return null;
    }
    Debug.Log($"[Frame:{Time.frameCount}] Cam movement ended");
}
```

Figure 29: The Camera_win_anim coroutine

The Camera_win_rotate will rotate the camera, while it moves along the winning Items. It gets the correct Quaternions through its parameter.

```
public IEnumerator Camera_win_rotate(Quaternion desired_Rotation_Q, Quaternion default_Rotation_Pos, Vector3 Second_Target, Vector3 End_Target)
{
    Debug.Log($"[Frame:{Time.frameCount}] First rotation started");
    while (Quaternion.Angle(main_camera.transform.rotation, desired_Rotation_Q) > 1 || Vector3.Distance(main_camera.transform.position, Second_Target) > 1f)
    {
        main_camera.transform.rotation = Quaternion.Slerp(main_camera.transform.rotation, desired_Rotation_Q, Cam_Rotation_speed * Time.deltaTime);
        yield return null;
    }
    Debug.Log($"[Frame:{Time.frameCount}] Second rotation started");
    while (Quaternion.Angle(main_camera.transform.rotation, default_Rotation_Pos) > .5f || Vector3.Distance(main_camera.transform.position, End_Target) > 1f)
    {
        main_camera.transform.rotation = Quaternion.Slerp(main_camera.transform.rotation, default_Rotation_Pos, Cam_Rotation_speed * 2 * Time.deltaTime);
        yield return null;
    }
    Debug.Log($"[Frame:{Time.frameCount}] Rotation ended");
}
```

Figure 30: The Camera_win_rotate coroutine