Documentation

Inhoud

Posture check

This system requires the following scripts:

- PositionChecker
- PositionCheckerEasy
- PositionCheckerMedium
- PositionCheckerHard

Scriptable objects (SO):

- PositionCheckerSO
- PosEasySO
- PosMediumSO
- PosHardSO

This system is made to make sure the user have/must hold a certain position in order to start an exercise. This can be used on every exercise if it is needed.

If it is needed you can experiment the many positions using float variables (PosEasySO, PosMediumSO, PosHardSO) which will deduct from the Vector3 Y-axis (currently the only axis that is needed) to create two lines. These are for the minimal and maximum lines (On Camera Offset -> Feedback). As a user you need to make sure you are in between these lines to start the exercise. Getting out of these lines should make the game/exercise pause.

The user also sees another line on top of the screen. It has two colours. Red and green. Red signifies the user is not in position while the green colour signifies the user is in position (in between the lines).

The lines are based on your current headset position after calibrating (Which stores the current Vector3 in your local machine. For example. If you're 1,70 meters which is simplified 1,70 Y-axis. The float values (PosEasySO, PosMediumSO, PosHardSO) will subtract from that Y-axis. Meaning if the MinY on PosMediumSO is 0,3 while the MaxY is 0,6. The lines will be set on the following values: MinimalLine -> 1,40 and MaximumLine -> 1,10.

If the system is not needed. The script will skip the system and go straight to the game/exercise chosen.

PositionChecker

```
27 references
public class PositionChecker
{
    public float HoldTime;
    public float MinX, MaxX, MinY, MaxY, MinZ, MaxZ;

    4 references
    public PositionChecker(float holdTime, float minX, float maxX, float minY, float maxY, float minZ, float maxZ)
    {
        HoldTime = holdTime;
        MinX = minX;
        MaxX = maxX;
        MinY = minY;
        MaxY = maxY;
        MinZ = minZ;
        MaxZ = maxZ;
    }
```

PositionCheckerSO

```
public abstract class PositionCheckerSO : ScriptableObject
{
    public float HoldTime;
    public float MinX, MaxX, MinY, MaxY, MinZ, MaxZ;

    4 references
    public virtual PositionChecker SetPosition()
    {
        return new PositionChecker(HoldTime, MinX, MaxX, MinY, MaxY, MinZ, MaxZ);
    }
}
```

This is the base for each and every difficulty that’ll be used in the following picture.

PositionChecker Easy/Medium/Hard

```
[CreateAssetMenu(fileName = "PositionCheckerHard", menuName = "Scriptable Objects/PositionCheckerHard")]
2 references
public class PositionCheckerHard : PositionChecker
{
    public float holdTime;
    1 reference
    public float minX { get; private set; }
    1 reference
    public float maxX { get; private set; }
    1 reference
    public float minY { get; private set; }
    1 reference
    public float maxY { get; private set; }
    1 reference
    public float minZ { get; private set; }
    1 reference
    public float maxZ { get; private set; }
    1 reference
    public PositionCheckerHard(float holdTime, float minX, float maxX, float minY, float maxY, float minZ, float maxZ) : base(holdTime, minX, maxX, minY, maxY, minZ, maxZ)
    {
        string chosenDifficulty = DifficultyManager.Instance.SelectedDifficulty.ToString();
        if (chosenDifficulty == "Hard")
        {
            this.holdTime = holdTime;
            this.minX = minX;
            this.maxX = maxX;
            this.minY = minY;
            this.maxY = maxY;
            this.minZ = minZ;
            this.maxZ = maxZ;
        }
    }
}
```

PosEasySO (Medium + Hard)

```
[CreateAssetMenu(fileName = "PosHardSO", menuName = "PositionChecker/PosHardSO")]
Unity Script | 0 references
public class PosHardSO : PositionCheckerSO
{
    2 references
    public override PositionChecker SetPosition()
    {
        return new PositionCheckerHard(HoldTime, MinX, MaxX, MinY, MaxY, MinZ, MaxZ);
    }
}
```

These six scripts makes sure to create the SO which you can alter for each and every exercise. In which you can right click in the project (Create -> all the way to the bottom PositionChecker)

If the values are set (Like the picture on the right).
the values are send to PositionCheckerMedium which base it from PositionChecker and set it.

GenericExerciseSO has everything needed for one exercise but we'll focus on two parts on the script.

PositionNeeded is there to make sure if the position feedback (the lines) are needed. If the checkbox is ticked off the system will skip everything regarding the position checker system. If the checkbox is ticked on. It'll check the difficulty afterwards. If the exercise on a certain difficulty doesn't need the system it'll also skip the entire system. For example Easy difficulty doesn't need the system.

A double check system to make sure the chosen exercise only triggers the positionChecker scripts to force an user to hold a certain position if it is truly needed.

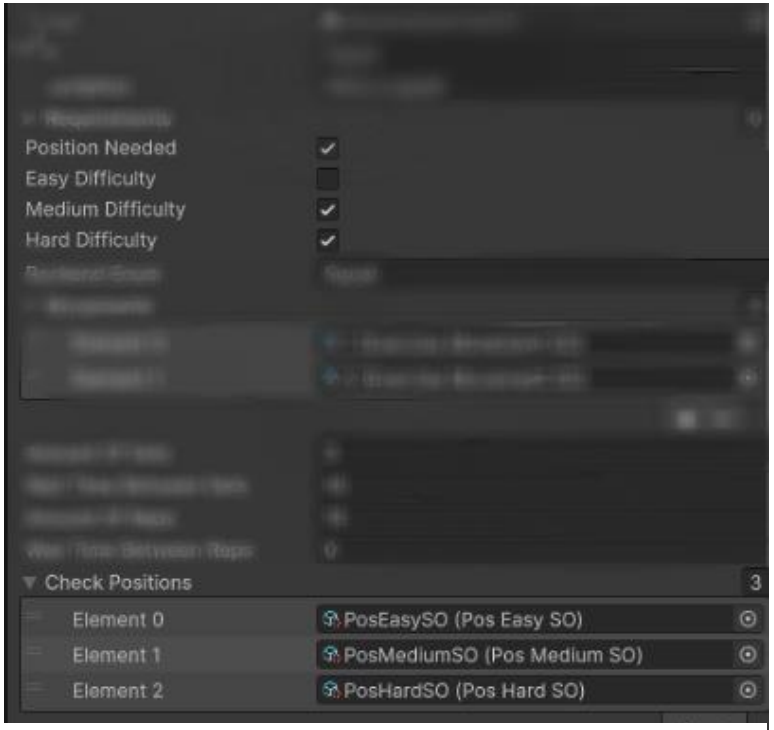

Figure 1: The data needed for the Posture System

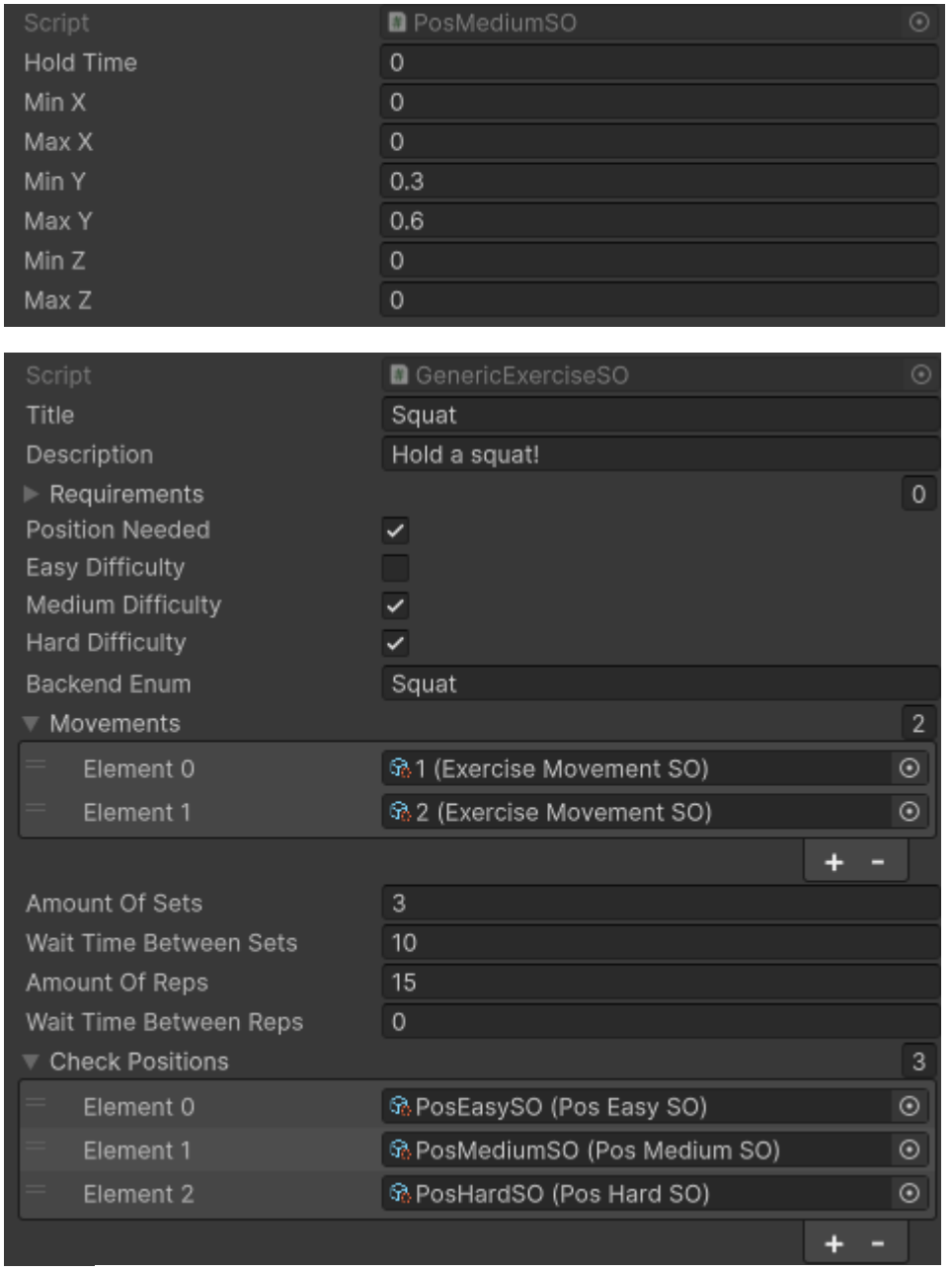

Figure 2: Whole Exercise SO and data you can add

Next the Check Positions List. If the exercise doesn't need the system (PositionNeeded is off) you can skip this.

If it is on. You'll need to create the three PosSO of the three exercises and assign it in the list accordingly.

```
if (PosNeeded == true)
{
    int Count = chosenDifficulty == "Easy" ? 0 : chosenDifficulty == "Medium" ? 1 : chosenDifficulty == "Hard" ? 2 : 0;
    if (Count >= 0 && Count < Checkers.Count)
    {
        refs.currentPosSO = Checkers[Count];
    }
}
```

The code above will check the chosenDifficulty variable. If the difficulty is easy. The int value will be 0 which corresponds to the list element 0. Which will also be assigned to the references class. A static class where you can use these values everywhere when needed.

```
[Header("Position Specific References for Exercises")]
public bool NeedsPosition;
public bool EasyDifficulty;
public bool MediumDifficulty;
public bool HardDifficulty;
public PositionChecker currentPosSO;

[Header("Feedback for exercises that needs specific positions")]
public GameObject FeedbackLine;
public GameObject RenderLineMinimal;
public GameObject RenderLineMaximal;
```

Figure 3: Variables used in references class

If the position is needed. We'll take the medium difficulty as an example. Both PositionNeeded and Medium Difficulty booleans are ticked on making it true to start the exercise with feedback on. The exercise is "Squat". Meaning it'll only take the PosMediumSO from the list on the squat exercise SO. Which has the values set on both MinY and MaxY. Those values will be set in the PositionCheckerMedium which overrides the PositionChecker. Meaning the PositionChecker now has the stored values from the PosMediumSO for the exercise to use.

When starting the exercise. The if statement will check the chosenDifficulty and set the currentPosSO. Then the exercise starts as normal.

```
// Checks if the current exercise difficulty needs position checker
bool setPositionNeeded = setPosition == true ? true : false;
// Checks currently chosen difficulty to see if the exercise/minigame needs position check
bool checkDifficulty = chosenDifficulty == "Easy" ? NormalExerciseReferences.Instance.EasyDifficulty :
                       chosenDifficulty == "Medium" ? NormalExerciseReferences.Instance.MediumDifficulty :
                       chosenDifficulty == "Hard" ? NormalExerciseReferences.Instance.HardDifficulty : false;

PositionChecker currentChecker = NormalExerciseReferences.Instance.currentPosSO;
Debug.Log("Current Position Checker: " + currentChecker);

Vector3 minBound = new Vector3(currentChecker.MinX, currentChecker.MinY, currentChecker.MinZ);
Vector3 maxBound = new Vector3(currentChecker.MaxX, currentChecker.MaxY, currentChecker.MaxZ);

minimalPos = headsetPos.y - minBound.y;
maximalPos = headsetPos.y - maxBound.y;

// Checks to see if the feedback for position check needs to be turned on.
bool turnOnFeedback = setPositionNeeded == true ? checkDifficulty == true ? true : false : false;



//Debug.Log("FeedbackCube is turned on: " + turnOnFeedback);
NormalExerciseReferences.Instance.FeedbackLine.SetActive(turnOnFeedback);
NormalExerciseReferences.Instance.RenderLineMinimal.SetActive(turnOnFeedback);
NormalExerciseReferences.Instance.RenderLineMaximal.SetActive(turnOnFeedback);
```

Figure 4: HeadswayMovement class code showcasing the feedback lines turn on script and position setter for the lines (Gitlab)

The following code show cases how the feedback lines get turned on. It checks first the positionNeeded Boolean and then the difficulty boolean. If both are true the turnOnFeedback will be turned on.

The feedback Lines will then be set on the given positions. The script above shows part of the code. Where it picks up the float variables from the PosMediumSO and create two Vector3 variables. minBound and maxBound. Which will subtract from the headsetPos.

The amountPositions are data of the three positions of headset and both controllers after calibration.

Said data (Vector3) will be stored in each variable you see on the right picture. We'll focus on the HeadsetPos.

The HeadsetPos is the calibrated headset position. Which is for this example 1,70 on the Y-axis.

```
Vector3[] amountPositions = LoadHeight.loadData.ToArray();

if (amountPositions.Length >= 0) {
    headsetPos = amountPositions[0];
    rightArmPos = amountPositions[1];
    leftArmPos = amountPositions[2];
}



currentHeadsetPos = ExerciseManager.Instance.Headset.transform.position;
```

Figure 5: setting position data from array picked out from the json file

minBound and maxBound now has the Vector3 values from the PosMediumSO. The headsetPos will now be subtracted using both min and max bound.

```
Vector3 defaultZ = new Vector3(0, 0, 1);

NormalExerciseReferences.Instance.RenderLineMinimal.transform.position = (headsetPos + defaultZ) - minBound;
NormalExerciseReferences.Instance.RenderLineMaximal.transform.position = (headsetPos + defaultZ) - maxBound;
//Add another feedback stuff using the turnOnFeedback

if (InBoundsY()) {
    elapsedWhileHolding += Time.deltaTime;
    NormalExerciseReferences.Instance.HoldMovementText.transform.parent.gameObject.SetActive(true);
    NormalExerciseReferences.Instance.HoldMovementText.text = (holdTime - elapsedWhileHolding).ToString("0.0") + "s";
    NormalExerciseReferences.Instance.InformationObject.SetActive(false);

    RenderCube(Color.green);
} else {
    // if exercise has been finished, player cannot be out of bounds

    Reset();
    elapsedWhileHolding = 0;
    NormalExerciseReferences.Instance.HoldMovementText.transform.parent.gameObject.SetActive(false);
    NormalExerciseReferences.Instance.InformationObject.SetActive(true);
    NormalExerciseReferences.Instance.InformationText.text = "Not in bounds!";
}
```

Figure 6: Showcases line positions and what happens when your in or out of position

Then the lines (RenderLineMinimal and Maximal) will be placed using the transform.position. HeadsetPos + defaultZ is for the offset. So it spawns in front of you instead of your current position. Then the minBound and maxBound will subtract the Y-axis and the lines are set.

Then the if(InBoundsY()) will take into effect.
(Figure 6)

That if statement will only check the InBoundsY function. That function will compare the minimalPos and maximalPos with the currentHeadsetPos which is stuck on the Main camera (users headset position). Since the person in question is 1,70 meters (1,70 Y-axis).

He/she will have to lower the body (squat) to hit the minimal of 1,40 and hold the position (in reps).

If you go below the 1,10 (maximalPos) the exercise will pause until you go back into position.

If later in the development. It is possible to combine InBounds to make sure you are in position in many kinds of positions.

```
private bool InBoundsX() {
    Vector3 headPos = ExerciseManager.Instance.Headset.transform.position;
    if (headPos.x > positiveX_Minimum && headPos.x < positiveX_Maximum ||
        headPos.x < negativeX_Minimum && headPos.x > negativeX_Maximum) {
        return true;
    } else {
        return false;
    }
}
private bool InBoundsY() {
    Debug.Log(headsetPos.y);
    Debug.Log(currentHeadsetPos.y + " BHEHIFBEHBSEHGBSHGSHEGHSBGHSBEGBSGEJ");
    Debug.Log(minimalPos);
    Debug.Log(maximalPos);
    if (currentHeadsetPos.y < minimalPos  && currentHeadsetPos.y > maximalPos) {
        return true;
    } else {
        return false;
    }
}
private bool InBoundsZ() {
    Vector3 headPos = ExerciseManager.Instance.Headset.transform.position;
    if (headPos.z > positiveZ_Minimum && headPos.z < positiveZ_Maximum ||
        headPos.z < negativeZ_Minimum && headPos.z > negativeZ_Maximum) {
        return true;
    } else {
        return false;
    }
}

private bool DefaultBound() {
    Vector3 headPos = ExerciseManager.Instance.Headset.transform.position;
    if (headPos.y > restPosition) {
        return true;
    } else {
        return false;
    }
}
```

Figure 7: compare current camera position with calibrated headset position and if the current camera position is in between the minimal and maximal line

Calibration

Figure 8: part of the dashboard menu highlighting the calibration section

Calibration is needed to make sure each and every person can play/exercise accordingly without the worry about height. This system makes sure the height of the current position is calibrated on three positions. Headset, right and left controller.

The user starts on the main menu on the dash screen. On the dash screen is has a big calibration button. Pressing this will put you into a new panel for the next 5 seconds. When the timer hits 0 and says it is done it'll return to the main menu. With your three positions saved in a json file on your local machine.

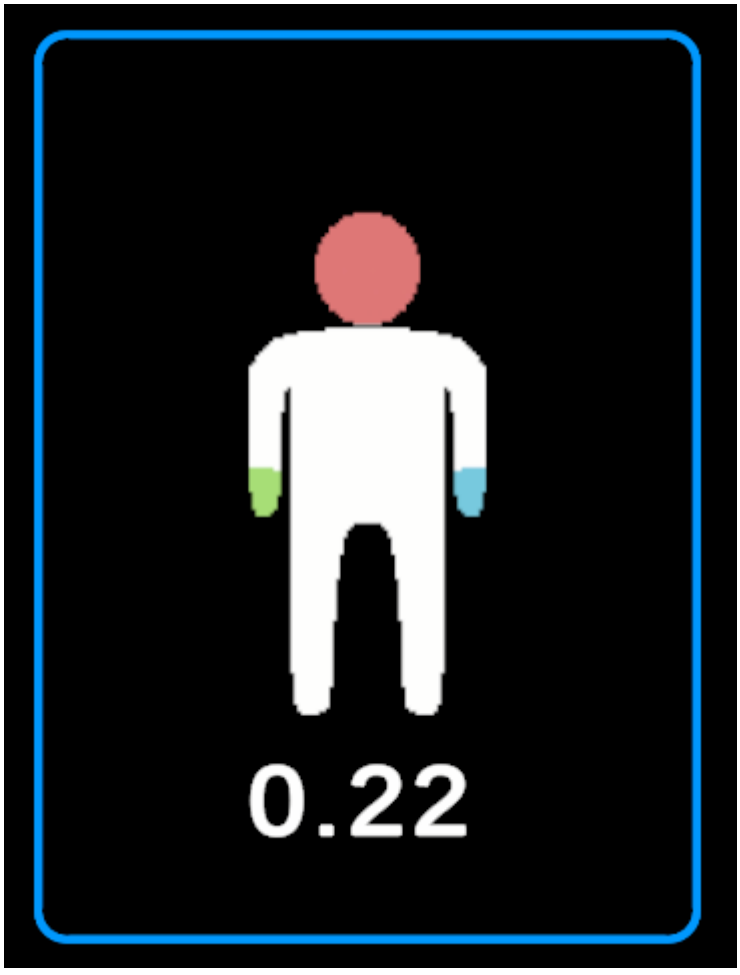

Figure 9: Calibration Panel

We'll focus on the following scripts:

- Calibrate
- LoadHeight

When you press the calibrate button. Unity onClick function starts the calibration process

When the Booleans is turned on, the Update if statement will now go towards calibrate user

The timer will subtract using a deltaTime and when it hits 0 does the actual calibration part.

```
0 references
public void TurnOn()
{
    calibrateOn = true;
}
```

```
Unity Message | 0 references
void Update()
{
    if (calibrateOn)
    {
        CalibrateUser();
    }
}

1 reference
public void CalibrateUser()
{
    calibrateTimer -= Time.deltaTime;
    timerText.text = calibrateTimer.ToString("F2");

    if (calibrateTimer < 0)
    {
        timerText.text = "Done!";
        calibrateOn = false;
        StartCalibration();
        StartCoroutine(Calibrated());
    }
```

We'll start with startCalibration. It'll first go through each three calibration parts to set the current MainCamera, rightArm and leftArm position and store it in a variable. Then it add a checker++. Since all three are calibrated the checker will now say 3. In which we'll go to the next stage. We'll clear the current addCalibration list and save the new local calibration and calculate the new arm distance from headset to each controller. Which then goes to a function to save the new calibration in your local machine.

```
1 reference
public void StartCalibration()
{
    HeadsetCalibration();
    RightArmCalibration();
    LeftArmCalibration();

    if (checker == 3)
    {
        addCalibration.Clear();
        SaveLocalCalibration(calibratedHeadsetPosition, calibratedRightArmPosition, calibratedLeftArmPosition);
        CalculateHeadArmDistance(calibratedHeadsetPosition, calibratedRightArmPosition, calibratedLeftArmPosition);
        loadingData.LoadHeightData();
        checker = 0;
    }
}
```

```
1 reference
public void HeadsetCalibration()
{
    Vector3 headHeight = mainCamera.transform.position;
    calibratedHeadsetPosition = headHeight;
    checker++;
}

1 reference
public void RightArmCalibration()
{
    Vector3 rightArmHeight = rightArm.transform.position;
    calibratedRightArmPosition = rightArmHeight;
    checker++;
}

1 reference
public void LeftArmCalibration()
{
    Vector3 leftArmHeight = leftArm.transform.position;
    calibratedLeftArmPosition = leftArmHeight;
    checker++;
}

public float rightArmLength;
public float leftArmLength;
1 reference
public void CalculateHeadArmDistance(Vector3 headset, Vector3 rightArm, Vector3 leftArm)
{
    rightArmLength = Vector3.Distance(headset, rightArm);
    leftArmLength = Vector3.Distance(headset, leftArm);
}
```

```
1 reference
IEnumerator Calibrated()
{
    yield return new WaitForSeconds(1f);

    calibrateTimer = calibrateTime;

    mainMenu.SetActive(true);
    calibrationMenu.SetActive(false);
}
```

While the calibration is happening the coroutine will start aswell. After 1 second has passed the calibration panel will vanish and go back to the main menu screen and reset the calibrationTimer. So the moment the calibrationTimer hits 0. The next second will calibrate and return to the menu.

Then the newly calibrated data will be stored and saved on your local machine.

And when logging in. It'll check the file on your localmachine and store it in a list. Then it'll remove all Vector3 elements and split the values and parse it as a new float to be stored in a new vector3 for each calibrated position (headset, rightarm and leftarm).

```
public List<string> addCalibration = new List<string>();

1 reference
public void SaveLocalCalibration(Vector3 headset, Vector3 rightArm, Vector3 leftArm)
{
    string path = Application.persistentDataPath + "/UserHeightList.json";
    addCalibration.Add(headset.ToString());
    addCalibration.Add(rightArm.ToString());
    addCalibration.Add(leftArm.ToString());

    HeightList list = new HeightList { heightList = addCalibration };
    string newJson = JsonUtility.ToJson(list);
    File.WriteAllText(path, newJson);
}
```

```
Unity Script (1 asset reference) | 2 references
public class LoadHeight : MonoBehaviour
{
    public List<string> addCalibration = new List<string>();
    public static List<Vector3> loadData = new List<Vector3>();

    0 references
    public void LoadHeightData()
    {
        string path = Application.persistentDataPath + "/UserHeightList.json";
        string json = File.ReadAllText(path);
        HeightList loadHeight = JsonUtility.FromJson<HeightList>(json);
        addCalibration = loadHeight.heightList;

        loadData.Clear();

        //Debug.Log(addCalibration);

        foreach (string height in addCalibration)
        {
            //Debug.Log("String height: " + height);

            string clean = height.Replace("(", "").Replace(")", ""); // Clean string from parentheses
            string[] split = clean.Split(',').Select(p => p.Trim()).ToArray(); // split the string
            float[] part = split.Select(s => float.Parse(s, CultureInfo.InvariantCulture)).ToArray(); //parse it into float

            Vector3 JsonVector3 = new Vector3(part[0], part[1], part[2]);
            loadData.Add(JsonVector3);
            //Debug.Log(JsonVector3);
        }
    }
```

Login

This system is necessary to keep the exercises personal and make sure the physiotherapist can keep tabs on every person.

The following three buttons (admin, gavin and john) are the current users on this system. Each user only have to write the password to login. When pressing the “andere gebruiker” you’ll get the same panel but you can write a new user name. To be added on the list.

Pressing the guest button will ignore logging in and jump straight to the main menu. Data will not be saved when using the guest account.

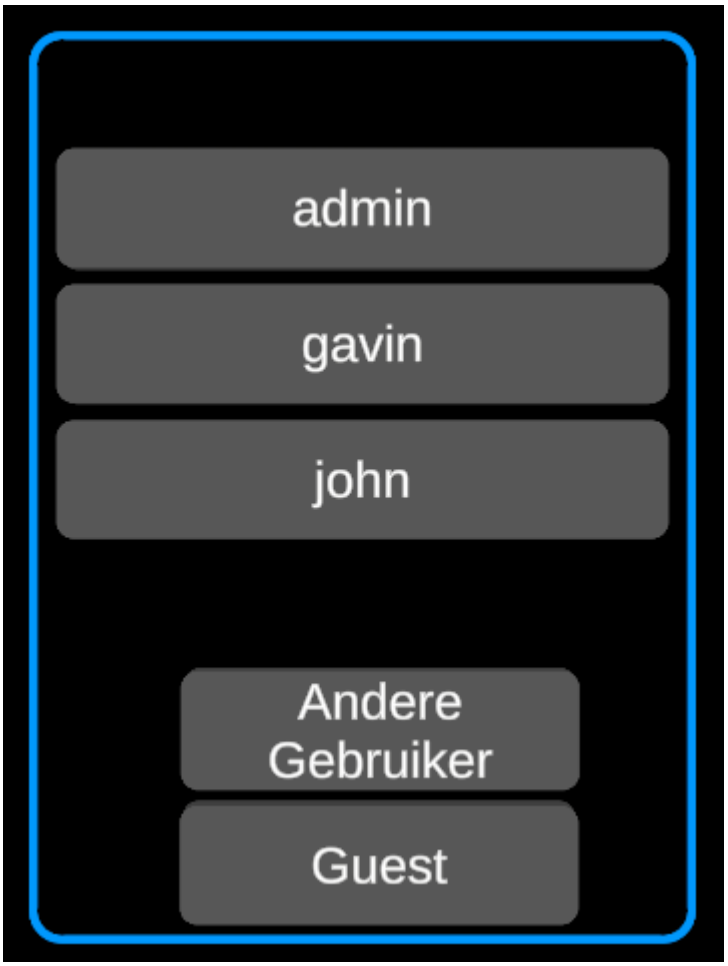

Figure 10: Start screen. The first screen you’ll see.

Figure 11: Login screen using "Andere Gebruiker"

Left shows what you see when pressing the "Andere gebruiker". Right shows when you press the button named admin.

The clear difference is the username. Right is grayed out meaning you cannot alter that section wheras the left you can add a new user if it is in the database. Both panels also has a "go back" button. So you can always go back when needed.

Pressing login without filling the textboxes will yield error messages.

When pressing the textboxes a keyboard will appear. Which is located just in front of you in a tilted manner. Pressing the login button or the enter button on the keyboard will make the keyboard disappear.

Figure 12: Login screen using an user button, For this Admin

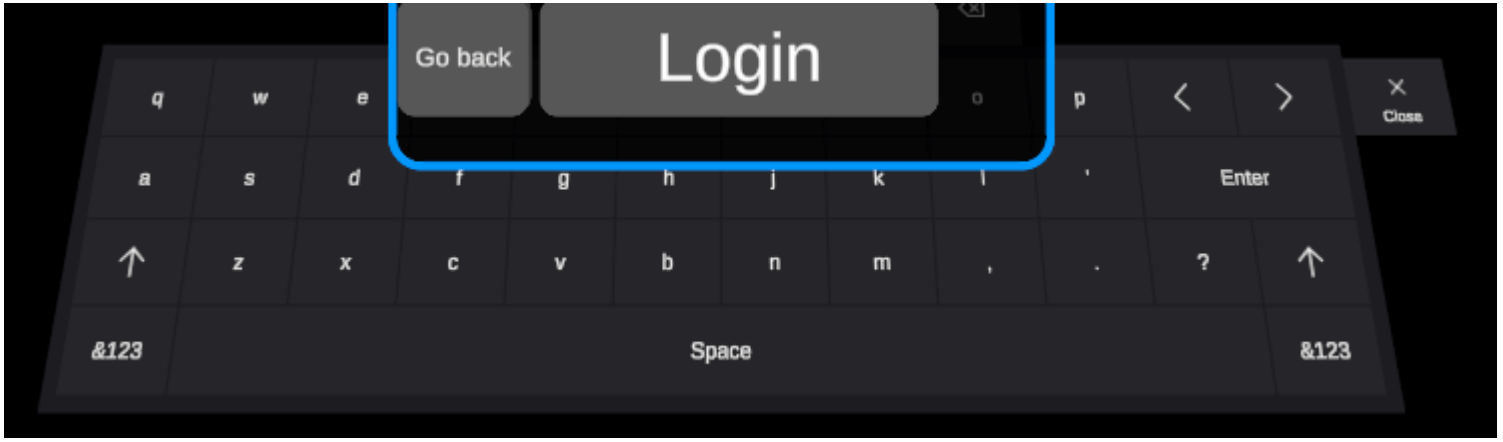

Figure 13: Keyboard that pops if you select a textbox

For the scripts we'll focus on these two:

- LoginManager
- LoadUserButton

There are two important parts in the login screen scripts. This is one of them. This is tied to the backend code using the API response. Which means you'll need IntelliJ and Docker to make sure you can login as intended.

If the textboxes aren't filled in you'll go the error messages.

If the textboxes are filled in. As example we'll go with the user "Test" So we'll write in that name and use the password "0000". The login is a success.

First we'll save the username and token. Then we go to the addUserToJsonFile and end it on LoginSucceeded.

```
// Checks the data on both text fields to ensure if the user can login
0 references
public void CheckLoginData()
{
    if (_identifier != null && _pincode != null)
    {
        StartCoroutine(userService.Login(
            new UserLoginDTO { identifier = _identifier, pincode = _pincode },
            // Login completed
            onSuccess: ApiResponse =>
            {
                // Saves token and username
                User.SetLogin(ApiResponse.data.token, "Needs to be implemented");

                // Save user in the json file
                AddUserToJsonFile();

                LoginSucceeded();
            },
            // Error message
            onError: error =>
            {
                if (error == null || string.IsNullOrEmpty(error.message)) {
                    errorMessage.text = "An unknown error occurred.";
                }
                else {
                    errorMessage.text = error.message;
                }

                StartCoroutine(RemoveErrorText());
            }
        ));
    }
    else
    {
        errorMessage.text = "Username or pincode not filled in!";
        StartCoroutine(RemoveErrorText());
    }
}
```

```
1 reference
public void AddUserToJsonFile()
{
    string path = Application.persistentDataPath + "/UserList.json";
    // Checks file for users
    if (File.Exists(path))
    {
        string existingJson = File.ReadAllText(path);
        UsernameList loaded = JsonUtility.FromJson<UsernameList>(existingJson);
        existingUserList = loaded.Usernames;
    }

    // Check if user exist or not
    if (!existingUserList.Contains(_identifier))
    {
        existingUserList.Add(_identifier);
    }

    // Saves the whole list in Json file, adding the new user to it
    UsernameList list = new UsernameList { Usernames = existingUserList };
    string newJson = JsonUtility.ToJson(list);
    File.WriteAllText(path, newJson);
}
```

It first checks if the list exists. Then we'll read every name from that json file to the existingUserList. Then we check if it exists or not. Since Test is not in the list we'll add it to the list. Then save the list.

So after starting the game again. The button with the name Test should appear in the list of users.

```
1 reference
private void LoginSucceeded()
{
    errorMessage.text = "Login success!";
    StartCoroutine(LoadingMenu());
    StartCoroutine(RemoveErrorText());
}

// After a set amount of time, remove the error text
3 references
IEnumerator RemoveErrorText()
{
    yield return new WaitForSeconds(3f);
    errorMessage.text = "";
}
```

Then we go to loginSucceeded and make sure to load Main menu and remove the error messages after a set amount of time.

```
// Checks users in Json file and initializes buttons
1 reference
public void LoadExistingUsers()
{
    string path = Application.persistentDataPath + "/UserList.json";

    Debug.unityLogger.Log(path);
    // Verwijder eerst bestaande knoppen
    foreach (Transform child in ButtonContainer)
    {
        Destroy(child.gameObject);
    }

    // Laad gebruikers uit JSON
    if (File.Exists(path))
    {
        string json = File.ReadAllText(path);
        UsernameList loaded = JsonUtility.FromJson<UsernameList>(json);
        existingUserList = loaded.Usernames;
    }
    else
    {
        Debug.Log("No users found.");
        existingUserList.Clear();
    }

    // Maak knop voor elke gebruiker
    foreach (string username in existingUserList)
    {
        GameObject buttonObj = Instantiate(ButtonPrefab, ButtonContainer);
        Button button = buttonObj.GetComponent<Button>();
        TMP_Text buttonText = buttonObj.GetComponentInChildren<TMP_Text>();

        buttonText.text = username;
        button.onClick.AddListener(() => ChosenUser(username));
    }
}
```

When starting the game it'll check the json file to load the buttons of each and every user saved.

First by deleting buttons otherwise it'll stack and make a mess then check if the file exists for the users to load. Then create a button for each and every user loaded.

```
// Choosing a specific initialized button focusses on that user
1 reference
public void ChosenUser(string chosenUser)
{
    startMenu.SetActive(false);
    loginMenu.SetActive(true);
    _loginManager.SetIdentifier(chosenUser);
}
```

Pressing the named user will show the panels showed earlier (figure 12) and gray out the username textbox. The name will still be stored and checked accordingly when logging in. Using credentials on the picture below

```
// Makes sure the text written on the field boxes are saved
0 references
public void ReadIdentifier(string user)
{
    _identifier = user;
}

0 references
public void ReadPincode(string pass)
{
    _pincode = pass;
}

1 reference
public void SetIdentifier(string username)
{
    _identifier = username;
    UserInputField.text = username;

    UserInputField.interactable = false;    }
```