

TEAM 23

SOFTWAREGRUNDPROJEKT 22/23

TUTORIENDE PERSON – TANJA ZAST

TEAMMITGLIEDER

JOHANNES MARTIN ERTLE

JONATHAN HIELSCHER

YANNIK BLEILINGER

MARVIN KRAUßER

SAMED BILGILI

SÜNDÜZ CAN

ENTWICKLERHANDBUCH - BENUTZERCLIENT

INHALTSVERZEICHNIS

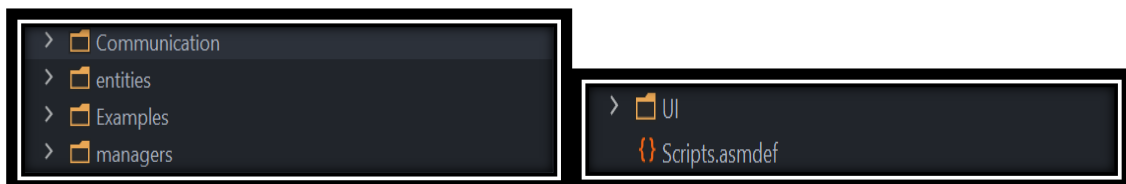
1. Allgemeine Grundstruktur.....	3
1.1 Softwarearchitektur.....	3
1.2 Weitere Elemente (Sprites, Tests, TextMeshPro Elemente)	3
2. Funktionalität.....	4
2.1 Communication.....	4
2.1.1 Read Message.....	6
2.1.2 Write Message.....	6
2.1.3 Connection Lost & Timeout.....	7
2.2 Entities.....	9
2.2.1 GameBoard Usage.....	9
2.2.2 Player Usage.....	12
2.2.3 Projectile Usage.....	15
2.3 Scenes.....	16
2.3.1 Hauptmenü.....	16
2.3.2 Connect-Server Scene.....	17
2.3.3 EndCard.....	18
2.4 Management.....	19
2.4.1 Game Manager.....	19
2.4.2 Message Manager.....	24
2.5 UI-Building.....	25
2.5.1 Card Building.....	25
2.5.2 Charakter Building.....	27
2.5.3 Camera Movement.....	29
2.5.4 Game Pause.....	30
3. Tests.....	31
3.1 EditMode Test.....	31
3.2 PlayMode Test.....	32
3.3 Testing in Unity.....	33

1. Allgemeine Grundstruktur

1.1 Softwarearchitektur

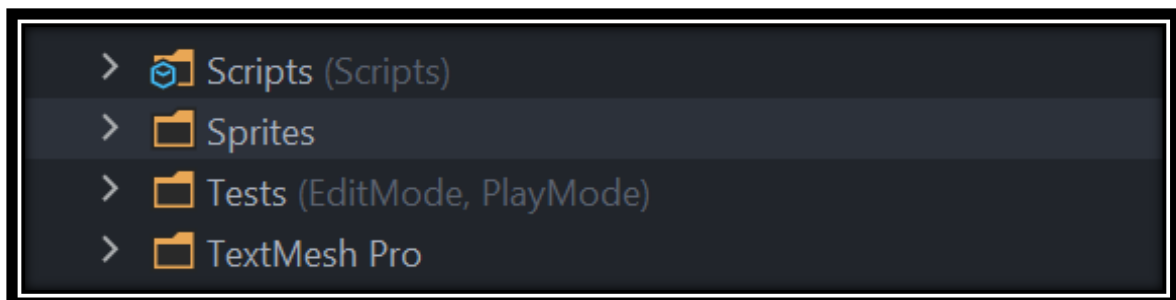
Die Softwarestruktur folgt dem Model-View-Controller. MVC bietet klare Verantwortlichkeiten und eine Aufteilung der Software in drei Hauptkomponenten: Modell, Darstellung und Steuerung. Dieses Handbuch gibt einen Überblick über MVC in Unity.

Der **Communication**-Namespace ermöglicht den Nachrichtenaustausch zwischen Server und Client. Die Entitäten sind grundlegende Bestandteile des Spiels und werden mit visuellen Objekten wie dem Gameboard, der Spielerdarstellung und dem Projektil verknüpft. Der **Manager**-Namespace behandelt die Spiellogik und Anwendungsverwaltung. Der **UI**-Namespace ist ausschließlich für die Benutzeroberflächendarstellung und Benutzerführung zuständig.



1.2 Weitere Elemente (Sprites, Tests, TextMeshPro Elemente)

Andere Elemente wie Sprites, Tests, TextMeshPro Elemente lassen sich in dem Ordner Sprites finden.



2.Funktionalität

2.1 Communication

Die Benutzeranleitung erklärt die Verwendung des Codes in der **Communication**-Klasse, die WebSockets zur Kommunikation mit einem Server nutzt. Hier ist eine Anleitung zur Nutzung der Funktionen und Methoden dieser Klasse.

Konfiguration der Kommunikation

```
public Communication(string ipAddress, int port)
{
    MessageHandler(ipAddress, port);
}
```

Zur Einrichtung der Serverkommunikation wird eine Instanz der **Communication**-Klasse erstellt. Dabei werden IP-Adresse und Port des Servers als Konstruktor Parameter übergeben.

Verbindung zum Server

```
MessageHandler(ipAddress, port);
```

Die Methode **MessageHandler** wird aufgerufen, um eine Verbindung zum Server herzustellen. Hierbei wird die WebSocket-Verbindung hergestellt und Event-Handler für verschiedene Ereignisse wie den Empfang von Nachrichten, das Schließen der Verbindung oder Fehler festgelegt.

Nachrichten Senden

```
public void SendMessage(string message)
{
    try
    {
        _webSocket.Send(message);
    }
    catch (Exception e)
    {
        Debug.Log(message: "There is no connection to a server");
    }
}
```

Um eine Nachricht an den Server zu senden, wird die Methode **SendMessage** aufgerufen, die die Nachricht als Zeichenfolge übergibt.

Nachrichten Empfangen

```
private void Ws_OnMessage(object sender, MessageEventArgs e)
{
    _messageQueue.Enqueue(e.Data);

    new Thread(start: () => ReceivedMessage()).Start();
}
```

Eingehende Nachrichten werden in eine Warteschlange (**_messageQueue**) eingereiht und in einem separaten Thread (**ReceivedMessage**) verarbeitet. Um empfangene Nachrichten abzurufen und zu verarbeiten, muss der **ReadMessage**-Handler implementiert und der **Communication**-Instanz zugewiesen werden.

WebSocket-Status überprüfen

```
public bool isWebSocketOpen()
{
    return _webSocket.IsAlive;
}
```

Der Status der WebSocket-Verbindung kann mithilfe der Methode **isWebSocketOpen** überprüft werden.

WebSocket-Verbindung schließen

```
public void CloseWebSocketConnection()
{
    _webSocket.Close();
}
```

Die WebSocket-Verbindung zum Server kann durch die Methode **CloseWebSocketConnection** geschlossen werden.

2.1.1 Read Message

Die **ReadMessage**-Klasse stellt Ereignisse für verschiedene Nachrichtentypen bereit und ermöglicht das Lesen von **JSON**-formatierten Nachrichten sowie das Auslösen entsprechender Ereignisse für jeden Nachrichtentyp.

Um die Kommunikation zwischen Server und Client zu ermöglichen, empfängt man die gewünschten Ereignisse, die vom Server gesendet werden. Für jeden empfangenen Nachrichtentyp wurde eine entsprechende Handler-Methode implementiert

```
public delegate void OnCHARACTER_OFFERMessage(CHARACTER_OFFER_Message message);
```

```
public void ReadMessageCHARACTER_OFFER(string message)
{
    CHARACTER_OFFER_Message jsonFile = JsonConvert.DeserializeObject<CHARACTER_OFFER_Message>(message);
    Debug.Log(message: "created a Json File - CHARACTER_OFFER");

    OnCHARACTER_OFFER?.Invoke(jsonFile);
}
```

Wenn eine "Character_Offer"-Nachricht empfangen wird, wird die entsprechende Handler-Methode aufgerufen. Um eine empfangene Nachricht zu verarbeiten, wird die Methode **ReadMessages** aufgerufen und die empfangene Nachricht als Zeichenfolge übergeben. Die **ReadMessage**-Klasse erkennt automatisch den Nachrichtentyp und ruft den entsprechenden Handler auf.

2.1.2 Write Message

Die **WriteMessage**-Klasse dient zum Senden verschiedener Nachrichten an den Server, um Aktionen wie Anmeldung, Charakterauswahl oder das Senden von Spielzügen abzuschließen.

Um die **WriteMessage**-Klasse zu verwenden, wird zunächst eine Instanz davon erstellt:

```
public WriteMessage(Communication com)
{
    _communication = com;
}
```

Die **WriteMessage**-Klasse bietet ähnliche Methoden für andere Nachrichtentypen wie **PLAYER_READY**, **CHARACTER_CHOICE**, **RECONNECT**, **GOODBYE_SERVER**, **CARD_CHOICE** und **PAUSE_REQUEST**. Diese Methoden folgen einem ähnlichen Aufbau. Für jeden Nachrichtentyp kann die entsprechende Methode aufgerufen und die benötigten Daten übergeben werden. Hier ein Beispiel für eine **WriteMessage**-Methode:

```
public void WriteMessagePLAYER_READY(bool ready)
{
    PLAYER_READY_Message message = new PLAYER_READY_Message();
    message.message = Message.PLAYER_READY;
    message.data = new PLAYER_READY_Message_Data();
    message.data.ready = ready;

    SendMessage(message);
}
```

Die **WriteMessage**-Klasse vereinfacht die Erzeugung und Übermittlung verschiedener Nachrichtentypen an den Server. Dadurch können Aktionen in der Anwendung schnell und einfach ausgelöst werden. Die Verwendung dieser Klasse ermöglicht eine effiziente Verwaltung der Kommunikation in einer Mehrspieleranwendung.

2.1.3 Connection Lost & Timeout

Die **ConnectionLost**-Klasse behandelt Verbindungsverlust und Timeouts in einer Anwendung und ermöglicht eine erneute Verbindung zum Server.

```
public void Reconnect()
{
    _disconnectText.color = Color.red;
    _disconnectText.text = "Can't reconnect";

    MessageManager.instance = FindObjectOfType<MessageManager>();
    MessageManager.reconnectSemaphore = new SemaphoreSlim(initialCount: 0);

    MessageManager.activeScene = "Connection Lost";
    MessageManager.instance.tryingToReconnect = true;
    MessageManager.instance.OpenWebSocket(StaticVariables.ip, StaticVariables.port);
    MessageManager.connectionAttempts++;
    Thread.Sleep(millisecondsTimeout: 1000);

    MessageManager.instance._writeMessage.WriteMessageRECONNECT(StaticVariables.playerName, StaticVariables.reconnectToken);
}
```

Bei Verbindungsverlust kann die Methode **Reconnect** aufgerufen werden, um eine erneute Verbindung zum Server herzustellen.

```

void Start()
{
    _disconnectText.color = Color.red;
    if (errorText is not null && !errorText.Equals(""))
    {
        _disconnectText.text = errorText;
        errorText = "";
    }
    else
    {
        _disconnectText.text = "Keinen Error Text";
    }

    if (onError || onInvalidMessage)
    {
        _reconnectButton.SetActive(false);
    }
}

```

Die **ConnectionLost**-Klasse handhabt auch Timeouts und Fehler während der Verbindungsphase. In der **Start**-Methode wird überprüft, ob ein Fehler-Text vorhanden ist.

```

[SerializeField] private TMP_Text _disconnectText;  🔗 ConnectionLostText (TextMeshProUGUI)
[SerializeField] private GameObject _reconnectButton;  🔗 Reconnect

public static string errorText;
public static bool onError;
public static bool onInvalidMessage;

```

Die **ConnectionLost**-Klasse umfasst alle Variablen und **UI**-Elemente, die mit der **ConnectionLost**-Szene verbunden sind. Sie ermöglicht die Behandlung von Verbindungsverlust und die erneute Verbindung zum Server. Durch die Verwendung dieser Klasse wird Benutzern automatisch bei Verbindungsproblemen geholfen, indem diese erkannt und behoben werden.

2.2 Entities

2.2.1 GameBoard Usage

```
[Header("Tiles")]
[SerializeField] private TileBase hole;    🗑️ tileSprites_1.asset
[SerializeField] private TileBase[] grass; 🗑️ Serializable
[SerializeField] private TileBase lembas;  🗑️ tileSprites_4.asset
[SerializeField] private TileBase river;   🗑️ tileSprites_6.asset
[SerializeField] private TileBase checkpoint; 🗑️ tileSprites_2.asset
[SerializeField] private TileBase eagle;   🗑️ tileSprites_5.asset
[SerializeField] private TileBase start;   🗑️ tileSprites_3.asset
[SerializeField] private TileBase wall;    🗑️ tileSprites_13.asset

[SerializeField] private TileBase eye_north; 🗑️ tileSprites_7.asset
[SerializeField] private TileBase eye_east;  🗑️ tileSprites_12.asset
[SerializeField] private TileBase eye_south; 🗑️ tileSprites_8.asset
[SerializeField] private TileBase eye_west;  🗑️ tileSprites_9.asset
```

Damit das **Gameboard** verwendet wird, um das Spielbrett im Spiel darzustellen werden verschiedene Variablen und Sprites für die verschiedenen Felder definiert. Diese ermöglichen die Interaktion mit dem Spielbrett, sowie die Darstellung einzelner Spielbrett Komponenten für den Benutzer.

```
public void initializeTilemap(BoardConfig config)
{
    HEIGHT = config.height;
    WIDHT = config.width;

    //initializes the size of the tilemap with all grass tiles
    for (int i = 0; i < HEIGHT; i++)
    {
        for (int j = 0; j < WIDHT; j++)
        {
            var randomInt = _random.Next(grass.Length);
            mainTilemap.SetTile(position: new Vector3Int(x: j, y: i), grass[randomInt]);
        }
    }
}
```

(Ein Beispiel Codeausschnitt aus der Methode **initializeTilemap**, zeigt die Implementierung für die Grasfelder)

Initialisierung des Spielbretts: Die Methode **initializeTilemap** wird aufgerufen, um das Spielbrett gemäß den übergebenen Konfigurationsdaten zu initialisieren. Dies umfasst das Platzieren der verschiedenen Feldtypen wie Gras, Löcher, Checkpoints, Flüsse, Lembas-Felder, Adler-Felder, das Auge und Wände. Außerdem wird die Kamera entsprechend der Größe des Spielbretts eingerichtet, um das gesamte Brett sichtbar zu machen.

```

public void onBoardState(BoardState state)
{
    foreach (var lembasField :LembasFields in state.lembasFields)
    {
        Debug.Log(message: "Lembas at "+lembasField.position[0]+" "+ lembasField.position[1] +" has Lembas: " + lembasField.amount);
        var pos = new Vector3Int(x:lembasField.position[0], y:lembasField.position[1]);
        //lembasAmounts[new Vector3Int(lembasField.position[0], lembasField.position[1])] = lembasField.amount;
        UpdateTextObject(pos, newText:lembasField.amount.ToString());
    }
}

```

Aktualisierung des Spielbrettzustands: Die Methode **onBoardState** wird aufgerufen, um den aktuellen Zustand des Spielbretts zu aktualisieren. Sie erhält Informationen über die Positionen der Lembas-Felder und deren Anzahl. Anhand dieser Informationen werden die entsprechenden Textobjekte aktualisiert, um die Anzahl der Lembas anzuzeigen.

```

private void CreateTextObject(Vector3Int tilePosition, string text)
{
    //create new gameObject
    GameObject textObject = new GameObject(name: "BoardInfo");

    //make gameObject TMPPro
    TextMeshPro tmp = textObject.AddComponent<TextMeshPro>();

    //get tile coordinates and connect relative to tile position
    Vector3 worldPosition = mainTilemap.CellToWorld(tilePosition);
    worldPosition += new Vector3(x:0.5f, y:0.5f, z:0f);

    textObject.transform.position = worldPosition;

    //insert text
    tmp.text = text;

    //set font size
    tmp.fontSize = 8;

    //anchor text to tile-middle
    tmp.alignment = TextAlignmentOptions.Center;

    //set textObject to parent and connect to gameboardObject to be visible for updates
    textObject.transform.parent = transform;
}

```

Erstellung von Textobjekten: Die Methode **CreateTextObject** verwenden wir, um Textobjekte zu erstellen, die mit bestimmten Kacheln auf dem Spielbrett verbunden sind. Diese Textobjekte zeigen Informationen wie die Anzahl der Lembas oder die Anzahl der Checkpoints an.

```

public void UpdateTextObject(Vector3Int tilePosition, string newText)
{
    //call GetTextObjectMethod based on tilePosition to change
    GameObject textObject = GetTextObject(tilePosition);

    if (textObject != null)
    {
        //get TextMeshPro-component of target object
        TextMeshPro textMeshPro = textObject.GetComponent<TextMeshPro>();

        //update displayed text
        textMeshPro.text = newText;
    }
}

```

Aktualisierung von Textobjekten: Die Methode **UpdateTextObject** aktualisiert den angezeigten Text auf einem bestimmten Textobjekt, das mit einem **GameObject** auf dem Spielbrett verbunden ist.

```

private GameObject GetTextObject(Vector3Int tilePosition)
{
    foreach (Transform child in transform)
    {
        //check for name BoardInfo and get TilePosition
        if (child.gameObject.name == "BoardInfo")
        {
            Vector3Int textMeshProTilePosition = mainTilemap.WorldToCell(child.position);

            //check if found object is on correct position method wants to change
            if (textMeshProTilePosition == tilePosition)
            {
                //tile found
                return child.gameObject;
            }
        }
    }

    return null;
}

```

Positionsermittlung von Textobjekten: Die Methode **GetTextObject** findet ein Textobjekt auf dem Spielbrett, dessen Text anschließend aktualisiert werden soll.

Insgesamt ermöglicht das **Gameboard** die Darstellung des Spielbretts und die Aktualisierung von Informationen darauf. Es erleichtert die Verwaltung der Spielbrett-Elemente wie Kacheln, Textobjekte und die Kameraeinstellungen.

2.2.2 Player Usage

Die Klasse **Player** repräsentiert einen Spieler im Spiel, mit den jeweiligen Eigenschaften und Funktionen während der Spielphase.

```
public void Initialize(string name, Role role, int lembas)
{
    this.name = name;
    this.role = role;
    ready = false;
    suspended = 0;
    reachedCheckpoints = 0;
    charakterSprite = GetComponent();
    animator = GetComponent();
}
```

Initialisierung des Spielers: Die Methode **Initialize** wird verwendet, um den Spieler mit einem Namen, der Rolle und der entsprechenden Lembas Anzahl zu initialisieren. Es werden weitere Eigenschaften des Spielers wie die Anzahl der erreichten Checkpoints, dem Lebensstatus, dem Sprite, sowie dem Animator festgelegt.

```
public void MovePlayer(Vector2Int newPosition)
{
    currPosition = newPosition;
}
```

Hier wird eine Methode **MovePlayer** erstellt, um den Spieler zu einer neuen Position zu bewegen. Die aktuelle Position des Spielers wird entsprechend aktualisiert.

```
public void TurnPlayer(Direction newDirection)
{
    direction = newDirection;
    setSpriteOnDirection();
}
```

Die Drehung des Spielers erfolgt durch die **TurnPlayer** Methode. Diese aktualisiert die Richtung des Spielers und ruft die **setSpriteOnDirection** Methode auf, um den entsprechenden Sprite einzustellen.

```

public void setSpriteOnDirection()
{
    switch (direction)
    {
        case Direction.NORTH:
            animator.SetTrigger(name: "turnup");
            break;
        case Direction.EAST:
            animator.SetTrigger(name: "turnright");
            break;
        case Direction.SOUTH:
            animator.SetTrigger(name: "turndown");
            break;
        case Direction.WEST:
            animator.SetTrigger(name: "turnleft");
            break;
    }

    Debug.Log(message: "Sprite was set for "+name);
}

```

Die Methode **setSpriteOnDirection** legt den Sprite des Spielers basierend auf seiner aktuellen Richtung festzulegen. Je nach Richtung des Spielers wird der entsprechende Trigger für den Animator ausgelöst.

```

private IEnumerator MoveToPosition(Vector3 newPosition)
{
    switch (direction)
    {
        case Direction.NORTH:
            animator.ResetTrigger(name: "stop");
            animator.SetTrigger(name: "up");
            break;
        case Direction.EAST:
            animator.ResetTrigger(name: "stop");
            animator.SetTrigger(name: "right");
            break;
        case Direction.SOUTH:
            animator.ResetTrigger(name: "stop");
            animator.SetTrigger(name: "down");
            break;
        case Direction.WEST:
            animator.ResetTrigger(name: "stop");
            animator.SetTrigger(name: "left");
            break;
        default:
            throw new ArgumentOutOfRangeException();
    }
}

```

Die Coroutine **MoveToPosition** wird gestartet, um den Spieler zur angegebenen Position zu bewegen. Abhängig von der aktuellen Richtung des Spielers wird der entsprechende **Animator-Trigger** ausgelöst, um die richtige Bewegungsanimation abzuspielen. Der Spieler wird kontinuierlich zur neuen Position bewegt, bis er diese erreicht hat.

```

public void shoot(Player target)
{
    //initialize projectile with correct settings and sprite
    GameObject projectile = Instantiate(projectilePrefab, transform.position, Quaternion.identity);

    SpriteRenderer sr = projectile.GetComponent<SpriteRenderer>();
    sr.sprite = Character.weapon; //todo: set sprite corresponding to Character

    Projectile proj = projectile.AddComponent<Projectile>();

    proj.Initialize(target, PROJECTILE_SPEED);
}

```

Nachdem der Spieler eine Laufrichtung besitzt, ist notwendig, dass der Spieler schießen kann. Die Methode **shoot** ist zuständig, um den Spieler einen Schuss auf ein Ziel abfeuern zu lassen. Ein Projektil-Objekt wird erstellt und mit den entsprechenden Eigenschaften initialisiert. Demnach wird die Projektil-Animation ausgeführt, und das Projektil wird auf das Ziel abgefeuert.

```

public void revivePlayer()
{
    isDead = false;
    animator.SetBool(name: "dead", value: false);
    //make the player visible again
    Color currentColor = charakterSprite.color;
    currentColor.a = 1f;
    charakterSprite.color = currentColor;
}

```

Demnach wird die Methode **revivePlayer** benötigt, um Spieler zu Wiederbeleben. Der Spieler wird als lebendig markiert, indem die entsprechende Animator-Variable und die Transparenz des Spieler-Sprites zurückgesetzt werden.

```

public void teleportPlayer(Vector3 newPosition)
{
    var adjVector = new Vector3(x: newPosition.x + 0.5f, y: Height - newPosition.y - 1);
    transform.position = adjVector;
    _currPosition = new Vector2Int((int)newPosition.x, (int)newPosition.y);
}

```

Diese Methode teleportiert den Spieler zu einer neuen Position im Spiel. Sie nimmt eine 3D-Vektorposition als Parameter an und berechnet dann eine angepasste Vektorposition, um den Spieler an den neuen Ort zu versetzen. Die Methode aktualisiert auch die interne Spielerposition auf eine neue 2D-Vektorposition, die auf ganzzahlige Koordinaten der neuen Position abgerundet wird.

2.2.3 Projectile Usage

```
private Transform targetTransform;  
private Player target;  
private float speed;
```

Um das Projektil zu konstruieren, werden private Variablen wie **targetTransform** (Ziel-Transform), **target** (Ziel-Spieler) und **speed** (Geschwindigkeit) initialisiert.

```
public void Initialize(Player target, float speed)  
{  
    this.targetTransform = target.transform;  
    this.target = target;  
    this.speed = speed;  
}
```

Die Methode **Initialize** initialisiert das Projektil mit einem Ziel und einer Geschwindigkeit.

Die Methode **Update** wird pro Frame aufgerufen. Sie prüft, ob das Ziel-Transform vorhanden ist oder die Distanz zwischen der aktuellen Position des Projektils und der Zielposition kleiner oder gleich 0,1f ist:

```
private void Update()  
{  
    if (targetTransform is null || Vector3.Distance(@this.transform.position, targetTransform.transform.position) <= 0.1f )  
    {  
        // Destroy the projectile when reached and plays hit animation  
        Destroy(gameObject);  
        StartCoroutine(routine: target.hit());  
    }  
}
```

Wenn dies der Fall ist, wird das Projektil zerstört und eine Coroutine **hit** des Ziels gestartet.

```
else  
{  
    // Move the projectile towards the target  
    transform.position = (Vector3) Vector2.MoveTowards(current: (Vector2) transform.position, (Vector2) targetTransform.position, maxDistanceDelta: speed * Time.deltaTime);  
}
```

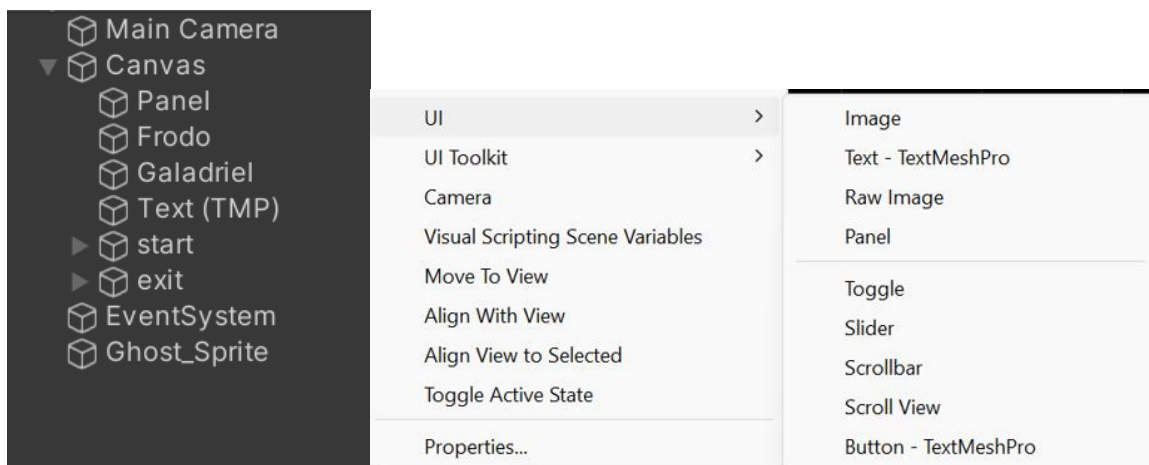
Wenn das Ziel nicht erreicht ist, bewegt sich das Projektil in Richtung des Ziels, indem es seine Position mithilfe von **Vector2.MoveTowards** schrittweise aktualisiert.

2.3 Scenes

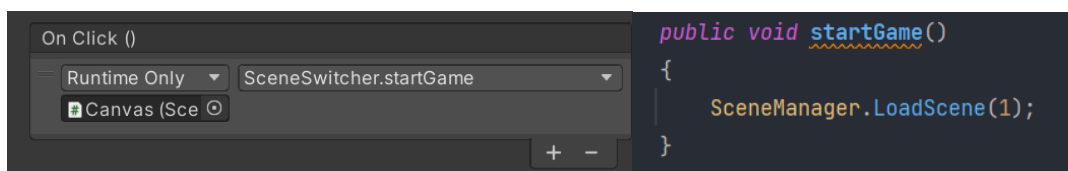
2.3.1 Hauptmenü



Das Hauptmenü besteht aus Textfeldern, einem Hintergrundbild und Buttons. Der **StartGame**-Button ist verantwortlich, um in die **Connect-To-Server** Scene zu wechseln, der **Quit**-Button verlässt das Spiel.



Buttons und Textfelder werden in Unity hinzugefügt, indem ein **Canvas**-Objekt erstellt wird. Dazu klickt man im Unity-Editor mit der rechten Maustaste auf **Canvas**, man navigiert zu **UI** und wählt **Button - TextMeshPro** aus. Um Textfelder hinzuzufügen, verwendet stattdessen die Option **Text - TextMeshPro**.



Um das **Button-Click** Ereignis auszulösen, lädt man die Klasse **SceneSwitcher** über das **Canvas** in die **OnClick** Ereignis und wählt die entsprechende Methode aus, die man ausführen möchte. Beispielsweise lädt die Methode **startGame** die Szene mit dem Build-Index von 1, um in entsprechende Szene zu wechseln.

2.3.2 Connect-Server Scene



Die **Connect-Server** Szene ist zuständig, um eine Verbindung zum Server herzustellen und demnach in die Spielszene zu wechseln. Dazu wurden vier Textfelder und zwei Buttons angelegt. Die Textfelder **IP-Adresse** und **Spieler Name** dienen als Input Textfelder. In diesen Feldern wird die entsprechende **IP-Adresse** (zu Verbindung zum Server) und der **Spieler Name** angegeben. Der Button **Main Menu** kehrt ins Hauptmenü zurück und der Button **Verbinden** verbindet sich mit dem Server und wechselt **Lobby** Szene.

```
[SerializeField] private TMP_InputField _ip; // InputField (TMP) IP-Adresse (TMP_InputField)
[SerializeField] private TMP_InputField _port; // InputField (TMP) Portnummer (TMP_InputField)
[SerializeField] private TMP_InputField _playerName; // InputField (TMP) Spieler Name (TMP_InputField)
[SerializeField] private TMP_Dropdown _playerOrSpectator; // Dropdown Spieler oder Zuschauer (TMP_Dropdown)
[SerializeField] private Button _connectButton; // Verbinden (Button)
[SerializeField] private TMP_Text _connectingText; // Connecting... (TextMeshProUGUI)

private MessageManager _messageManager;

private readonly int defaultPort = 3018;
private readonly Role defaultPlayerOrSpectator = Role.PLAYER;

private bool _connection;
private bool _ipCorrect;
private bool _portCorrect;
private bool _playerNameCorrect;

private bool connectionFailedBool;
```

Hier legen die Referenzen zu verschiedenen Unity **UI-Elementen** fest, wie z.B. **Texteingabefelder**, **Dropdown-Menüs** und **Buttons**. Zudem werden verschiedene boolesche Variablen und eine statische Fehlermeldung definiert.

2.3.3 EndCard

Die **EndCard**-Klasse repräsentiert die **Endscene**, die angezeigt wird, wenn das Spiel beendet ist:

```
public void ExitGame()
{
    Debug.Log( message: "Quitting Game..."); // Gibt eine Debug-Nachricht aus
    Application.Quit(); // Beendet die Anwendung
}

/// <summary>
/// Updates the UI to display the winner from the Game_End message
/// </summary>
/// <param name="message"></param>
1 usage  Samed Bilgili +2
public void OnGAME_END(GAME_END_Message message)
{
    Debug.Log( message: "GAME_END_Message");

    // Spielergebnisse anzeigen
    string winner = message.data.winner;

    // Gewinner anzeigen
    DisplayWinner(winner);
}
```

```
private void DisplayWinner(string winner)
{
    // Gewinner im TextMeshPro-Feld anzeigen
    winnerText.text = "Gewinner: " + winner;
    Debug.Log( message: "Gewinner: " + winner);
}
```

Mit diesem Codebeispiel können wir den Gewinner und die Informationen des Servers auf der **Endscene** anzeigen lassen. Die **ExitGame** Methode ermöglicht es dem Benutzer, das Spiel zu beenden. Der entsprechende Aspekt ist es den korrekten Gewinner anzeigen zu lassen. Dies geschieht in der **DisplayWinner** Methode, indem der Gewinner mit einem **TextMeshPro** Feld korrekt angezeigt wird.

2.4 Management

2.4.1 Game Manager

Die **GameManager**-Klasse ist zuständig für den Spielablauf und die Aktualisierung des Spielzustands. Sie verarbeitet Servernachrichten, steuert die Spiellogik und führt entsprechende Aktionen aus.

```
[Header("Player Prefab")]
public Player playerPrefab;  # PlayerPrefab (Player)

public GameObject projectilePrefab;  # ProjectilePrefab

private Player mainPlayer;
public List<Player> allPlayers;  # Serializable
public List<Player> referencedPlayers;  # Serializable
public List<Player> spectators;  # Serializable
public List<Player> ais;  # Serializable
public List<Player> readyPlayers;  # Serializable
public int currentRound;  # Unchanged

[Header("Shot Event Variables")]
//variables for shot message
public TextMeshProUGUI Shot_Text;  # Shot_Text (TextMeshProUGUI)
public RectTransform panelRectTransform;  # Shot_Panel (Transform)
private bool isAnimating = false;
private Queue<string> messageQueue = new Queue<string>();

private Color originalTextColor;
private Color originalPanelColor;
```

Es werden Variablen und Objekte für das Spiel definiert, darunter das **Header**-Attribut zur besseren **Code**-Organisation.

```
instance = this;
allPlayers = new List<Player>();
spectators = new List<Player>();
ais = new List<Player>();
readyPlayers = new List<Player>();
referencedPlayers = new List<Player>();
currentRound = 0;

panelRectTransform.anchoredPosition = new Vector2(x:Screen.width*3, panelRectTransform.anchoredPosition.y);

//get test config
_gameboard = gameObject.GetComponent<Gameboard>();
_gameboard.initializeTilemap(StaticVariables.boardConfig);

onParticipants_Info(StaticVariables.participantsInfoMessage.data);

MessageManager.gameStateSemaphore.Release();
```

Verschiedene Listen für Spieler, Zuschauer und KI-Spieler werden initialisiert. Die Position eines **RectTransform**-Panels wird festgelegt. Das **Gameboard**-Skript wird referenziert und das Spielbrett initialisiert. Informationen über die Teilnehmer des Spiels werden verarbeitet und eine Semaphore wird freigegeben.

```

public void onParticipants_Info(PARTICIPANTS_INFO_Message_Data info)
{
    foreach (var player:string in info.players)
    {
        //if player not in array
        if (!allPlayers.Any(p:Player => p.name.Equals(player)))
        {
            var newPlayer = Instantiate(playerPrefab, Vector3.zero, Quaternion.identity);
            newPlayer.Initialize(player, Role.PLAYER, lembas: 3 ); //todo set lembas to config
            newPlayer.Height = _gameboard.HEIGHT;
            Debug.Log(message: "adding Players: " + newPlayer.name);
            allPlayers.Add(newPlayer);
        }
    }
}

```

In der Methode zur Initialisierung neuer Spieler, KIs oder Zuschauer werden diese hinzugefügt oder in den Bereitschaftszustand versetzt. Eine Schleife durchläuft jedes Element in der Sammlung **info.players**.

```

public void onGameState(GAME_STATE_Message state)
{
    referencedPlayers.Clear();

    var stateData = state.data;

    foreach (var playerState in stateData.playerStates)
    {
        onPlayerState(playerState);
    }

    _gameboard.onBoardState(stateData.boardState);
    currentRound = stateData.currentRound;

    deleteDisconnected(allPlayers.Except(referencedPlayers).ToList());
}

```

Spielerreferenzen in der Liste **referencedPlayers** werden gelöscht. Die Spielerzustände aus **stateData** werden verarbeitet und die Spielbrettzustände übergeben. Spieler, die vom Spiel getrennt wurden, werden aus der Liste **allPlayers** entfernt.

```

public void onCardEvent(CARD_EVENT_Message message)
{
    //todo: maybe a log chat for played moves

    referencedPlayers.Clear();
    foreach (var multiPlayerState :List<PlayerState> in message.data.playerStates)
    {
        foreach (var playerState in multiPlayerState)
        {
            onPlayerState(playerState);
        }
    }

    foreach (var boardState in message.data.boardStates)
    {
        _gameboard.onBoardState(boardState);
    }

    deleteDisconnected(allPlayers.Except(referencedPlayers).ToList());
}

```

Die Methode **onCardEvent** wird dementsprechend aufgerufen, wenn eine **CARD_EVENT**-Nachricht empfangen wird. Sie aktualisiert den Zustand der Spieler und des Spielbretts basierend auf den Daten in der Servernachricht und entfernt getrennte Spieler aus der Spielerliste.

```
public void onPlayerState(PlayerState state)
{
    Debug.Log(message: "allPlayers Length: " + allPlayers.Count);
    var player = allPlayers.Find(match: p:Player => p.name.Equals(state.playerName));

    if (player is null) {
        Debug.Log(message: "Player is null!!!");
        var newPlayer = Instantiate(playerPrefab, Vector3.zero, Quaternion.identity);
        newPlayer.Initialize(state.playerName, Role.PLAYER, lembas: 3); //todo set lembas to config
        newPlayer.Height = _gameboard.HEIGHT;
        Debug.Log(message: "adding Players: " + newPlayer.name);
        allPlayers.Add(newPlayer);

        newPlayer.ready = true;
        readyPlayers.Add(newPlayer);

        player = newPlayer;
    }

    if (player.isDead && state.lives > 0)
    {
        //revive player - make him visible again
        player.revivePlayer();
        AddMessage(player.name + " was revived");
    }

    if (player.reachedCheckpoints+1 == state.reachedCheckpoints)
    {
        //player has reached a checkpoint
        player.reachedCheckpoints = state.reachedCheckpoints;
        AddMessage(player.name + " has reached a checkpoint");
        //todo: mark checkpoint as reached
    }

    Card[] playedCards = state.playedCards;

    referencedPlayers.Add(player);
}
```

Die Methode **onPlayerState** aktualisiert den Zustand eines Spielers. Sie sucht den Spieler in der Liste **allPlayers**, erstellt einen neuen Spieler, falls nicht gefunden, aktualisiert Lebenspunkte, Lembas-Anzahl und andere Eigenschaften, steuert Animation und Bewegung, behandelt Tod und Wiederbelebung sowie gespielte Karten und aktualisiert die Referenzliste. Der Zustand eines Spielers wird aktualisiert, Darstellung und Aktion des Spielers zu gewährleisten.

```

player.Lives = state.lives;
player.Lembas = state.lembasCount;
player.suspended = state.suspended;
player.turnOrder = state.turnOrder;
player.spawnPosition.Set(x: state.spawnPosition[0], y: state.spawnPosition[1]);

//set sprite and controller for animation
player.Character = getCharakterSprite(state.character);
player.animator.runtimeAnimatorController = player.Character.overrideController;

player.TurnPlayer(state.direction);
player.MovePlayer( newPosition: new Vector2Int(x: state.currentPosition[0], y: state.currentPosition[1]));

```

Der Zustand eines Spielers wird aktualisiert, einschließlich Lebenspunkte, Lembas-Anzahl, Aussetzungsstatus, Zugreihenfolge, Position, Sprite und Animation, um die korrekte Darstellung und Aktion des Spielers zu gewährleisten.

```

public void onShotEvent (SHOT_EVENT_Message message)
{
    referencedPlayers.Clear();

    var shooter:Player = allPlayers.Find( match: p:Player => p.name.Equals(message.data.shooterName));
    var target:Player = allPlayers.Find( match: p:Player => p.name.Equals(message.data.targetName));

    AddMessage(shooter.name + " has shot " + target.name);
    //the player class handles the shot animation etc.
    shooter.shoot(target);
    //start animation for shot event in Game
    //adds message to queue which then gets executed
    AddMessage(shooter.name + " has shot " + target.name);

    foreach (var playerState in message.data.playerStates)
    {
        onPlayerState(playerState);
    }

    deleteDisconnected(allPlayers.Except(referencedPlayers).ToList());

    Debug.Log( message: shooter.name + "has shot " + target.name);
}

```

Die Behandlung eines Schussevents erfolgt in der Methode **onShotEvent**. Spieler und Ziel werden gesucht, eine Nachricht wird hinzugefügt und die Schussanimation ausgelöst. Der Zustand der beteiligten Spieler wird aktualisiert und getrennte Spieler entfernt.

```

public void onRiverEvent(RIVER_EVENT_Message message)
{
    referencedPlayers.Clear();
    AddMessage(message.data.playerName + " is moved by river");

    foreach (var multiplayerStates : PlayerState[] in message.data.playerStates)
    {
        foreach (var playerState in multiplayerStates)
        {
            onPlayerState(playerState);
        }
    }
    deleteDisconnected(allPlayers.Except(referencedPlayers).ToList());

    foreach (var boardState in message.data.boardStates)
    {
        _gameboard.onBoardState(boardState);
    }
}

```

Bei einem Flussevent werden die Bewegung der Spieler entlang des Flusses und die Aktualisierung des Spielbretts behandelt, um die veränderte Flussstruktur und Auswirkungen auf die Spieler darzustellen.

```

public void AddMessage(string message)
{
    messageQueue.Enqueue(message);
    if (!isAnimating)
    {
        StartCoroutine(routine: showPanel());
    }
}

```

Eine Nachricht wird zur Warteschlange hinzugefügt und auf einem Panel im Spiel angezeigt.

2.4.2 Message Manager

Die **MessageManager**-Klasse dient dazu, die Kommunikation zwischen dem Spiel und einem **WebSocket**-Server zu verwalten. Sie enthält verschiedene Methoden und Event-Handler, um auf eingehende Nachrichten zu reagieren und die entsprechenden Aktionen im Spiel auszuführen.

Ein Beispiel für die Verwendung der **MessageManager**-Klasse ist die Methode **OnHELLO_CLIENT**, die auf die **HELLO_CLIENT**-Nachricht reagiert:

```
private void OnHELLO_CLIENT(HELLO_CLIENT_Message message)
{
    _changeSceneToWaitingOnOtherPlayers = true; //set this variable = true, to switch the scene

    StaticVariables.boardConfig = message.data.boardConfig;
    StaticVariables.gameConfig = message.data.gameConfig;
    StaticVariables.reconnectToken = message.data.reconnectToken;

    _firstHello_Client = true;

    Debug.Log(message: "HELLO_CLIENT_Message");
}
```

Wenn diese Nachricht empfangen wird, wird die Variable **_changeSceneToWaitingOnOtherPlayers** auf **true** gesetzt, um zur Game-Scene zu wechseln. Daten aus der empfangenen Nachricht, wie **boardConfig**, **gameConfig** und **reconnectToken**, werden extrahiert und in entsprechenden Variablen gespeichert.

```
public static SemaphoreSlim gameStateSemaphore = new SemaphoreSlim(initialCount: 0);
public static SemaphoreSlim participantsInfoSemaphore = new SemaphoreSlim(initialCount: 0);
public static SemaphoreSlim gameEndSemaphore = new SemaphoreSlim(initialCount: 0);
public static SemaphoreSlim reconnectSemaphore = new SemaphoreSlim(initialCount: 0);
public static SemaphoreSlim cardOfferSemaphore = new SemaphoreSlim(initialCount: 0);
```

Die **MessageManager**-Klasse verwaltet **SemaphoreSlim**-Objekte als Sperren, um die Synchronisierung zwischen Threads sicherzustellen. Diese Semaphore dienen dazu, den Nachrichtenfluss zu steuern und sicherzustellen, dass bestimmte Aktionen erst ausgeführt werden, wenn bestimmte Bedingungen erfüllt sind.

Es werden verschiedene **Event**-Handler definiert, die auf bestimmte Nachrichtentypen reagieren. Diese Handler führen entsprechende Aktionen aus, wie das Aktualisieren des Spielzustands, das Anzeigen von Karten oder den Szenenwechsel. Die **MessageManager**-Klasse enthält auch Methoden zum Öffnen und Schließen der **WebSocket**-Verbindung sowie zur Überprüfung des Verbindungsstatus.

Zusammenfassend ist die **MessageManager**-Klasse die zentrale Komponente für die Kommunikation mit dem **WebSocket**-Server. Sie ermöglicht die Verarbeitung eingehender Nachrichten, die Aktualisierung des Spielzustands und die Steuerung der Spiellogik.

2.5 UI-Building

2.5.1 Card Building

Die **Kartendarstellung** in Unity ermöglicht die visuelle Darstellung von Karten. Mit Hilfe der leistungsstarken Grafikfunktionen von Unity können detaillierte 2D- und 3D-Kartenlayouts erstellt und Texturen implementiert werden.

```
public Text nameText, descriptionText;  ⓘ DescriptionText (Text)
public CardData? data;
public Image rotationImage;  ⓘ RotationImage (Image)
public Image MoveImage;  ⓘ MoveImage (Image)
```

Öffentliche Variablen für Texte (**nameText** und **descriptionText**), sowie Bildobjekte für Rotation (**rotationImage**) und Bewegung (**MoveImage**) werden deklariert.

```
public void UpdateData(CardData data)
{
    var sprite = data.GetSprite();
    var spritemove :Sprite = data.GetMoveSprite();

    if (rotationImage != null && sprite != null)
    {
        this.rotationImage.sprite = sprite;
    }

    if (MoveImage != null && spritemove != null)
    {
        this.MoveImage.sprite = spritemove;
    }

    if (sprite == null)
    {
        Debug.Log(message: "BUG: Sprite at path '" + data.spritePath + "' is null!");
    }

    this.data = data;
}
```

Die Methode **UpdateData** erhält eine **CardData**-Instanz als Parameter und aktualisiert die Sprites für das Rotationsbild (**rotationImage**) und das Bewegungsbild (**MoveImage**) basierend auf den Informationen aus der **CardData**. Wenn ein Sprite verfügbar ist, wird es den entsprechenden Bildobjekten zugewiesen.

```

void Update()
{
    if (this.data != null)
    {
        nameText.text = this.data.Value.CardName ;
        descriptionText.text = this.data.Value.CardDescription ;
    }
}

```

Die **Update**-Methode wird einmal pro Frame aufgerufen und aktualisiert die Texte auf der Karte (**nameText** und **descriptionText**) basierend auf den Informationen in **data.Value**. Wenn **data** nicht null ist, werden die Texte entsprechend aktualisiert.

```

public List<Card> selectedCards = new List<Card>(); // Serializable

[SerializeField] public Button confirmButton; // ConfirmButton (Button)

public static Button buttonInstance;

```

Es wird eine leere Liste von Kartenobjekten namens **selectedCards** erstellt. Der öffentliche Button **confirmButton** wird als serialisierbar markiert, und eine statische Variable **buttonInstance** vom Typ Button wird für den globalen Zugriff erstellt.

```

public Card[] GetSelectedCards()
{
    selectedCards.Clear();
    foreach (Transform cardTransform in transform)
    {
        Dragable card = cardTransform.GetComponentInChildren<Dragable>();
        if (card != null)
        {
            selectedCards.Add(card.cardType);
        }
        else
        {
            selectedCards.Add(item: Card.EMPTY); // Füge den Wert "Empty" hinzu, wenn keine Karte vorhanden ist
        }
    }

    return selectedCards.ToArray();
}

```

Die Methode **GetSelectedCards** leert zunächst die Liste **selectedCards** und durchläuft dann alle untergeordneten Transform-Komponenten des aktuellen **GameObjects**. Dabei wird überprüft, ob eine Karte (repräsentiert durch das Skript **Dragable**) im Panel vorhanden ist, falls dies der Fall ist, wird der Karten-Typ zur Liste **selectedCards** hinzugefügt.

2.5.2 Charakter Building

```
public GameObject optionPrefab;  🔄 Option (1)
public Transform prevCharacter;  🔄 Unchanged
public Transform selectedCharacter;  🔄 Unchanged
```

```
private void Start()
{
    foreach (CharacterUI c in CharacterSelectionUI.instance.characters)
    {
        GameObject option = Instantiate(optionPrefab, transform);
        Button button = option.GetComponent<Button>();

        button.onClick.AddListener(call: () =>
        {
            CharacterSelectionUI.instance.SetCharacter(c);
            if (selectedCharacter != null)
            {
                prevCharacter = selectedCharacter;
            }

            selectedCharacter = option.transform;
        });

        Text text = option.GetComponentInChildren<Text>();
        text.text = c.name;

        Image image = option.GetComponentInChildren<Image>();
        image.sprite = c.icon;
    }
}
```

Die Klasse **CharacterSelectionSkript** ist verantwortlich für die Charakterauswahl in einem Spiel. Beim Start wird für jeden verfügbaren Charakter in der **CharacterSelectionUI** eine Option erstellt. Jede Option wird mit einem Button versehen, der beim Klicken den ausgewählten Charakter auf der **CharacterSelectionUI** festlegt. Der ausgewählte Charakter wird dabei als das ausgewählte **GameObject** gespeichert, während der zuvor ausgewählte Charakter als vorheriges **GameObject** gespeichert wird.

```
public void Update()
{
    if (selectedCharacter != null)
    {
        selectedCharacter.localScale = Vector3.Lerp(selectedCharacter.localScale, new Vector3(1.2f, 1.2f, 1.2f), Time.deltaTime * 10);
    }

    if (prevCharacter != null)
    {
        prevCharacter.localScale = Vector3.Lerp(prevCharacter.localScale, new Vector3(1f, 1f, 1f), Time.deltaTime * 10);
    }
}
```

In der **Update**-Methode des Skripts wird das Skalierungsverhalten der ausgewählten und vorherigen Charaktere aktualisiert. Die ausgewählte Charakter-Option wird dabei schrittweise vergrößert, um die Auswahl hervorzuheben. Gleichzeitig wird der vorherige Charakter schrittweise auf seine ursprüngliche Skalierung zurückgesetzt.

```

[SerializeField] private Sprite Galadriel; 🐾 Serializable
[SerializeField] public Sprite Gandalf; 🐾 Serializable
[SerializeField] private Sprite Sam; 🐾 Serializable
[SerializeField] private Sprite Gollum; 🐾 Serializable
[SerializeField] private Sprite Baumbart; 🐾 Serializable
[SerializeField] private Sprite Arwen; 🐾 Serializable
[SerializeField] private Sprite Legolas; 🐾 Serializable
[SerializeField] private Sprite Boromir; 🐾 Serializable
[SerializeField] private Sprite Gimli; 🐾 Serializable
[SerializeField] public Sprite Pippin; 🐾 Serializable
[SerializeField] private Sprite Merry; 🐾 Serializable
[SerializeField] private Sprite Frodo; 🐾 Serializable
[SerializeField] private Sprite Aragorn; 🐾 Serializable

```

Hier werden die Sprites für die Charaktere festgelegt.

```

public static Character[] characterForSelect;
public static CharacterSelectionUI instance;

public static Character character1;
public static Character character2;
public static bool charactersChanged;

public CharacterUI[] characters= {}; 🐾 Serializable

public static CharacterUI currentCharacter;

```

Hier definieren wir statischen Variablen, um Charaktere für die Auswahl zu speichern und den Status der Charakterauswahl zu verfolgen. Zudem haben wir eine Array-Variable deklariert, um verfügbare Charaktere darzustellen.

```

public void setCharacters()
{
    characters = new CharacterUI[2];

    // Erstelle die CharacterUI-Objekte und weise sie den entsprechenden Indizes zu
    characters[0] = new CharacterUI();
    characters[1] = new CharacterUI();

    characters[0].icon = CharacterEnumToTexture(character1);
    characters[0].name = character1.ToString();
    characters[1].icon = CharacterEnumToTexture(character2);
    characters[1].name = character2.ToString();
}

```

Hier erstellen wir die **UI**-Objekte für die Charaktere und weisen den die entsprechenden Indizes zu.

```

public Sprite CharacterEnumToTexture(Character enumeration)
{
    Sprite characterImage;

    switch (enumeration)
    {
        case Character.GANDALF:
            characterImage = Gandalf;
            break;
        case Character.GALADRIEL:
            characterImage = Galadriel;
    }
}

```

Zuletzt wird die Methode **public Sprite CharacterEnumToTexture(Character enumeration)** aufgerufen, die anhand des übergebenen Character-**Enums** eine entsprechende Textur für den Charakter zurückgibt, indem sie einen Switch-Case verwendet, um das entsprechende Sprite für jeden Charakter zu setzen. Dies führt man dann fort für die andere Charaktere.

2.5.3 Camera Movement

```

protected virtual void move()
{
    Vector2 mov = new Vector2(x:0, y:0);
    if (Input.GetKey(name: "s"))
    {
        mov.y -= 1;
    }

    if (Input.GetKey(name: "w"))
    {
        mov.y += 1;
    }

    if (Input.GetKey(name: "a"))
    {
        mov.x -= 1;
    }

    if (Input.GetKey(name: "d"))
    {
        mov.x += 1;
    }

    mov.Normalize();
    transform.Translate(translation: (Vector3)(mov * (speed * camera.orthographicSize * 2)));
}

```

In dieser Funktion wird die Kamera basierend auf den Benutzereingaben bewegt, indem die Eingabe überprüft wird und die Kamera entsprechend in die angegebene Richtung verschoben wird.

2.5.4 Game Pause

```
[SerializeField] public TextMeshProUGUI buttonText; 
public static PauseButtonScript instance;
 Frequently called  6 usages
public bool paused { set; get; }
[NonSerialized] public bool pauseCopy;
```

Hier erstellen wir eine öffentliche **TextMeshProUGUI**-Variable namens **buttonText**, die im Unity Inspector angezeigt und bearbeitet werden kann, zusätzlich eine statische Variable **instance** vom Typ **PauseButtonScript**, um auf eine Instanz der Klasse zuzugreifen.

```
public void SendPauseMessage()
{
    MessageManager.instance._writeMessage.WriteMessagePAUSE_REQUEST(!paused);
}

 Event function  Johannes Martin Ertle +1
private void Awake()
{
    buttonText.text = "Pause";
    if (instance is not null && instance.paused)
    {
        ChangePauseStatus();
        pauseCopy = true;
    }
    instance = this;
}

// Update is called once per frame
 Event function  sgc40
void Update()
{
    if(pauseCopy != paused) ChangePauseStatus();
}
```

Die Methode **SendPauseMessage** sendet eine Pause-Nachricht an den **Server**, um den Pausenstatus zu ändern. In der **Awake**-Methode wird der Text des **buttonText** auf **Pause** gesetzt, und wenn eine Instanz bereits existiert und pausiert ist, wird der Pausenstatus geändert und **pauseCopy** auf „true“ gesetzt. Die **Update**-Methode überprüft, ob **pauseCopy** und **paused** unterschiedlich sind, und ändert gegebenenfalls den Pausenstatus.

3. Tests

Die Tests dienen dazu, sicherzustellen, dass die Spielfunktionen ordnungsgemäß funktionieren und auf verschiedene Events und Zustände korrekt reagiert. Wir überprüfen, ob die verschiedenen Objekte ordnungsgemäß initialisiert und aktualisiert werden. Die Tests können ausgeführt werden, um sicherzustellen, dass keine Fehler oder unerwarteten Verhaltensweisen auftreten.

Beim Testen unterscheiden wir zwischen dem **Editmode** und dem **Playmode**. Im **Editmode** arbeiten wir in der Unity-Editor-Umgebung, um Szenen zu erstellen und zu bearbeiten, während der **Playmode** das Ausführen der Szene als eigenständige Anwendung ermöglicht, um das tatsächliche Verhalten der Anwendung zu testen und zu debuggen.

3.1 EditMode Test

```
public class PlayerTestEditMode
{
    [Test]
    // Johannes Martin Ertle
    public void Initialize_WithNameAndRole_SetsNameAndRole()
    {
        // Arrange
        GameObject playerObject = new GameObject();
        Player player = playerObject.AddComponent<Player>();

        string name = "TestPlayer";
        Role role = Role.PLAYER; // replace SomeRole with a valid Role

        // Act
        player.Initialize(name, role);

        // Assert
        Assert.AreEqual(expected: name, actual: player.name);
        Assert.AreEqual(expected: role, actual: player.role);
    }
}
```

Der Test überprüft, ob die Methode korrekt den Namen und die Rolle eines Spielers setzt. Zunächst wird ein **GameObject** erstellt und die Komponente Spielers hinzugefügt. Als nächstes wird der Name und die Rolle des Spielers definiert. Anschließend ruft **Initialize**-Methode mit den entsprechenden Werten auf. Die **Assert**-Anweisungen vergleichen den Namen und die Rolle des Spielers mit den erwarteten Werten und stellen sicher, dass die Initialisierung korrekt erfolgt ist.

3.2 PlayMode Test

Das Beispiel der **GameMangerTestPlayMode**-Klasse veranschaulicht den Ansatz der Implementierung einer beliebigen Testklasse:

```
[UnityTest]
Johannes Martin Ertle
public IEnumerator initializeGameTest()
{
    var boardConfig = JsonObjects.getBoardConfig();
    var participantsInfo = JsonObjects.getParticipantsInfo();

    StaticVariables.boardConfig = boardConfig;
    StaticVariables.participantsInfoMessage = participantsInfo;

    SceneManager.LoadScene("Scenes/Game");
    yield return null;
    var gameManager = GameObject.Find("_manager").GetComponent<GameManager>();

    Assert.NotNull(gameManager.allPlayers);
    Assert.NotNull(gameManager.ais);
    Assert.NotNull(gameManager.readyPlayers);
    Assert.NotNull(gameManager.spectators);
}
```

Für die Durchführung der Tests wird zunächst eine Instanz des **GameManager**-Objekts erstellt. **SceneManager.LoadScene** wird verwendet, um die Szene zu laden, und anschließend wird eine Wartezeit mit **yield return null** eingefügt, um sicherzustellen, dass der **GameManager** vollständig initialisiert wurde.

```
[UnityTest]
Johannes Martin Ertle
public IEnumerator onGameStateTest()
{
    var boardConfig = JsonObjects.getBoardConfig();
    var participantsInfo = JsonObjects.getParticipantsInfo();

    StaticVariables.boardConfig = boardConfig;
    StaticVariables.participantsInfoMessage = participantsInfo;

    SceneManager.LoadScene("Scenes/Game");
    yield return null;
    var gameManager = GameObject.Find("_manager").GetComponent<GameManager>();

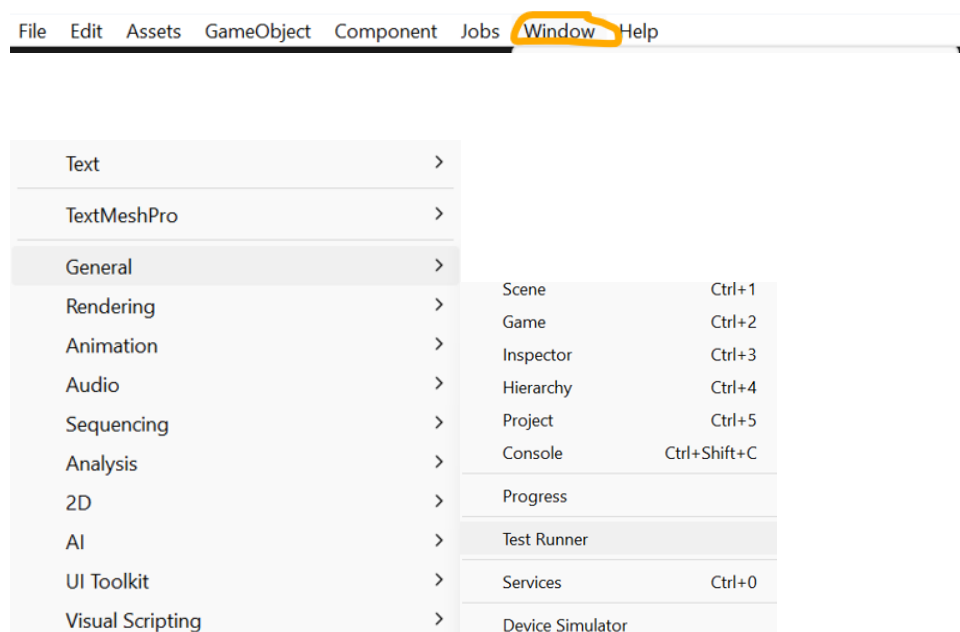
    gameManager.onGameState(JsonObjects.getGameState());
}
```

Nachdem der **GameManager** initialisiert wurde, werden verschiedene Methoden aufgerufen, um die entsprechenden Events zu simulieren. Beispielsweise ruft der Test **onGameStateTest** die Methode **onGameState** mit einem vordefinierten Spielzustand (**JsonObjects.getGameState**) auf. Ähnlich werden andere Events wie **onRiverEvent**, **onShotEvent** und **onCardEvent** simuliert.

```
Assert.NotNull(gameManager.allPlayers);
Assert.NotNull(gameManager.ais);
Assert.NotNull(gameManager.readyPlayers);
Assert.NotNull(gameManager.spectators);
```

Nachdem die Events aufgerufen wurden, werden **Assertions** (Behauptungen) verwendet, um sicherzustellen, dass bestimmte Bedingungen erfüllt sind. **Assert.NotNull** wird verwendet, um sicherzustellen, dass die verschiedenen Felder und Objekte im **GameManager** nicht null sind. Die Tests verwenden auch vordefinierte Konfigurations- und Informationsobjekte (**boardConfig** und **participantsInfo**), die aus der Klasse **JsonObjects** abgerufen werden.

3.3 Testing in Unity



Um die Tests zu testen, öffnen wir unser Unity-Projekt und stellen sicher, dass die Unity-Test Runner-Erweiterung installiert ist. Um zu testen, navigieren wir zu "Window" -> "General" -> "Test Runner":

Insgesamt dienen die Tests dazu, sicherzustellen, dass der **GameManager** ordnungsgemäß funktioniert und auf verschiedene Events und Zustände korrekt reagiert. Wir überprüfen, ob die verschiedenen Objekte und Felder im **GameManager** ordnungsgemäß initialisiert und aktualisiert werden. Die Tests können ausgeführt werden, um sicherzustellen, dass keine Fehler oder unerwarteten Verhaltensweisen auftreten.