ParkGame - program documentation

Park Game is a mobile multiplayer RTS game where the player becomes a **commander** on a battlefield with the ability to order units that follow him. The twist is that the **battlefield is mapped on a real park** and the player's **position is determined by GPS** (the game will be in some sense an **augmented reality game**). Players then will be divided into **teams** in which the players will **share units and structures**. Teams will then use those units to battle for **outposts** and most importantly **victory points**. There will be for simplicity **only a few units and structures**, the focus of the game should be on the **novelty of GPS controls** and cooperation within the team.

Dependencies

- Unity
  - Unity is a cross-platform engine, and it is appropriate to use for the development of a mobile game. It is quite easy to use, in addition, all members of our team have prior experience with it. That is why we have chosen to use it for our game.
- Unity Gaming Services
  - Unity Gaming Services is a specialized suite of tools and features tailored specifically for game development within the Unity engine. It encompasses a range of functionalities including multiplayer networking, analytics, monetization, and collaboration tools. These services are designed to streamline the game development process.
  - Out of which the application uses:
    - Unity.Netcode
      - Unity.Netcode is a multiplayer networking solution for Unity that enables real-time synchronization of a shared game session between players.
      - It makes it possible to host a game from the player's device, without having to run a dedicated server. The package mainly offers these tools to handle network communication: synchronization of variables, synchronization of game scenes and remote procedure calls. The package handles all lower levels of network communication.
    - Unity Authentication
      - Unity Authentication Service provides a framework for user authentication within Unity applications. It allows implementation of various authentication methods like email/password and social media logins.
      - Unity Authentication Service streamlines the authentication process, however, the application uses this service only out of necessity for Unity Lobby Service for which user needs to be signed in via Unity Authentication Service to make any calls using the Lobby SDK.
    - Unity Lobby Service
      - Unity Lobby Service is a component within applications, facilitating the creation of lobbies which players connect to. It provides functionality for creating lobbies, each associated with a unique code.
      - Moreover, this service offers a feature allowing players to join different teams based on the map structure.

            - In addition, Unity Lobby Service includes functionality for monitoring player disconnects, with a particular focus on the host's perspective. This is essential due to Unity Relay, used for host-client connections, lacking the capability to deliver disconnect notifications.

- NavMeshPlus
    - NavMeshPlus is used for navigation in a 2D environment. The package is an extension of Unity AI Navigation. In our game, soldiers use it for pathfinding in user-created maps with obstacles. Units follow their commander, or get close to an enemy to attack, using this library.
    - Prefabs **`Pawn`**, **`Archer`** and **`Molerider`** have component **`Agent`**. The agent component is set to be small (ca. 1/7 of the sprite) so the soldiers don't collide during their movement in formations. We had problems with this because the soldiers got stuck or blocked each other. The navmesh navigation does not work well for a lot of moving actors. With the smaller size of the agent, the soldiers can move fluidly to their positions.
- Firebase
    - Firebase for Unity is a comprehensive platform offering a suite of services aimed at backend development for Unity game developers. Firebase consists of several modules, each serving distinct purposes. This application takes advantage of these Firebase modules:
        - Authentication
            - The Authentication module within Firebase serves as a component for managing user authentication in Unity applications. Its primary function revolves around user identity verification. Additionally, Authentication facilitates the creation and management of user profiles.
            - In our app, Authentication is utilized for account creation and account management.
        - Realtime Database
            - The Database module in Firebase serves as a repository for storing and managing structured data in real-time. While primarily designed for storing metadata. Database is used in conjuction with other Firebase services, such as Storage, to provide a storage solution for larger files. In our app, Database plays a role in storing metadata for maps.
        - Storage
            - Firebase Storage serves as a solution for storing static assets such as images. Specifically, within the context of this application, Firebase Storage is utilized for storing static map images and layout drawings.

- Relay
    - When connecting directly using Unity.Netcode, there can be problems due to the player's firewall or network setup. Relay is a Unity service which solves this problem by connecting players through a middleman server. This can be at the cost of slower connection and higher costs, but it is useful for prototyping.
    - In the game Relay is used to establish connection between the players. Relay lets the host generate join codes which the players can use to join the host's room.
- Mapbox
    - The Mapbox Unity package is a toolkit that allows developers to integrate mapping and location-based services into their Unity projects. It provides access to Mapbox's mapping data and APIs, enabling users to create custom maps, incorporate location-based features, and tailor map styles to their application's requirements.
    - From Mapbox extensive feature set, the application uses:
        - Zoomable Map
            - Within our application, we primarily utilize the **`ZoomableMap`** scene from the Mapbox Unity package for map region selection. This scene allows users to interactively zoom in and out of the map to select specific regions of interest. These selected regions serve as templates for creating maps for our games.
        - Static Images API
            - Additionally, we leverage the Static Images API provided by Mapbox to generate snapshots of the selected regions. These snapshots serve as templates for creating the layout of game maps, providing a visual reference for developers during the map creation process.
- Free draw
    - Free draw is used for creating custom maps by the players. User draws with brushes of different colors into canvas which are then converted to playable maps.
- DOTween
    - DOTween is a lightweight animation library designed for Unity, offering a set of tools for creating smooth and dynamic motion effects.
    - In our application, DOTween is primarily employed for UI animations and small gameplay animations, enhancing user interface transitions and adding dynamic movement to gameplay elements.
- ParrelSync
    - ParrelSync is a tool that we use to test the multiplayer side of the game. It allows us to create clones of the game project, which can be run side by side.

Map Creation

This component is used for creating game maps by the players. Every player can create a new map in their desired location by drawing into the canvas which overlays a real geographic map. This hand drawn map is then converted into a game tilemap, saved into a database and available for other players.

- ## Map Selection
    - First the player needs to select a region in a map. Real geographical maps are provided by mapbox. Currently a free version of Mapbox is used which provides 50000 requests per month. Map selection is located in the scene **`ZoomableMap.`** Player can search on the map manually or by search bar. Map should also be focused on the current player's position. By clicking the select button the player can select a rectangular area for his map. Script **`MapSpriteBuilder`** takes care of recalculating selected area to map coordinates and saving them into the **`MapDisplayer`** component of **`mapSprite`**. This area is also checked if it is not too small (< 100 m^2) or too large (> 1km^2). Saved coordinates will be then used for fetching the selected map in **`MapDrawingScene`**.
- ## Free Draw
    - This package is used for drawing custom maps by the player. Setting up texture for drawing is handled in **`CreateMapWithOverlay`** script. Maps can be drawn in a scene called **`MapDrawingScene.`** When a map is fetched from the MapBox a new texture for drawing is created with the same dimensions as the fetched map. Resolution of the drawn map is lower for performance reasons. This is not an issue because the drawn texture is scaled even more later in the process (explained in Tile-map Creation). Scale of the tilemap is adjusted based on the real size of the map region, so larger regions in the real world have larger game maps.
    - There are 3 colors with different meanings in the map creation process.
        - Red - bounds which represent the outer bounds of the map, where units cannot go. This must be an enclosed area drawn on a map.
        - Blue - walls which represent obstacles in the walkable (inside of bounds) parts of the map.
        - Yellow - paths which increase speed of your units.
    - Uncolored space within the red bounds are walkable areas with no effects
    - Map can be also erased partially by erasing the brush (transparent color) or completely by resetting the canvas. Those functions are located in script **`Drawable.`**

- Structures
    - Player also needs to place structures into the map. There are 3 types of structures - Castles, Outposts and one Victory point.
        - Castles
            - Castles are base camps for each team, every team has to have one castle so the number of castles in a map determine the number of teams in the game. Castles generate new units. Map can have from 2 to 4 castles.
        - Outposts
            - Outposts are similar to castles but they are neutral in the beginning of the game and they can be conquered by other teams. Every map needs to have at least 1 outpost. Upper limit is set to 100 but in reality not that many outposts will be needed.
        - Victory points
            - Victory point can be only 1 on a map. If conquered by a team, this team gains a point needed towards a victory.
    - During the map creation process new structures can be placed by selecting them from the Draw options menu in **`MapDrawingScene`**. Each structure has its own item slot, which handles adding and removing structures to the map (script **`ItemSlot`**). Item slot takes structure item prefab (Prefab variants of **`MapStructureDraggable`**), which will be instantiated as a draggable object in canvas. Those structures have also script **`DragAndDrop`** for placing and moving them on the map. To remove structure from the map drag it on the appropriate item slot. For counting structures placed on the map there is a script **`StructureCounter`**, which checks the structure limits. Those limits can be set by the Max Structures option by each of the DragAndDrops game objects.
    - Castles are also color coded by the **`ColorSettings`** Scriptable object.
    - When structures are placed on the map and map processing is initiated, there is validation that structures are not too close to each other.
- Tile-map Creation
    - When the map is drawn and structures are placed, tilemap creation is initiated by pressing Process Tilemap. Function **`CreateTilemapFromTexture`** in script **`CreateMapWithOverlay`** is called. This function calculates the corners of the tilemap in tilemap coordinates and sets those tiles to recalculate tilemap bounds. Then the drawn texture is scaled down to resolution of the tilemap which results in low resolution texture.
    - 3 tilemaps are used:
        - baseTilemap - Used as an underlay walkable tilemap with grass or path texture
        - blockingTilemap - Used as not walkable area - outside of boundaries and walls

        - propsTilemap - Used as overlay for grass sprites on top of base grass texture, only visual feature
    - For each tile corresponding pixel in low resolution texture is checked and based on the closest color to red, blue, yellow or transparent, a corresponding tile is assigned. Euclidean distance for difference measurement is used. Correct tilemap is selected by the tile type. When all tiles are assigned, flood fill from top left corner of blocking tilemap is initiated to fill the outside region with bounds tile. If the area is not enclosed, an error is displayed. Then some grass foliage sprites are placed randomly onto the map. The random generator is seeded for a particular map always with the same seed so it is consistent. Then all structure prefabs are placed into the map. There are checks if the structures are accessible or not too close to each other. If no errors are found, the created tilemap is displayed to the user and can be saved.
- Database
    - The Firebase database is used to store information about created maps.
    - Saving map is implemented in `MapDataFirebaseManager` script.
    - Saved map has 2 parts:
        - image - low resolution image of the drawn map (in tilemap resolution)
        - map metadata
            - MapId - unique map id
            - MapQuery - mapbox query to fetch the full resolution map image
            - MapName - map name
            - Longitude - longitude coordinate of the map
            - Latitude - latitude coordinate of the map
            - Width - width of the drawn texture
            - Height - height of the drawn texture
            - Structures - 3 lists of structure positions in tilemap coordinates
            - TopLeftTileIdx - top left index in tilemap coordinates
            - BottomRightTileIdx - bottom right index in tilemap coordinates
    - From those data can be the game map recreated. The process is the same as described above for creating map or loading existing map.

Screens/Scenes

- Organization

In the Unity game, the scene organization is structured to manage different aspects of gameplay and user interaction. Here's a summary of the scene organization:

    - Menu Scene:
        - This scene encompasses all screens related to the game's main menu.

        - Screens are implemented via **`UIController`**, **`UIPage`**, **`UIPageController`** which allow for flow of screens and their logic in a single scene.
    - Screens include the title screen, login/signup screen, main menu with options like creating/hosting or joining a lobby, and miscellaneous screens for settings or information.
    - It serves as the hub for navigating various sections of the game, excluding map creation functionalities.
  - ZoomableMap Scene:
    - This scene features a draggable and zoomable real-life map.
    - Players can interact with this map to select regions for their gameplay maps.
    - It provides an interface for choosing specific areas or locations to be incorporated into the game.
  - MapDrawingScene Scene:
    - Dedicated to facilitating map editing features.
    - Allows players to draw or modify elements on the game map.
    - Provides tools and functionalities for customizing map layouts according to player preferences.
  - GameScene Scene:
    - This scene handles the core gameplay of the game.
    - It includes all elements necessary for the game mechanics, player interactions, and progression.
    - Manages game logic, player controls, AI behavior, and other gameplay-related features.
  - CleanUpScene Scene:
    - Responsible for resource cleanup after a game session concludes.

- UI

The following section is about screen management which is heavily used in Menu scene UI.

  - UIController

The **`UIController`** class is a component in managing the user interface (UI) navigation within the application. It is used in scenes with multiple UI screens, such as a main menu with various submenus and pop-ups.

It is usually held by Canvas GameObject in the scene.

- The class maintains a stack of **`UIPage`** objects, representing the navigation history. This stack-based approach allows for complex UI flows where pages can be pushed onto the stack (shown) and popped from the stack (hidden) in a controlled manner. This is particularly useful for

managing navigation between different screens in the UI, such as moving from a main menu to a settings page and back.

- The **`initialPage`** field is used to specify the first page that is displayed when the application starts. The **`firstFocusItem`** field is used to set the first UI element that should receive focus when the application starts.
- The **`PushUIPage`** method is used to display a new page. It pushes the page onto the stack and handles the transition animation. If a page is already being displayed, it will be hidden or removed based on its **`ExitOnNextPage`** property which specifies if the page should be removed from the page stack on new page push.
- The **`PopUIPage`** method is used to hide the current page. It pops the page from the stack and handles the transition animation. If there are any pages left on the stack, the topmost page will be displayed.
- The **`ShowPopUp`** method is used to display a pop-up message. It creates a new pop-up screen, sets its properties, and pushes it onto the page stack.

○ UIPage

The **`UIPage`** class is a component for managing individual screens within a scene in Unity, particularly in scenarios where there are multiple different UI screens, such as in a main menu with various options, settings, and submenus.

It is held by the respective GameObject that the **`UIPage`** controls.

- Each instance of **`UIPage`** represents a single screen or page in the UI. It contains methods and properties that control how the page is displayed and hidden, including animations and sounds.
- The **`Prepare`** method is used to set up the page for animation based on its **`entryMode`**, which determines how the page will be animated when it is displayed.
- The **`Enter`** method is used to display the page with the appropriate animation, based on the **`entryMode`**. It also optionally plays a sound when the page is displayed. The **`entryMode`** can be set to various options such as **`None`**, **`Fade`**, **`SlideRigh`**t, **`SlideLeft`**, **`SlideUp`**, **`SlideDown`**, or **`Zoom`**, each triggering a different animation.
- The **`Exit`** method is used to hide the page with an animation, based on the **`exitMode`**. It also optionally plays a sound when the page is hidden. The **`exitMode`** can be set to the same options as **`entryMode`**.
- The **`onEnter`** and **`onExit`** actions are triggered when the page is displayed and hidden, respectively. These actions can be used to perform

additional tasks when the page is displayed or hidden, such as updating the UI or saving data.

○ UIPageController

The **`UIPageController`** class serves as a blueprint for managing logic of individual screens in a Unity application. As an abstract class, it outlines a common structure for all UI pages.

It is held by the respective GameObject that the **`UIPage`** controls.

Classes that inherit from **`UIPageController`** typically encapsulate the logic specific to individual screens. For instance, a **`UIMainMenuScreenController`** inherits from **`UIPageController`** and implements the logic specific to the main menu screen, such as handling button clicks or initializing menu items.

- The **`OnEnter`** and **`OnExit`** methods in **`UIPageController`** are designed to be overridden in these subclasses. They act as hooks that subclasses can use to inject their own behavior when a page is entered or exited.
- The **`OnEnter`** method is called when a page becomes active. Subclasses might override this method to initialize UI elements, start animations, fetch data, or perform any other setup required when the page is displayed.
- The **`OnExit`** method is called when a page is about to be deactivated. Subclasses might override this method to clean up resources, stop animations, save data, or perform any other cleanup required when the page is hidden.

● Screens

This is a brief description of the major screens in the Unity project. All these can be found in their in their respective directories in **`/Assets/UI/Pages`** subfolder.

Scenes like ZoomableMap and MapDrawingScene consist of one screen only and therefore are handled by their own scripts and not by the **`UIController`** class.

○ Title Screen

The **`UITitleScreenController`** class is a controller that manages the title screen of the application.

Here's a brief description of the key methods that are tied to the buttons on the title screen:

- **`Enter`**: This method is the main method that is called when the **`enterButton`** is clicked. It plays a click sound effect, disables the **`enterButton`**, and attempts to sign up and log in to Unity Auth.

Depending on whether the user is already signed in to Firebase, it pushes either the **`mainMenuPage`** or the **`welcomePage`** onto the UI stack. Finally, it makes the **`enterButton`** interactable again.

Services are started via **`ServicesManager`**.

○ Sign Up Screen

The **`UISignUpScreenController`** class is a controller that manages the sign up screen of the application.

Here's a brief description of the key methods that are tied to the buttons on the login screen:

- **`SignUp`**: This method is triggered when the user clicks on the "Sign Up" button. It attempts to create the user with the provided username, email and password. If the sign up is successful, it transitions to the main menu screen. If the login fails, it changes the color of the text field to indicate an error.
- **`Back`**: This method is triggered when the user clicks on the "Back" button. It transitions back to the welcome screen.

Signup is facilitated via the **`ServicesManager`** class.

○ Login Screen

The **`UILoginControlle`**r class is a controller that manages the login screen of the application.

Here's a brief description of the key methods that are tied to the buttons on the login screen:

- **`Login`**: This method is triggered when the user clicks on the "Login" button. It attempts to log the user in with the provided email and password. If the login is successful, it transitions to the main menu screen. If the login fails, it changes the color of the email and password input fields to indicate an error.
- **`Back`**: This method is triggered when the user clicks on the "Back" button. It transitions back to the welcome screen.

Login is facilitated via the **`ServicesManager`** class.

○ Main Menu Screen

The **`UIMainMenuController`** class is a controller that manages the main menu screen of the application. It is responsible for handling user interactions with the main menu, such as button clicks, and transitioning to other screens.

Here's a brief description of the key methods that are tied to the buttons on the main menu:

- **`Create`**: This method is triggered when the user clicks on the "Create Map" button. The specific implementation of this method involves transitioning to a screen where the user can create a new map.
- **`Host`**: This method is triggered when the user clicks on the "Host Game" button. The specific implementation of this method involves transitioning to a screen where the user can host a new game (specifically Prepare Game Menu Screen).
- **`Join`**: This method is triggered when the user clicks on the "Join Game" button. The specific implementation of this method involves transitioning to a screen where the user can join a lobby via lobby code.
- **`About`**: This method is triggered when the user clicks on the "About" button. It transitions to the "About" page, providing the user with information about the game or application.
- **`SignOut`**: This method is triggered when the user clicks on the "Sign Out" button. It signs the user out from the application, typically returning them to a welcome or sign-in screen.
- **`Exit`**: This method is triggered when the user clicks on the "Exit" button. It quits the application.

○ Prepare Game Menu Screen

The **`UIPrepareGameMenuController class`** is a part of the game's user interface (UI) that prepares the game menu.

Here's a summary of its methods:

- **`OnEnter`**: This method is called when entering this UI page. It disables interactivity, sets the map picker to non-interactable, resets the processing state, and starts downloading maps.
- **`OnExit`**: This method is called when exiting this UI page. It deletes the maps from the map picker.
- **`Back`**: This method is called when the back button is clicked. It plays a click sound effect and navigates to the main menu page.
- **`Create`**: This method is called when the create button is clicked. It plays a click sound effect, disables interactivity, and starts the process of creating a lobby for the selected map. If the lobby creation is successful, it

navigates to the lobby page. If it fails, it logs a warning and re-enables interactivity.

`MapPicker` class facilitates map thumbnail downloads and their subsequent display to the UI for the player. It handles the data reconstruction by downloading all metadata from Firebase Database and then reconstructs each thumbnail by pulling map static images from Firebase Storage.

○ Lobby Menu Screen

The `UILobbyMenuController` class is a part of the game's user interface (UI) that manages the lobby menu. It extends the `UIPageController` class and controls the behavior of the lobby menu where players can see the room code, their team, and other players in the game. The host can start the game from this menu.

Here's a summary of its methods:

- `OnEnter`: This method is called when entering this UI page. It downloads the map data if it's not available and initializes the UI with the map data. It also sets up the listeners for lobby invalidation and disconnection.
- `OnExit`: This method is called when exiting this UI page. It clears the room code and map name labels, destroys the team UIs, and removes the listeners for lobby invalidation and disconnection.
- `UpdateUI`: This method updates the UI based on the current state of the lobby. It sets the visibility and interactivity of the start game button, updates the room code label, and updates the team UIs.
- `InitializeUIwithMapData`: This method initializes the UI with the given map data. It sets the textures and colors of the drawn and GPS images, sets the map name label, and initializes the team UIs.
- `isGameValid`: This method checks if the game is valid based on the given map data. A game is valid if each team has at least one player.
- `InitializeTeamUI`: This method initializes the team UI for the given team number.
- `Back`: This method is called when the back button is clicked. It plays a click sound effect, disables the back button, leaves the lobby, shuts down the network manager, and re-enables the back button.
- `StartGame`: This method is called when the start game button is clicked. It plays a click sound effect and starts the game.
- `OnDisconnect`: This method is called when the player is disconnected. It navigates to the main menu page.

- In Game Screen

The **`UIInGameScreenControlle`**r class is a part of the game's user interface (UI) that manages the in-game screen. It extends the **`UIPageController`** class and controls the behavior of various in-game buttons and actions.
It controls screen flow in GameScene between the In Game Screen, Options Screen, How to Play Screen and End Game Screen.

Additionally, it displays players units and the outpost structure that the player is currently holding. Their UI elements are implemented via classes **`UIUnit`** and **`UIOutpost`** classes respectively.

- Pop Up Screen

The **`UIPopUpScreenController`** class is a part of the game's user interface (UI) that manages pop-up screens. It extends the **`UIPageController`** class and controls the behavior of pop-up screens, including the title, message, and dismiss button.

It serves only as a screen which is filled out by the **`UIController`**'s method **`ShowPopUp`**. Calling this method fills the screen with defined data and subsequently displays the pop up to the screen.

Gameplay

Multiplayer

- Lobby Manager

The **`LobbyManager`** class serves as the central component for managing lobby-related operations in a Unity project. It utilizes the Unity Lobby SDK to facilitate multiplayer lobby functionality. Below are the main methods and functionalities provided by the **`LobbyManager`**:

- **`CreateLobbyForMap`**: This method creates a lobby for a specific map. It sets up lobby parameters such as name, maximum number of players, and team allocation. It also interacts with the Unity Lobby SDK to create the lobby asynchronously.
- **`JoinLobbyByCode`**: Allows a player to join a lobby using a lobby code. It retrieves necessary lobby data, subscribes to lobby events, and sets up network configurations for joining the lobby. It also establishes a client host connection via Unity Relay.

- `LeaveLobby`: Handles the process of a player leaving a lobby. It removes the player from the lobby, updates lobby information accordingly, and cleans up any associated resources.
- `JoinTeam`: Enables a player to join a specific team within the lobby. It verifies if the team has available slots and updates the player's team information accordingly.
- `StartGame`: Initiates the start of the game once the lobby is ready. It prepares the game scene and relevant data before transitioning to the game scene.
- `DownloadMapData`: Downloads map data required for the game. It retrieves map metadata and images from Firebase Storage and Database.
- `GetPlayerData`: Retrieves player data based on Firebase ID or client ID. It is used for obtaining information about players within the lobby such as a team number.
- `onConnectionApproval`: Handles connection approval requests from the network manager. It manages client connections to the lobby and ensures that only authorized players can join.
- `HandleLobbyHeartbeat`: Manages the heartbeat functionality for the lobby. It sends periodic heartbeat pings to the lobby server to maintain the lobby connection and ensure its stability.
- `OnLobbyEventConnectionStateChanged`: This method is triggered when the lobby event connection state changes. It receives a `LobbyEventConnectionState` parameter indicating the current state of the connection. If the state transitions to `Unsubscribed`, it signifies that the subscription to lobby events has been stopped, possibly due to a disconnection. In response, the method invokes the `OnDisconnect` action to notify other parts of the application about the disconnection. It also displays relevant UI updates to inform the player about the connection status.
  `OnDisconnect`: This action is invoked when a player disconnects from the lobby. It can trigger various actions or UI updates, such as showing a disconnection message, prompting the player to reconnect, or handling cleanup tasks.

In general, the LobbyManager class encapsulates lobby-related functionalities essential for creating, managing, and participating in the multiplayer lobby within our game. It also holds information about players, like their name. It integrates with the Unity Lobby SDK and Firebase.

- Game synchronization
  - Being a multiplayer game, the state has to be synchronized among all the clients. Normally we use the server authoritative model, meaning that if the client wants to do an action which needs to be synchronized, they have to ask the host to do

it. The only point where this is broken are the positions of the commanders, because the positions have to come from the client's device anyway. Otherwise inputs are handled as in the following example. The **`PlayerController`** receives inputs like changing the formation from the player, and sends them to the host. The host then remembers the player's chosen formation. Other units are then simulated on the host, and behave according to the chosen formation. Any events that happen on the host that the client has to know about are communicated through remote procedure calls. Any properties which need to be shared, like units health points for example, are synchronized with the clients through Unity's **`NetworkVariable`**.

GPS

The **`GPSLocator`** class is a part of a Unity application that manages the GPS location services. It extends the MonoBehaviour class and controls the behavior of the GPS location services, including initialization, updates, and checks.

Here's a summary of its methods and properties:

- **`initializationTimeout`**, **`invokeDelay`**, **`updateRate`**: These are configurable parameters for the GPS location service.
- **`Longitude`**, **`Lattitude`**, **`HorizontalAccuracy`**: These properties provide access to the last known GPS location and its accuracy.
- **`OnLocationInitialized`**: This is an event that is triggered when the location service is successfully initialized.
- **`Start()`**: This method initializes the location service if the GPS permission is granted.
- **`Initialize()`**: This method starts the initialization of the location service if it's not already running.
- **`InitializeLocator()`**: This is a coroutine that handles the initialization of the location service. It checks and requests the location permission, starts the location service, waits for it to initialize, and starts updating the location if the initialization is successful.
- **`UpdateLocation()`**: This method updates the last known GPS location and its accuracy each **`updateRate`** seconds.

The users need to grant GPS permissions to be able to use **`GPSLocator`**. These permissions are checked on each application focus in method **`OnApplicationFocus(bool hasFocus)`** in **`ServicesManager`** class.

Match

- ## Game Initialization
    - The game has to be initialized before it can begin. After the match is started, the hosts wait for all the clients to load their scenes. This is done in the **`PlayerManager`** class. After that, the host loads the map and spawns all its structures through the **`Map`** class. At this point, the host also spawns commanders for the players as instances of **`PlayerController`**, the commanders are then placed on their teams' castles. Then, the host waits until all the players have walked close to their castles in the real world. Once they have, it starts a countdown. At the end of the countdown, the input for commanders is unlocked, and they follow the players' real world positions. Lastly, initial three basic units are spawned in all castles, and the castle starts spawning units over time. At this point the game has been fully initialized and can begin.

- ## Commands
    - The player can gather more soldiers, command them to attack or fallback, or move in some formation. For this, a column of three buttons is located in the panel below the game map.
    - ### Attack/Fallback
        - If the button for Attack is clicked (crossed swords), the following soldiers get ready to attack enemy units. A red icon appears above their heads to visualize this state. The icon on the button changes to a white flag.
        - The button can be toggled to fall back to return to the previous formation around the commander.
    - ### Gather Soldiers
        - If there are less than 15 soldiers following the player's commander, soldiers can be called to the player's army.
        - The soldiers that can be called to follow have to be newly spawned and cannot be following any other commander. They also have to be of the same affiliation as the commander.
        - The following soldiers are listed in the panel Units below the map, next to the command buttons.
        - The newly added soldiers take over the commands of other soldiers following the commander (i. e. follow in Box formation, or be ready to attack).
    - ### Formations
        - The third button is for formatting the soldiers. There are three types of formations: Box, Free, and Circle. The toggle button always shows the formation that is the next in order: after clicking on the formation button, the soldiers will go to stand in that formation.

    - Calling the soldiers into a different formation overrides their Attack command (if the last command was Attack), and the Attack button changes to crossed swords.

Command buttons

- Capturing Outposts
    - The map contains some outposts that start without team affiliation, but are capturable. After a team captures the outpost (by standing on it for a while with some units) it starts generating units for the team (the type of unit generated by the outpost can be changed in the UI). This provides the team a place where to pull more soldiers from.
    - More on the capturing outposts and spawning units in the chapter “Gameplay Prefab”.
- Soldier Stats
    - The stats cannot be changed during the game. But it is easy to change them in the Unity project for a new build of the game, for example after playtesting for balancing. We chose these numbers to reflect our idea about the soldier types.

    - Swordsman is available right from the start. The swordsmen spawn on the uncapturable castles. We chose HP and cooldown time randomly from the start, and they served as a comparison when assigning the stats of other soldiers. Damage is 1 because it is the default soldier. The range of attack should match the sprite size and animations (same for the Moleriders).
    - The archers and riders are special soldiers because they spawn only in outposts. They deal more damage but have some disadvantages to balance it out.
    - Archers shoot from their bows/slings. They have a longer range than others. It takes some time to take an arrow/rock, notch it, and shoot it, so the cooldown time is a bit bigger than for the Swordsman. They have an advantage because they deal damage from a distance. To balance it, they have less HP. The attack range is finetuned based on playtesting, and it matches the distance for the automatic arrow shooting from castles (to prevent spawn kills).
    - Moleriders are brave and charge into a battle head-on while the others catch up with them later. Because they tend to be overwhelmed by the number of enemy

soldiers, they have a big HP, so they don't die immediately and can deal some damage. The cooldown time between attacks is longer, so they don't have more advantage over others.

	hp	**damage**	**cooldown**	**range**
Swordsman	7	1	1	0.12 - 0.67
Archer	4	2	1.2	1.50 - 3.00
Molerider	8	3	1.5	0.23 - 0.67

- Victory Point
  - Victory point is a structure in the game, which once captured adds a point to the team that captured it. It is the main objective of the game to capture the points. The class **`VictoryPoint`** is responsible for handling this logic, and keeping track of teams' scores. It uses the **`ConquerorModule`** class to determine which team is conquering it. When the victory point starts to get conquered, it announces it through the **`Announcer`**. It also announces that the victory point will open soon (10 seconds in advance), and that it has opened.

Gameplay Prefabs

- Conqueror Module
  - Conqueror module is a prefab that should be a child object of something that can change teams (the object must contain a **`MonoBehaviour`** script that fulfills the interface **`IConquerable`**).
  - This module works by using **`Rigidbody2D`** and **`CircleCollider2D`**, to detect all units (not commanders) that are near them. The radius of the conqueror module can be adjusted by changing the parameters of the **`CircleCollider2D`**.
  - The logic of the **`ConquererModule`** script is in its **`Update`** method where if there are only units of one opposite team it slowly adds up the **`_ConquerPoints`**. If there are units from more than one team the system doesn't award any **`_ConquerPoints`**, but doesn't subtract them either. And if there are only units that are not owned by the conqueror, the **`_ConquerPoints`** are reset to zero. If there are no units at all, it drops to zero gradually.
  - When the **`_ConquerPoints`** reach the **`ConquerPointsRequired`** variable (that is adjustable in editor), the team change of the parent structure is invoked.
  - *note: more units don't gain points faster than less (unless it's zero)*

- Bubble Progress Bar
    - The bubble progress bar is a prefab that displays the rough progress of something (in our case it shows the progress of the conqueror module). To accomplish this it uses a list of its sprite child-objects (in our case they are circles arranged into a larger circle). The bubble progress bar then colors a percentage of the list based on the **`Value`** and **`MaxValue`** variables.
- Outpost
    - Outpost is a building that starts with neutral team affiliation and contains a conqueror module as a child object, so it can be conquered by players.
    - Once the outpost becomes part of some team it starts generating one specific type of unit for that team (this type can be changed in the user interface), but it does so only if there aren't 3 units on more units already present. Also conquering stops the outpost from generating units. The spawning of a unit happens every 10 seconds and is done using the **`ToggleSpawner`** script.
    - *note: units are responsible for reporting who they are following, so the outpost script only implements `ICommander` interface and stores those units in a list.*
- Castle
    - Castle is a special case of outpost and even shares with the outpost the **`Outpost.cs`** script, but it has following notable differences.
        - It starts with a team affiliation and can't be conquered (even lacks the conqueror module).
        - There is exactly one castle for each team.
        - Generates only one type of a unit (Swordsman), but stops generating them only if there are 6 units present in the castle (unlike the outpost that stops at 3).
        - If there are any enemy units within the range of the castle it shoots at them arrows using the **`ShootScript`** script.
- Victory Point
    - The victory point is a building like an outpost or castle. It is present on each map exactly once and handles the victory condition of the game.
    - The victory point contains a conqueror module just like the outpost, but starts with this module deactivated. After 60-100 seconds (chosen at random) the victory point opens (the conqueror module is activated). After any team conquers the victory point it gains one point (stored in the **`teamScores`** list) and then the victory point closes again for another 60-100 seconds.
    - If at any point one of the teams giant 3 points, the team is victorious and the game ends.
- Enemy Observer (EnemyDetector)
    - **`EnemyDetector`** is a prefab that is attached to units and holds a list of all enemy units that are nearby and list of all friendly units that are close by. It accomplishes this using **`RigidBody2D`** and **`CircleCollider2D`** components. Those

components provide the **EnemyObserver** script with **OnCollisionEnter2D** and **OnCollisionExit2D** methods.
    - **EnemyObserver** class also contains quality of life methods like **GetClosestEnemy**, **GetRandomEnemy** and more.
    - **EnemyObserver** is used in all units so they can easily detect all enemy units nearby and the commander uses it to call upon units that are not already following him.
- Arrow
    - It is spawned by **ShootScript** which is placed on **Castle** prefab and **Archer** prefab.
    - The arrow has **Arrow** script as its component. It is given a goal position (which is the position of an enemy) as an argument to function **Shoot**.
    - The script is concerned with moving the arrow to the goal position.
    - Function **Hit** gets called after the arrow arrives at the destination set during its firing. This function calls **Physics2D.OverlapCircleAll** to gather colliders in a small radius (0.1 Unity units) from the hit position. If a collider belongs to an enemy soldier, the soldier takes damage. If one enemy gets hit, no other soldier gets hurt. The arrow destroys itself afterward.
    - It can play a sound effect upon hitting the goal.
- Soldiers
    - There are three types of soldiers in the game.
    - Each soldier has hp, cooldown time for attack, and attack range (described in the section Soldier Stats).

    - The units are spawned from prefabs **Pawn** (for Swordsman, with script **Soldier**), **Archer** (with script **Archer**), and **Molerider** (with script **Molerider**).
    - The scripts implement the soldier's behavior. They are inherited from the **SoldierBase** script which implements the functions for movement and attack, taking damage and dying, with triggering appropriate animations (section Effects/Animations) and calling the **SoundManager** for pooling the audio clips (more in the Sound Design section).
    - The inherited scripts differ from the superclass mainly in the **GetEnemy** function. The Swordsmen attack on the closest enemy. The Moleriders run first to the farthest enemy they see, after the enemy is no more, they attack on the closest enemy next to them.
    - The **Archer** overrides the function **AttackEnemyIfInRange**.
    - The prefabs have **EnemyDetector** for enemy vision and **Agent** for navmesh navigation. The **Archer** prefab also has **ShootScript** for spawning and shooting arrows/rocks at enemies.

- Soldier Commands
  - The soldiers have three possible types of behavior, commands they act upon. The behaviors are implemented by scripts `SoldierBase`, `Soldier`, `Archer`, and `Molerider`.
  - `InOutpost` is the default behavior for newly spawned soldiers in castles and outposts. Each soldier stationed in an outpost defends that outpost in some radius (1.5 Unity units). The soldier goes to attack enemies while keeping his distance in this set defense radius from the outpost.
  - Another soldier's command is `Following`. The soldier updates his `Agent`'s goal position (from NavMeshPlus) in `Update()`. He can follow two types of moving objects: his commander's position or a position in a current formation around the commander. The player calls his soldiers to follow when toggling the formation button or calling for a retreat on the panel under the map.
  - The `Attack` command calls the player by toggling the attack button. Each soldier checks `EnemyDetector` for enemies. He starts attacking if there is an enemy within his `AttackDistanceFromCommander` (6 Unity units). If the enemy is outside his attack range (described in section Soldier stats), he first gets closer. Otherwise, the soldier acts as in the case of `Following`.

- Commander
  - This prefab is used for the representation of a player.
  - Soldier Select
    - In the beginning, each player had to tap on a soldier sprite (belonging to the same team) to add it to his army. The soldier could also be deselected and sent to the closest outpost by another click on his sprite. But because the game is played on a mobile phone and tapping on the soldiers was often problematic - as soldiers can be standing close to each other and selecting multiple could also deselect some of them - we decided to change the whole system for this selection. Instead of tapping on the soldiers, we added a button for gathering unassigned soldiers in close range around the player.
    - The prefab has an `EnemyDetector` component, but the component is for listing friendly soldiers belonging to the same team as the commander. When the player taps the button for gathering soldiers, the function `Gather` in his `PlayerController` (also on this prefab) is called.
    - Each commander has a set limit of 15 soldiers in his army at once. The `Gather` function first controls how many soldiers are following the

commander. If there is still a place for adding new followers, friendly soldiers are listed from the `EnemyDetector` component. A soldier then can be added to the army if he is not following some other commander. The commander takes all the free soldiers in his small radius up to the capacity of 15 units.
        - Each new follower automatically takes over the commands of other following soldiers. To explain it in an example, if the rest of the soldiers are in a box formation, the newly added soldier gets assigned to a position in that formation. If the soldiers get the command to attack, the new soldier is also ready to attack in the same formation as the rest of the retinue.
        - The called gather command is also highlighted by a visual effect. It is an animation of scaling and fading a circular sprite of the player's team color up to the size of the `EnemyDetector` on the commander. That way the player can see how close he needs to be to the soldiers so they can hear his command and start following him. This effect is created by `DOTween` sequences to fade in the color to 80 % of the alpha in 0.3 s, then fade out to 10 % of alpha in 0.6 s. While scaling the colorful sprite to the max size in 0.3 s and then hiding it to size 0 in the next 0.6 s.
    - Path Speedup
        - When the commander is on a path or nearby (3 tiles away), all of the units get speed boost.
        - Speed boost is 1.5x their normal speed.
        - This logic is implemented in the script **`PathTileChecker`**. The script simply checks the surrounding of the commander and returns if any path tile is nearby or not.
    - Formations
        - Soldiers have several types of formations they can move in. In our game, we implemented two with more internal logic and also a Free formation for more disorderly movement on the map.
        - Each soldier's `Agent` goal position is updated to some object's position - commander's position in Free formation, position in a formation in Circle and Box formation.
        - The Free formation is the most direct. Follow the commander's position. Because this simple condition directed all the soldiers to the same place, we now generate a random number (in the range of 0.2 to 1.2 Unity units) while spawning the soldier. This number is the range up to which the soldier goes to the commander. He can stop moving if he is in this range from the commander. This way the problem of never stopping the soldiers - because they were all trying to get to the same one position and constantly colliding with each other - was solved.

- The other two types of formations (Circle and Box) are more complex. The type of soldier (`Swordsman`, `Archer`, `Molerider`) plays a part in the soldier positioning in the formation.
- First, we will describe the rules for soldier placements in the Circle and Box formations. Then the logic behind it will be explained in more depth.
- The Circle formation is three concentric circles around the commander. The `Arches` are placed in the innermost circle (with the least hp and the farthest attack range), followed by `Swordsmen`. The much faster `Moleriders` are running in a counterclockwise fashion around these circles in the largest circle.
- The Box formation is a rectangle following behind the commander. Each row is made of three soldiers (`Swordsmen` and `Archers`). On both sides of this rectangle, `Moleriders` ride. Because the `Moleriders` are faster than the rest of the soldiers, they can run into the battle sooner without disturbing the rest of the formation.
- The **`Commander`** prefab has the scripts for managing the formations, with helper objects for positions in Box and Circle formation. The Box formation follows the commander and rotates as he moves on the map.
- After the player commands the soldiers to some formation, each soldier following him gets assigned to a position in that formation. **`CircleFormation`** and **`BoxFormation`** are implementing the **`IFormationType`** interface. The **`Formation`** script has references to both scripts and calls the functions according to the currently selected formation. It is the starting point for a soldier getting a position in a formation.
- Positions are instantiated from a prefab where a **`PositionDescriptor`** is its component. It keeps information if the position is assigned to some soldier or is free. If the soldier dies, the position gets unassigned. Switching the formation type resets the formation, creates new positions for the following soldiers, and assigns them.
- The Circle formation creates positions on a circle. As the soldier count grows, the positions start resembling the circle more and more - from a line to a triangle, square, etc. Each type of soldier has a set radius in advance around the commander (**`Archers`** ~ 0.53, **`Swordsmen`** ~ 1.06, **`Moleriders`** ~ 1.73 Unity units). If a soldier in some position dies, his position gets destroyed and the circle adjusts to fill the gap. The **`Moleriders`** follow positions on a circle that rotates around the commander.
- The Box formation creates rows of three positions. These positions get `Archers` and `Swordsmen` on a first come, first served basis. The `Moleriders` get positions on the left and the right around these rows of three. If a soldier dies, the position gets unassigned (not destroyed as in

the Circle formation). Newly added soldiers go first into these vacated positions.
    - The position objects are created only on the host and invisible to the players. For testing purposes, we render them with the material of the team's color.

Effects

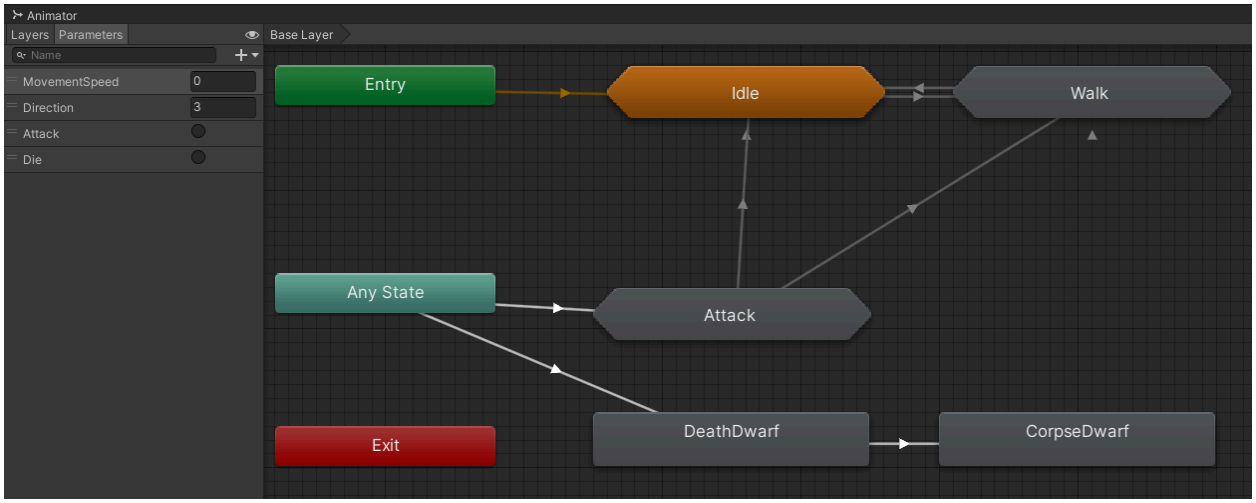
- Color Swap
    - In the game, the teams have their specific color and it should be obvious which team is the unit member of. For this we change the color of the dwarfs hat to have the team's color. For this we have implemented the color swap shader which when applied to the material keys out the base red color of the hat and boots and changes it to the color of the teams color.
    - Color swap shader in Unity's basic shader with modified `fixed4 frag(v2f IN): SV_Target` function. In this function, we compare the input color and compare it to the color we want to replace (the red color of the hat) by measuring distance between those 2 vectors. If the distance is close enough we return our desired color, if not we return the original one. However this alone isn't enough, because the dwarf beard is quite similar in color to the red hat. What we in fact do is that we add the condition that the input color must also have a smaller distance between itself and the hat color than between itself and the color of the beard. Finally we have to modify the output color so the dwarf hat isn't one single color and has some shades. We do that by multiplying it with the sum of the values of the input color divided by 3 and then multiplied by our brightness constant.

- Fog of War
    - The game has a fog of war system, which prevents players from seeing units and structures of other teams, if they aren't close enough. The fog of war is set up in the following way. All the friendly units are rendered with the normal `ColorSwap` shader. All enemy units are rendered with the same shader, but with stencil condition to render only if there is a 1 written in it. All the friendly units have a child with a `Revealer` component and a square sprite rendered with the `Revealer` shader. This sprite is rendered as first in the sorting order, and writes a 1 into the stencil buffer for the rendered pixels. This reveals the enemy units. The fragment function makes sure to clip the pixels, such that the shader creates a pixelated circle around the unit. The size of the blocks matches the size of the tiles on the map. The pixels which aren't clipped are set to fully transparent black, but because they are rendered, they write a 1 into the stencil buffer. The last

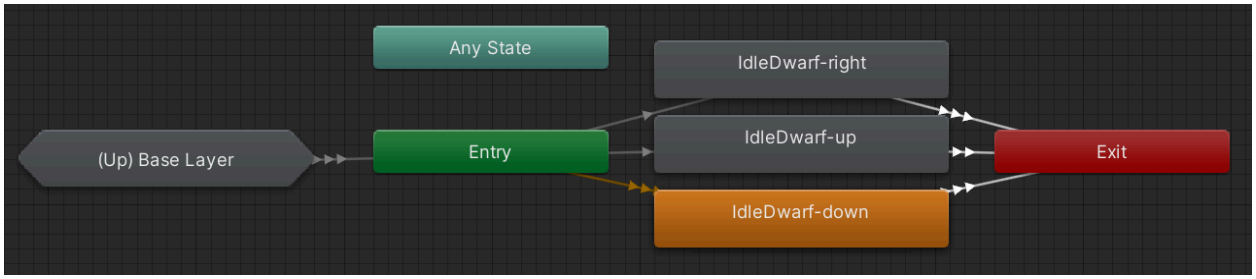
piece is a black semi transparent sprite over the whole game field, which renders all the pixels for which the value in the stencil buffer is not 0. This way, the parts where the black overlay is not rendered, are the parts where the enemy units are visible to the player, giving the impression of fog of war.

- Animations
  - The game uses Unity's **`Animator`** and animations. The animator is basically a state machine with the nodes representing the animation that is being played on repeat. These states are connected by edges that represent transitions from one state to another based on set conditions. Parameters for these conditions can be bool, int and float. All of them can be changed in code, thus influencing the state of the animation. Moreover the state can be influenced by triggers, which are events that can also be called from code. Triggers are best used for actions that happen once before transitioning to some more common state (when you're about to shoot one arrow you don't want the animation to be played more than once). Unity’s animator also allows for using substate machines which allows us to group up states so they can have a singular entry and exit point.
  - Animations that we use are simple sprite animations that switch between a few sprite key frames on a loop. We have implemented animations for our 3 units and a commander. Each unit has unique idle, walk and attack animation for 3 sides front, back and right side (left side is accomplished by mirroring the sprite in code) and all units share the death animation. The commander is similar to the unit, but lacks death animation and the attack animation is replaced with the wave animation.
  - In our animator, we use variables for movement (float), direction (int) and two triggers for attack (/ wave) and die (commander doesn’t have this one). For better organization we use 3 substate machines for attack, walk and idle animations. Those submachines only decide which direction the unit is facing. Using this we can have much simpler transitions. For the idle and walk submachines it also holds, that the have transitions also back to themselves, if the dwarf at any point changes the direction that he is facing.

Dwarf animator



Idle submachine



- Soldier Status Drawer
  - Each soldier has a small icon above their head to depict the current command. It is visible to all players. So the players know if other players want to attack or are just passing around in peace. However, this can change in a matter of moments, so the players should be vigilant and prepared to call a new command.
  - The script `UnitBehaviourDrawer` is placed on each soldier prefab. A new soldier's command invokes an event of command change. This script then listens for the events and shows icons for `Attack` and `InOutpost` commands.
  - `InOutpost` is a small castle. `Attack` icon is a pair of red crossed swords.
  - The InOutpost means that the soldier is not yet following any player/commander and can be called to arms. After the soldier starts following a commander, he cannot return to this state anymore.
  - When a soldier is called to arms, he confirms it by a mumbling sound effect and a speech bubble shows above his head for a second. We call a coroutine for timing, and after it ends, it gets the current command from the soldier and displays it.

Sound Design

For sound, we use the Unity Audio system. The **`AudioManager`** script is a singleton and manages everything sound-related. It plays one-shots from the correct locations, mutes or unmutes the sources, and resets the sound settings upon returning to the menu.

In the menu, there are no sound settings. Buttons play a clicking sound.

In the game, there are two types of sounds - notifications and sound effects (SFX). They can be individually muted or unmuted in the pause menu.

Notifications are mono - the audio listener and the source are placed on the player.

- Trumpet commands: there is only one trumpet player in the commander's army. That means that only one sound can be played at once. Otherwise, fade out the playing sound and start the new one. Each command triggered by a button has its trumpet melody (Attack, Gather, Box/Circle/Free Formation). Fallback commands the soldiers into a previously selected formation, with the appropriate melody playing. As the melodies differ for each command, the soldiers could go into attack or some formation upon hearing it. It also gives the player audio feedback on the command selected.
- Countdown starting the game, notifications concerning the victory point (opening, start of conquering).
- Happy and sad chimes mark whether something is gained or lost (outpost, victory point, game).

Sound effects are stereo - the audio source moves to the sounding object, and players with earphones can distinguish the place of origin. It doesn't account for the orientation of the player respective to the source, though. Each type of SFX has several variations from which a sound clip is randomly picked and played.

- Sounds that dwarves make like murmured affirmations signifying following the player, sounds of attack (clanging, flight of arrows, rock hitting a target), death.
- The player hears SFX only in a limited range, with linear falloff.

Pooling of SFX (script AudioPool) is needed to limit the number of currently playing sound effects. There are six audio sources dedicated to SFX. Each time there is a request for playing an SFX clip, measure the distance between the source and the player. Do not play the sound if the source is further than the player can hear. Otherwise, check whether there is an unused source to play the sound.

The base sounds for attack are from freesound.org. More variations were created by adjusting the pitch and speed. Sword attack (Sword clash 1.wav by spycrah, Classic Bike Bell Misfire 1 by manofham), shooting arrows (Regular Arrow Shot with rattle by brendan89), rock impact (cut and adjusted pitch and speed of G39-19-Parachute Swish Snap.wav by craigsmith). The rest of the sounds were created for the game. The sound clips have normalized volume. Sound effects play at 80 % of the volume, while notifications have 100 %.

Graphics

- Fonts
  - Title screen font was made by Adien Gunarta and can be freely download here https://www.dafont.com/wizzta.font
  - The default font was made by Sebastian Kosch and it can sourced here https://fonts.google.com/specimen/Crimson+Text
- Tiles
  - For tilemap there is a free tileset (https://cainos.itch.io/pixel-art-top-down-basic) used. Because the set has a lot of variations, there is RuleTile used for all of the tile types. The most advanced is the **`PathTile`**, which uses 11 different cases based on neighboring tile locations to create naturally flowing paths. To add even more variety there are also multiple options within a single case to choose from.
  - For tiles **`RandomGrass`** and **`RandomBoundGrass`** there is a single case with multiple tiles which are randomly sampled to break the symmetry and repetitiveness of the map.
  - The **`StoneTile`** is used for walls and it has 2 types of tile sprites. Stones and artifacts. The rules are defined in a way so the artifacts are on the edge of the tile area and stones can be anywhere. There are also multiple choices of stones and artifacts.
  - To add additional detail, there is a **`FoliageTile`** with multiple sprites of grass stems which are also randomly sampled and mirror horizontally.
- Unit sprites
  - Unit animation sprites were drawn by Votěch Pinc specially for this project.
- Background image
  - Image was made by upklyak and can be used freely with attribution according to the license at https://www.freepik.com/free-vector/cartoon-forest-background-nature-park-landscape_8067055.htm
- Dwarf images
  - Dwarf images that appear on title screen and main menu were generated via https://starryai.com/
- Buttons and panels
  - Images used for buttons and panels are part of a free Unity asset made by bonk! https://assetstore.unity.com/packages/2d/gui/rpg-fantasy-mobile-gui-with-source-files-166086

Build

- ## Access keys

The repository will not hold all access keys for packages which the application uses. It is advised to

  - create a custom project in Firebase Console (https://firebase.google.com/),
  - Mapbox account (https://www.mapbox.com/) and
  - custom project in Unity Service (https://docs.unity3d.com/Manual/SettingUpProjectServices.html) with an account to Unity Cloud (https://cloud.unity.com/).

- ## Packages

Before building, ensure that Mapbox (https://www.mapbox.com/unity) and Firebase (https://firebase.google.com/docs/unity/setup) modules Auth, Database and Storage are imported. The repository already includes a copy of Mapbox but Firebase needs to be reimported.

For Mapbox importing the **`Mapbox`** folder from the root of the package should suffice. Importing other folders containing AR assets can prevent successful builds due to deprecated technology of Mapbox AR.

- ## Setup
  - ### Mapbox

    For Mapbox to properly function the package needs a key to Mapbox API. The key can be retrieved from your Mapbox account. The package then accepts the key through the top panel menu **`Mapbox > Setup > Access Token`**.

    Mapbox and Firebase use different build systems for Android which can lead to conflicts when building the same classes. If Gradle errors show up during the build delete **`*-25.1.0.aar`** files in **`Assets/Mapbox/Core/Plugins/Android/`** according to this Github issue (https://github.com/mapbox/mapbox-unity-sdk/issues/1564)

  - ### Firebase

    Ensure that the file **`google-services.json`** which can be downloaded via Firebase Console is present somewhere in the **`Assets`** folder (e.g **`Assets/Firebase/`**). Also ensure that the package name entered in the Firebase project matches the Android package name in Unity project (which can be changed in Unity through the top panel menu **`Edit > Project Settings`**

`> Player` then under Android tab `Identification > Package name`. For more information consult Firebase Unity setup documentation at (https://firebase.google.com/docs/unity/setup).

- Unity Services

  Unity Services need to be activated in Unity Cloud. Please follow guidelines for
  - Player Authentication - https://docs.unity.com/ugs/en-us/manual/authentication/manual/get-started
  - Relay - https://docs.unity.com/ugs/en-us/manual/relay/manual/get-started
  - Lobby API - https://docs.unity.com/ugs/en-us/manual/lobby/manual/get-started

To build the application use Unity Build System. If the steps above were followed the project should successfully build. If not, also ensure that Unity builds the Android application with Java 11 SDK.