Chimera Design Document

by Adrian C

This document will take you through the project structure and its design, detailing how it was built.

The project took 7 days to complete.
Apart from Asset Store models & FX, everything was built from scratch in Unity.

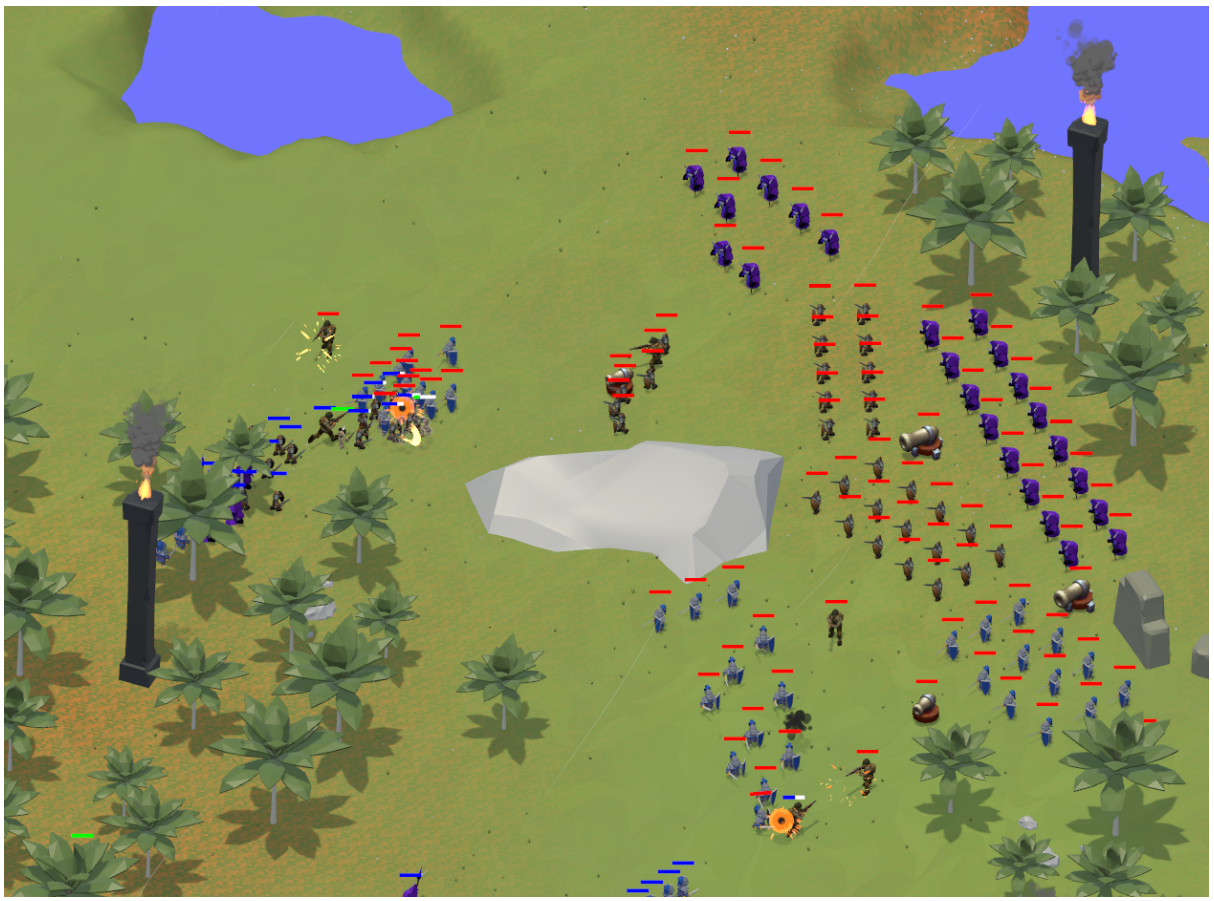
Unity version: 2021.3.24f1

GitHub Repository: https://github.com/whoadrian/Chimera

Windows Build: Check the Builds folder.

Contents:

3 .... How to start & Controls
4 .... Goal of the Game & Unit Types
6 .... Balancing & Design
6 .... Project Structure
7 .... Game State, Game Config & Main Menu
7 .... Levels
8 .... Actors
10 .. AI & Behaviour Trees
11 .. Combat & Weapons
12 .. Selection & Commands
13 .. Camera
14 .. Object Pooling



How to start

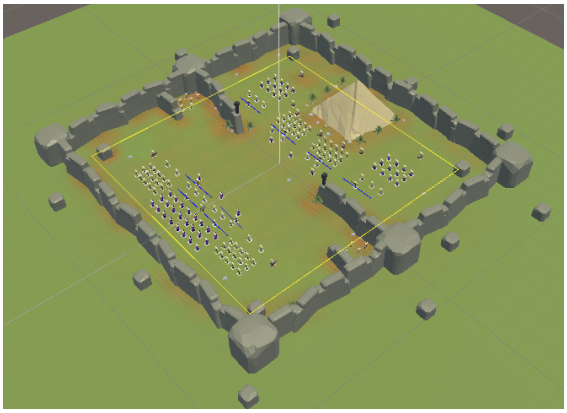
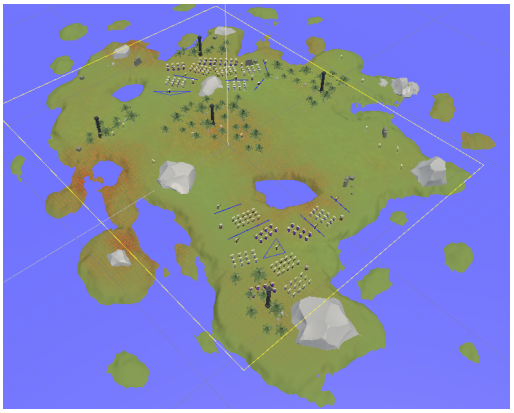
If you have opened the project in Unity, launch the scene called ***Init***, at ***Assets/Scenes/Init***

If you're on *Windows*, you can play the executable game from the ***Builds*** folder, at the root of the repository.

The game start in the Main Menu, which will take you through several screens, where you will select:

- The level: **Island** or **Desert**
- Faction: **Red** or **Blue**

Depending on your choices, the corresponding level will be loaded:

Controls

Camera:

- Movement: **WASD**
- Rotation: **right-click + drag** or **QE**
- Zoom: **scroll wheel** or **RF**

Units:

- Selection: **click + drag**
- Move command: **right-click on empty land**
- Attack command: **right-click on enemy unit**

Pause Game & Resume:

- **Escape key**

Goal of the Game

In order to win, the player needs to destroy all enemy units.
Note: Turrets are excluded from the win/lose state. This is because the turrets cannot move and the game can reach a deadlock state where only turrets are alive.

Unit Types

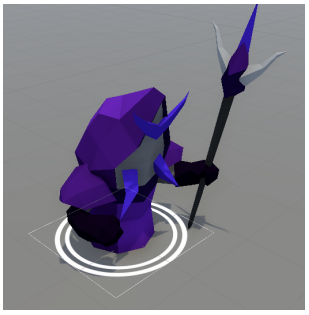
- **Melee_01**
  - The knight is a bit slower, but has extra life and good damage. Useful to use in the first lines of battle!

- **Melee_02**
  - This unit is a bit faster than the previous one and deals rapid damage, but its health is not so good! Manage carefully and you will not lose all.

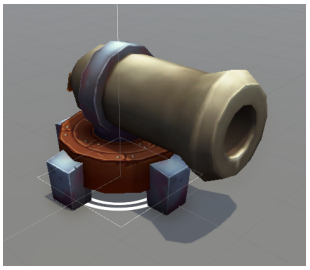
- **Melee_03**
  - The skeleton belongs to the **Green** faction and can be found spread across the levels. It's health and field of view are not as good, and, actually, is pretty bad at everything.

- **Range_01**
    - The soldier has a long field-of-view and attack range, shoots one projectile at a time and deals a good damage. Is very good at patrolling and guarding things.

- **Range_02**
    - The wizard can match the soldier, but what makes it very special is that it shoots 3 projectiles in one move, covering a wide area. Very useful for fighting large armies.

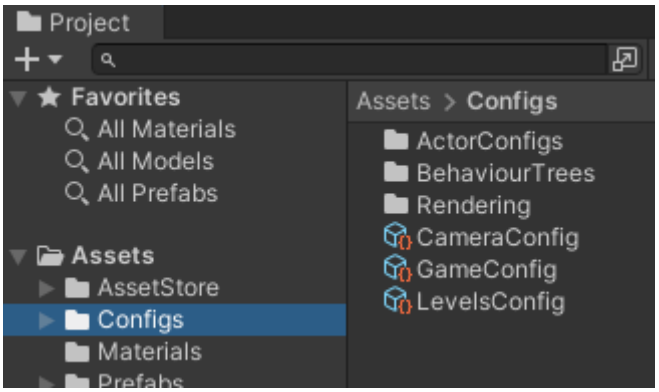


- **Turret_01 & Turret_02**
    - The turrets can deal a great damage, but, sadly, cannot move. They have a sturdy health, though! They are excluded from the win / lose state.

Balancing & Design

The goal of the project is to be as modular and designer-friendly as possible. All balancing of the game can be made from the **Configs** folder - from unit stats, behaviour trees, combat settings and camera controls, most data is stored in scriptable objects as read-only data and used throughout the levels or referenced in prefabs:



Project Structure

The project bundles together assets that are similar in format, per folder.
Here is a list of folders at the root level of the project:

- **AssetStore**
  - Contains all assets that have been downloaded from the Asset Store. They are the actor prefabs, FX particle systems, materials and textures.
- **Configs**
  - Contains scriptable object instances. Most of the game balancing can be made from here, as well as selecting the player faction (GameConfig), camera settings (CameraConfig) and level data.
- **Prefabs**
  - Contains prefab variants of the asset store models and their animation controllers, as well as prefabs used for setting up the levels, such as cameras & UI.
- **Scenes**
  - Contains all scenes: Init (used for starting the game), the menus and the levels.
- **Scripts**, **Terrains, Materials & Textures**

Game State, Game Config & Menus

The **GameState** script manages the *Play / Pause / Quit / MainMenu* state:

- It loads the appropriate scenes depending on the state.
- It is initialized in the **Init** scene, and is statically made available to other scripts. There should only be one instance of it at a time.

The **GameConfig** scriptable object

- Determines the player faction. When launching a level directly from its scene (ex: Level_Island), this config determines the faction of the player.
- Stores the **Level** index, using Unity's **PlayerPrefs**. This way, the game remembers the last played level, acting as a 'Saved State'.

The **Main Menu** lives in its own scene and is loaded by the **GameState**.
The **Overlay Menu** (paused game) also has its own scene, and is loaded additively over the current level. When it's loaded, the time scale is set to 0, so the game pauses.

Levels

There are 2 levels in the game. Each level is loaded additively over the **Game** scene.

- **Level_Island**
- **Level_Desert**

They can be played individually by starting the scene directly in Unity. The player faction will be determined by the **GameConfig** scriptable object instance in the **Configs** folder.

Each level has a **Nav-Mesh & Terrain**, and is split into 'folders' in the hierarchy, separating the **Characters** from the **Environment**, camera **Spawnpoints**, **UI**, etc.

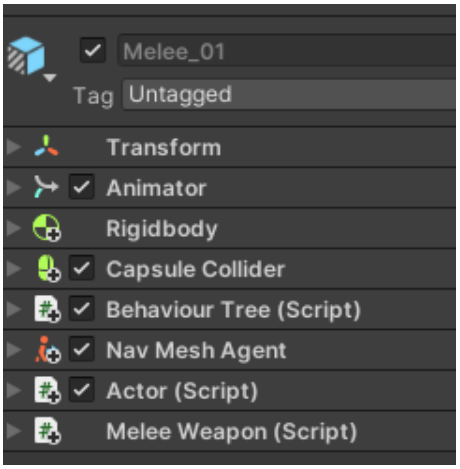
Each level contains a **Level** prefab, which has the **Level** component. This component checks the win / lose state everytime an actor dies. The actors are responsible to register themselves to the **Level** instance. This component also contains various data about the levels, such as the level bounds used by the camera.

The levels are managed by the **LevelManager** script, which uses the **LevelsConfig** scriptable objects to asynchronously load the scenes, including the Win / Lose scenes. The **LevelManager** is initialized when the **Game** scene is loaded, and is available to all levels, by setting it to **DontDestroyOnLoad.**

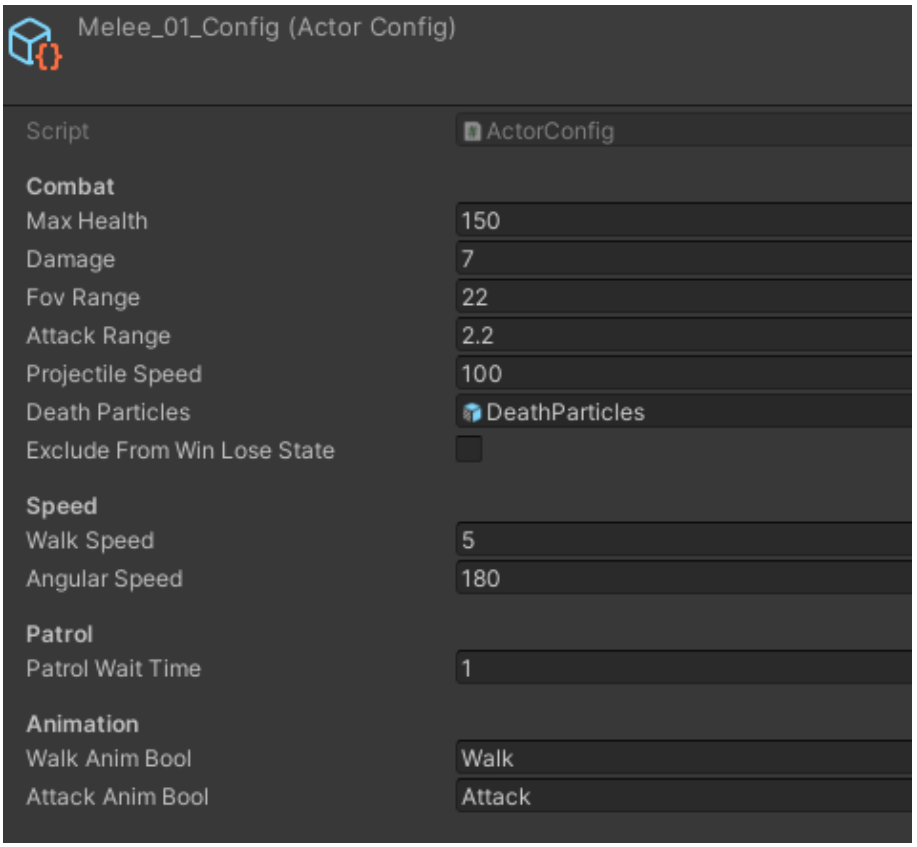
Actors

The **Actor** class represents all playable units in the game. It uses the **ActorConfig** scriptable object for its settings, which has several instances in the **Configs** folder, for the different unit types.

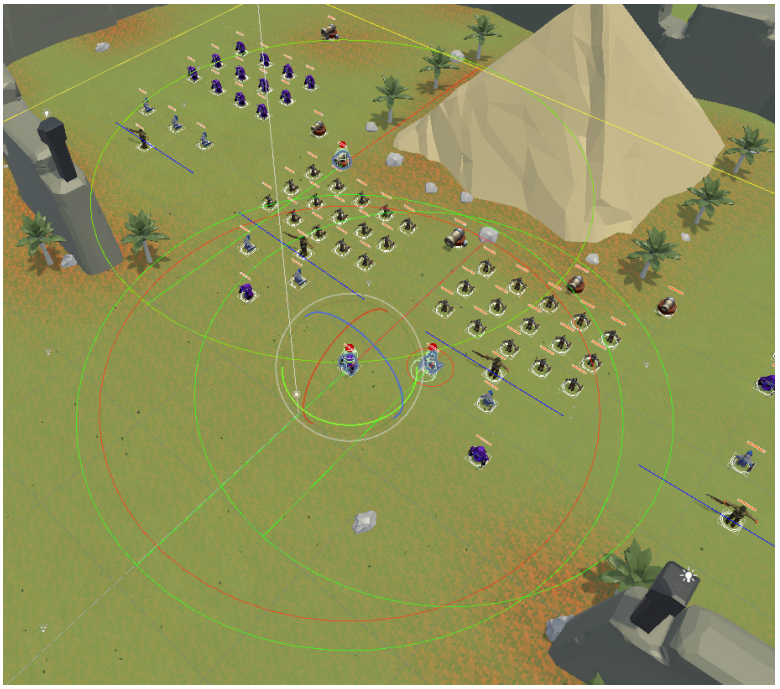
Here is the Melee_01 Prefab:



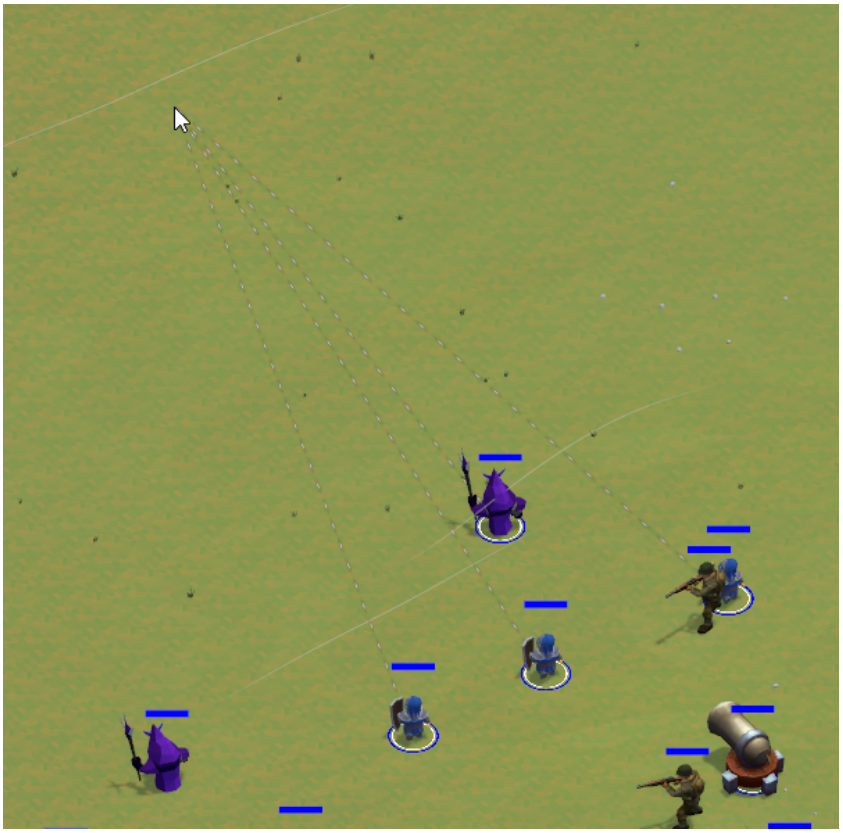
And its Melee_01 Actor Config:



Debugging actor config values is done via Gizmos, in the editor scene. Here, you can see the field-of-view and attack ranges of the units when selected in the Scene view:



Visualizing the selected actors and their destination path is done via the **ActorHUD** prefab, in-game:

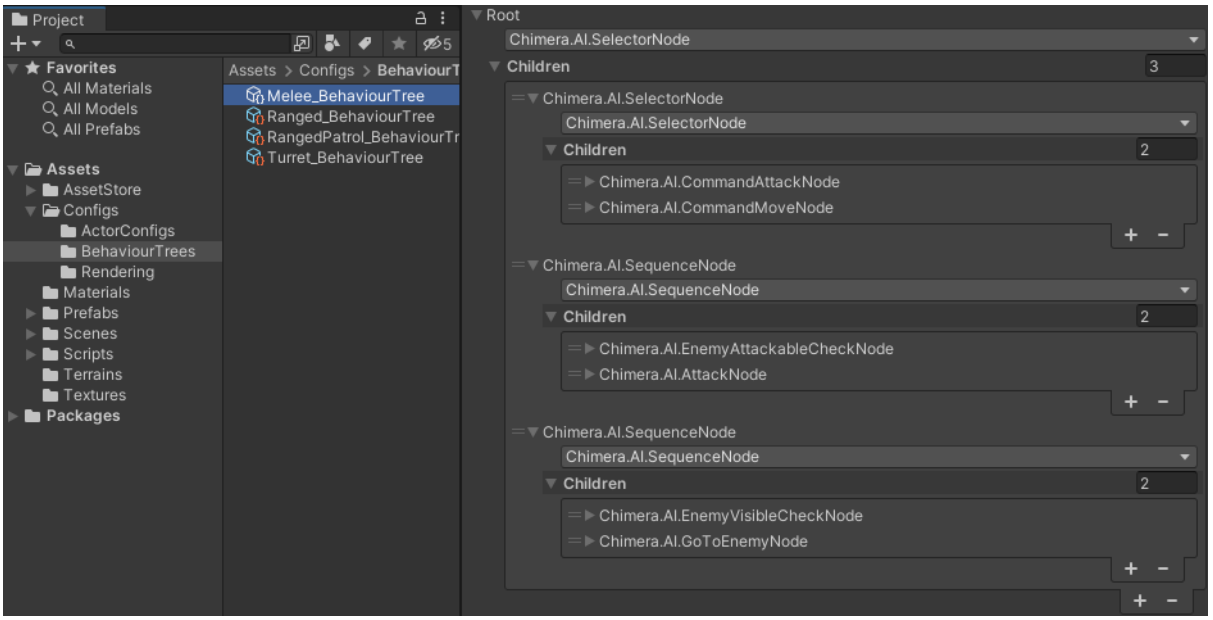


AI & Behaviour Trees

For the gameplay mechanics of the units, a Behaviour Tree approach has been implemented. This modular approach to building behaviours is widely used and is extremely flexible, as each node in the tree can logically be separated from the rest.

To make it designer-friendly, the trees can be built using the **BehaviourTreeBlueprint** scriptable objects.

Here you can see the blueprint for the Melee units of the game:



This blueprint is referenced from the Actor prefabs. When the **BehaviourTree** component is initialized, it uses this blueprint to build the tree and evaluate it every frame.

Each dropdown is a custom property drawer (see **TypeDropdownAttribute**), which uses reflection to scan for all types inheriting from the **Node** class, and lists all available nodes.

<u>The order of the nodesin the tree determine their priority</u>.
The player-commands (attack & move) are first, followed by other nodes:

- **EnemyAttackableCheck** : checks if enemy is within the attack range
- **EnemyAttack** : triggers the attack anim, which in turn activates the weapon
- **EnemyVisible** : finds the closest enemy within fov range, if none already
- **GoToEnemy** : triggers the walk anim, set the agent destination to enemy
- **Patrol** : uses the PatrolWaypoints component to patrol between locations
- **StopMovement** : stops the navmesh agent

There are 2 other simple, but important nodes that allow complex behaviours:

The **Selector Node**:

- This node acts as an **OR** gate, going through each children until one succeeds. The player-commands (move & attack) are parented to such a node, as only one player command can exist at a time.

The **Sequence Node**:

- This node acts as an **AND** gate, trying to run all nodes until all are succeeding, in sequence. For example, checking first if an enemy is visible, then going to the enemy, then attacking the enemy.

The tree implements a generic **Dictionary<string, object>** shared between the nodes, so that they can communicate about enemies, and other context-dependent data. Boxing & unboxing of data can occur here, and is a tradeoff for flexibility.

The designers can therefore have creative freedom over the behaviours of the units, relying on developers only when new node types should be implemented.

To improve performance, each **BehaviourTree** registers itself to the **BehaviourTreeRunner**, which evaluates all trees in its Update method, avoiding Unity callbacks on each individual **BehaviourTree** component.

Combat & Weapons

Each Actor implements the **ICombatant** interface, which exposes details about the health of the unit, and can receive damage, as well as being healed.

This **ICombatant** interface is used throughout the game for checking health and displaying it in the Actor HUD, as well as for making sure that only the units of the player's faction can be selected.

Weapons:

There are 2 types of Weapons implemented:

- Ranged Weapon
- Melee Weapon

Each of them implements a Unity Animation Event callback - the **ActivateWeapon** method. The **MeleeWeapon** activates a **DamageCollider**, while the **RangedWeapon** spawns a pooled **RangedProjectile** at a specified spawnpoint.

This callback is referenced in the actors' attack animation clips.

Selection & Commands

The selection and player-issued commands is done via the **PlayerControl** script & prefab.

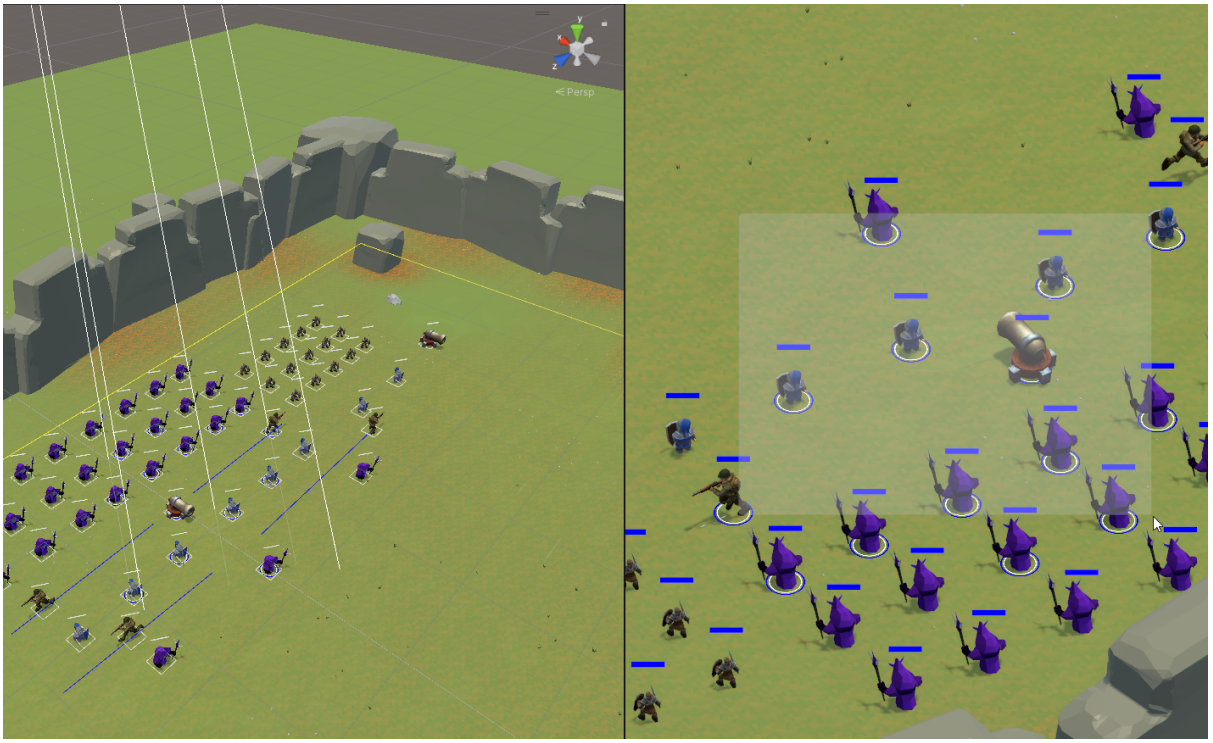
Selection:

- **PlayerControl** creates a **RectTrasform** when the mouse is clicked & dragged. Upon released, a box structure is created extending from the screen onto the terrain, which is used to detect **Colliders** that implement the **ISelectable** interface.

Attack & Move commands:

- When right-click is released, a similar approach to selection (see above) is done, only that now, if a unit of a different faction is detected, an **attack** command is issued - otherwise a **move** command. This command is sent to all currently selected units, via the **IControllable** interface, implemented by the **BehaviourTree** class. The **Move** command tries to place all units in a grid.

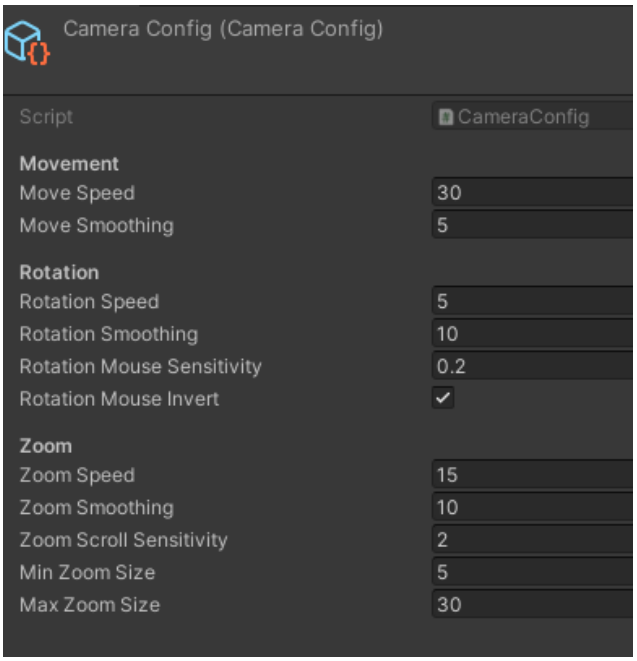
The selection logic can be visualized with Gizmos in the Scene View. Here you can see the selection box edges (left) and the selection rectangle UI (right):



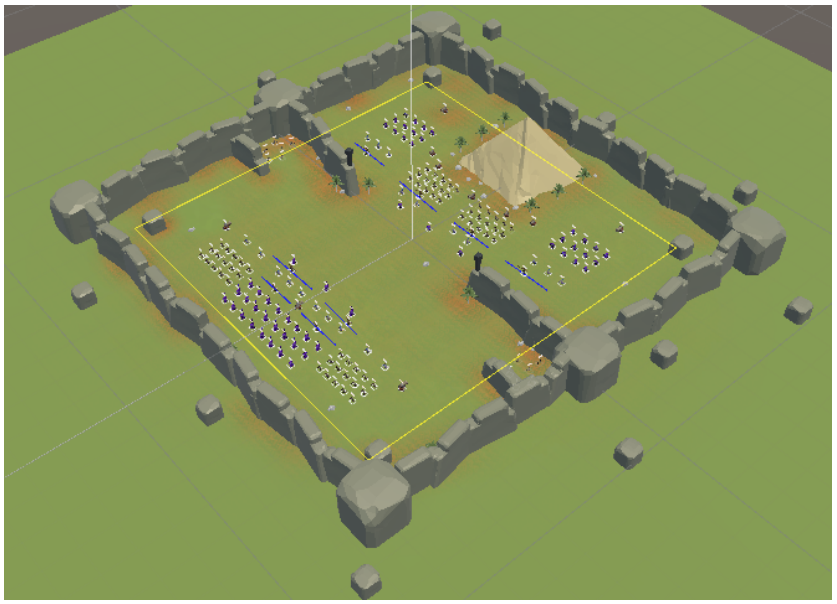
Camera

The camera mechanics are implemented via the **Camera** script & prefab. A reference to this prefab is available in all levels.

The **CameraConfig** scriptable object instance is available in the **Configs** folder and can be easily edited and tweaked at runtime:



The camera position is limited by the level bounds, if a **Level** instance exists.
The **camera bounds** can be visualized in the Scene View as the yellow rectangle:



Object Pooling

The **ObjectPool** singleton, together with the **IPoolable** interface ensure object pooling for objects that are spawned & destroyed frequently, such as FX and projectiles.

All **FX particle systems**, **RangedProjectile** instances and **DamageArea** colliders implement the **IPoolable** interface and are being spawned via the **ObjectPool**.

Each pool has its own parent in the scene hierarchy, named after its prefab instance.

- If such a pool already exists, it returns the first inactive object.
- If no inactive objects are found, it creates a new one.

Furthermore, each pooled object has a **timestamp** and is destroyed after a number of seconds (currently 20 seconds) after it was last used.

Here are the ObjectPool instance and its children pools during a combat scene: