SuperTiles

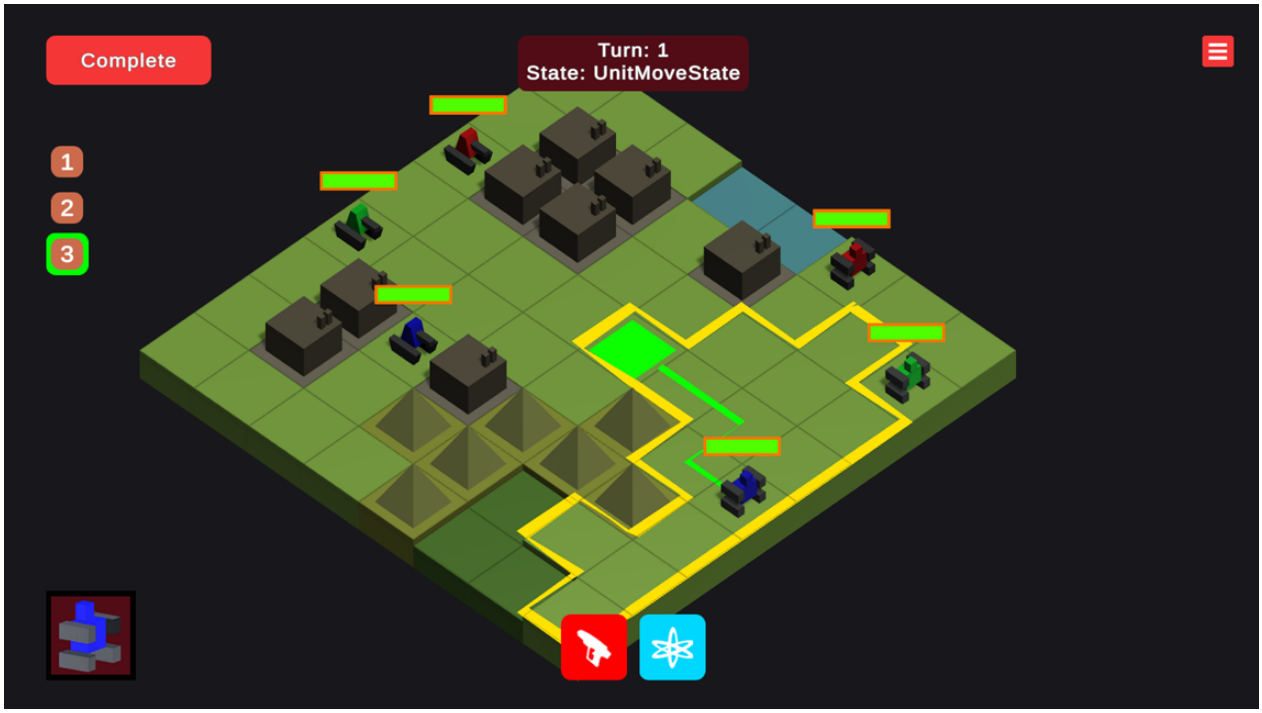
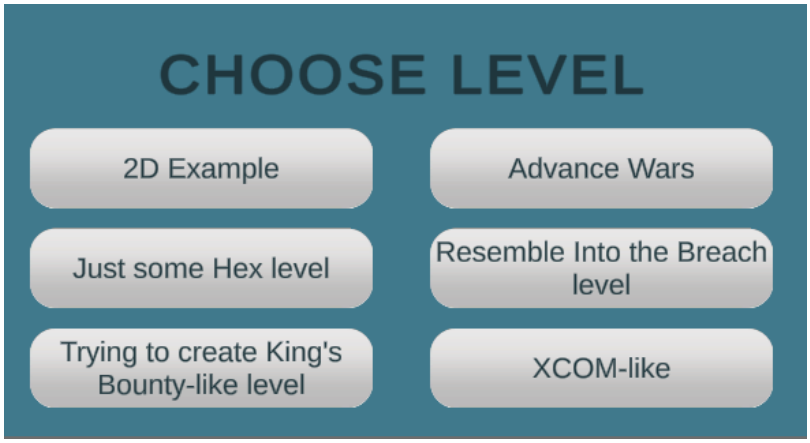


Table of contents

Overview

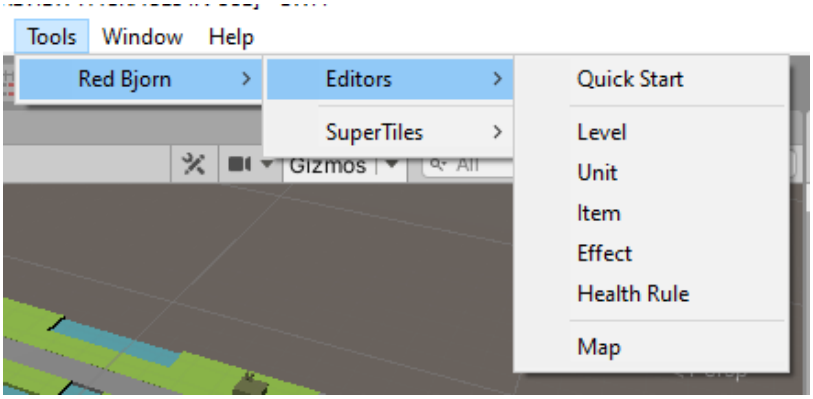
This asset is developed to speed up creation of turn-based games.

To have a better representation of asset functionality you could check several examples of levels, by loading Menu scene:



- Some of them have square tile grid and some have hex
- With squad or move range turn rule.

SuperTiles has slightly more than 200 C# scripts, but as it is based on the other asset –ProtoTiles which helps in tile map creation, it consists of 140 C# own scripts. Even though it looks like a lot, most of them match the Single Responsibility Principle, and the average row count of a single script is small enough. *SuperTiles* code is fully available: no dll or any other libraries. Key points are presented as interfaces or abstract classes to give you a power of easy modification to comply with game features which have not been implemented yet in this asset.



Also, the asset contains a bunch of Editor windows for simplification the process of main stuff creation (items, units, levels). They could be found under Tools/Red Bjorn/Editors submenu

The main purpose of *SuperTiles* is to become the most helpful asset for creation turn-based games, so it will not stay rigid and will get many updates. To achieve this your feedback is highly needed. Feel free to contact at:

Online-documentation

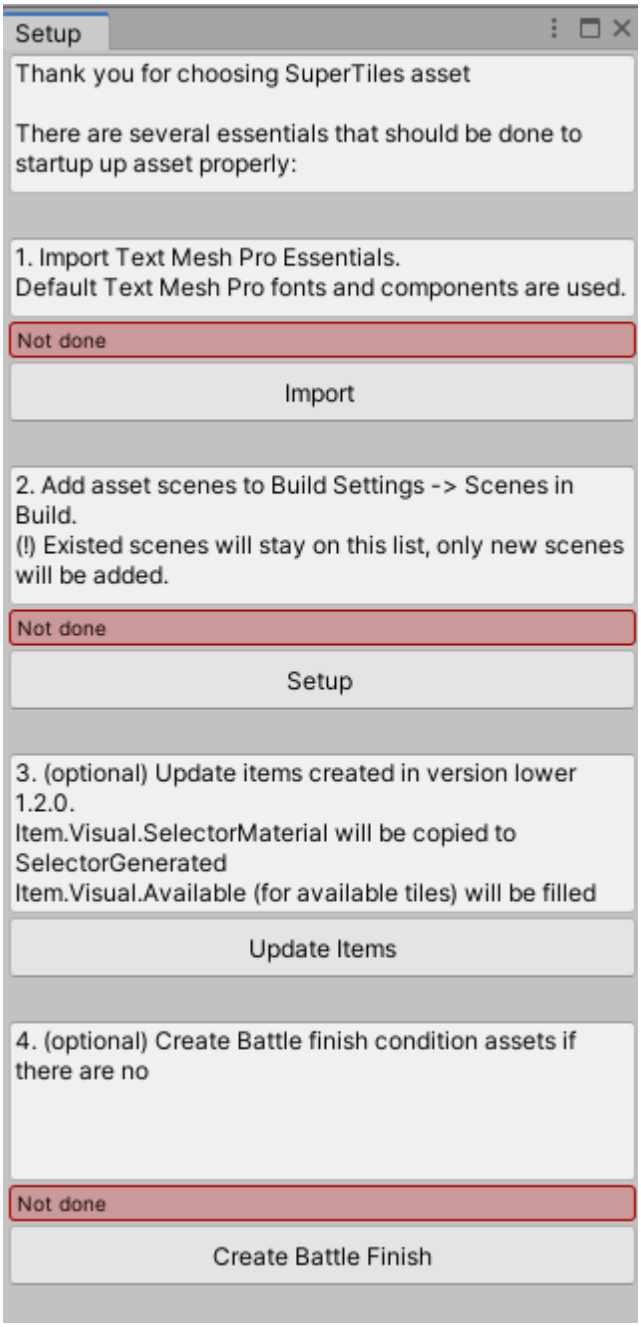
https://discord.gg/5dTM9SfqME

help@redbjorn.dev

1.4.7

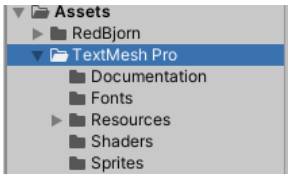
Setup

After the asset import process is done the Setup window will appear.



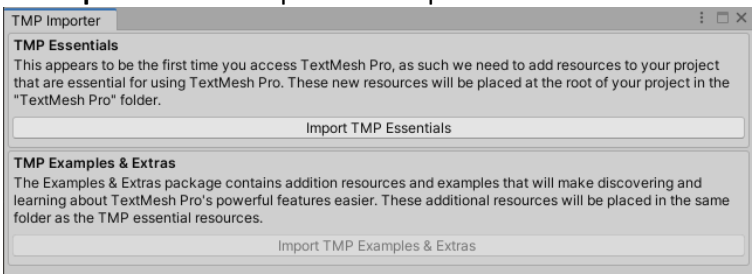
There are two rules which will be checked:

1.Does the project contain Text Mesh Pro essentials?



If “Not done” label appears, you could fix it this way:

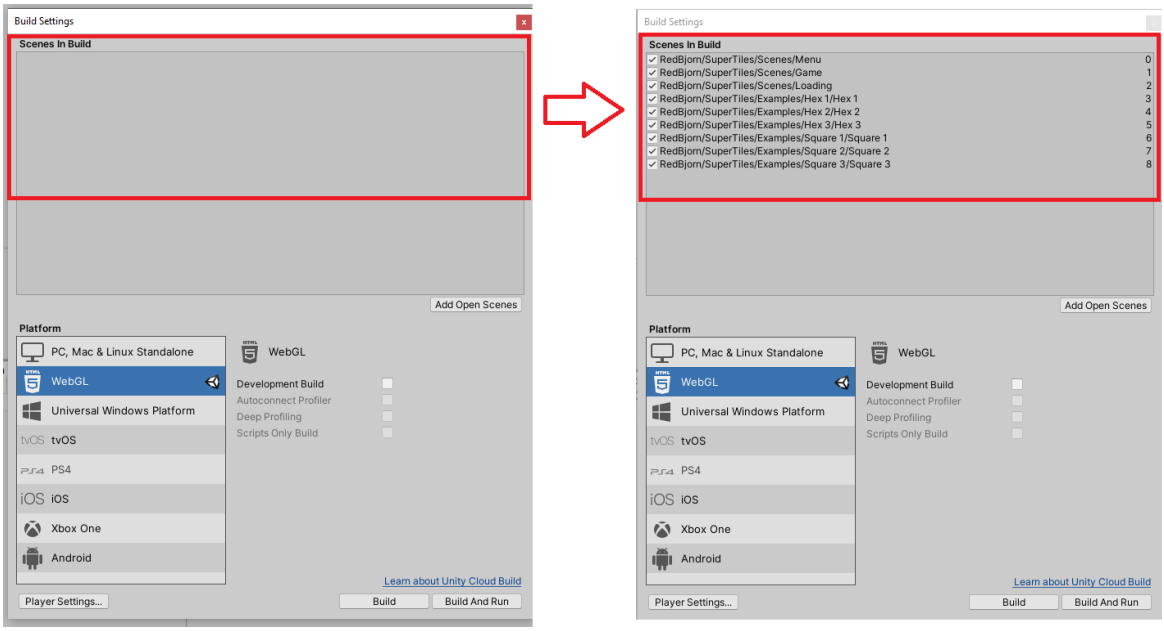
- Click **Import** button to open TMP Importer window



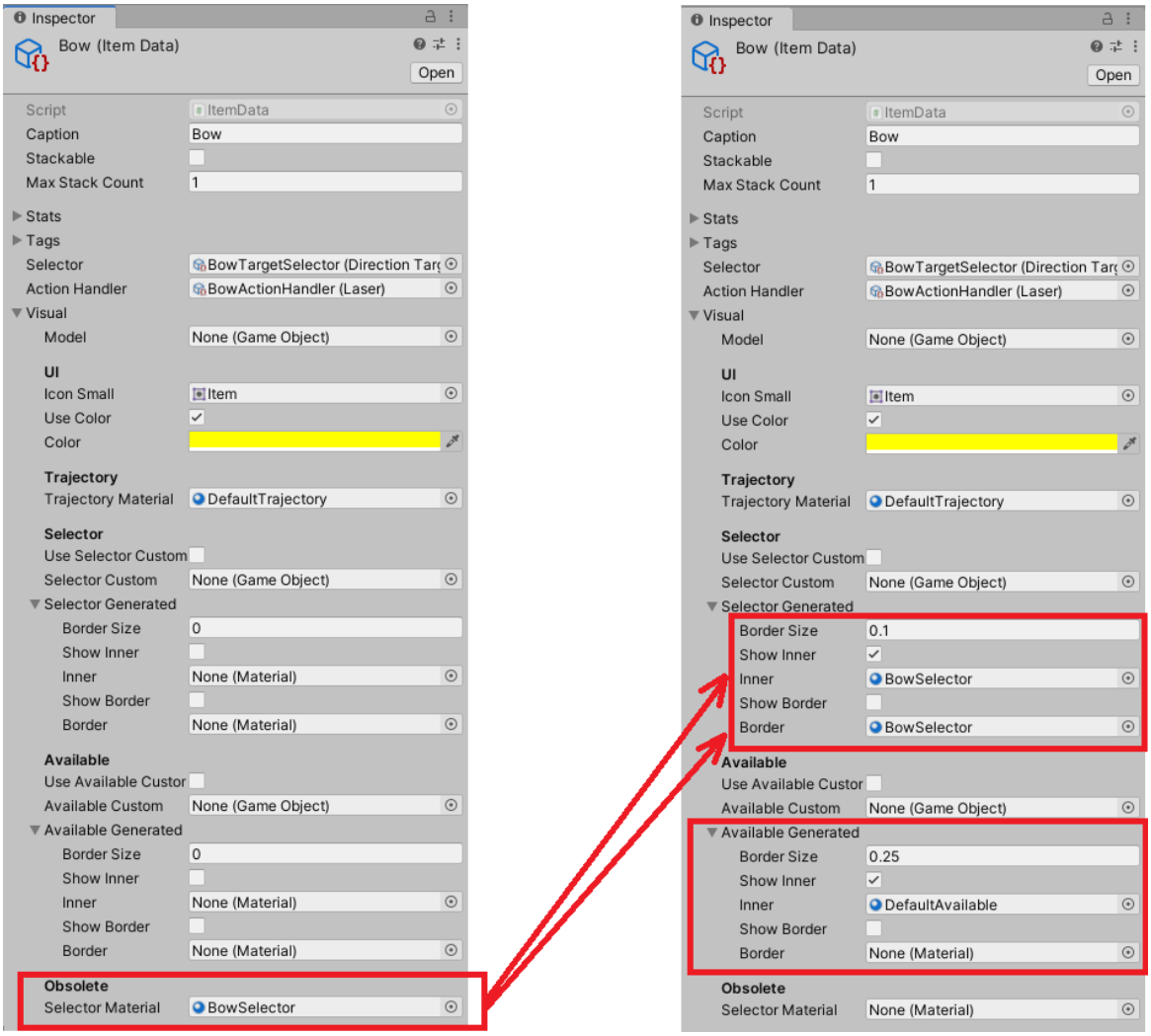
- Click **Import TMP Essentials**.

2.Does Scene in Build list contain essential scenes?

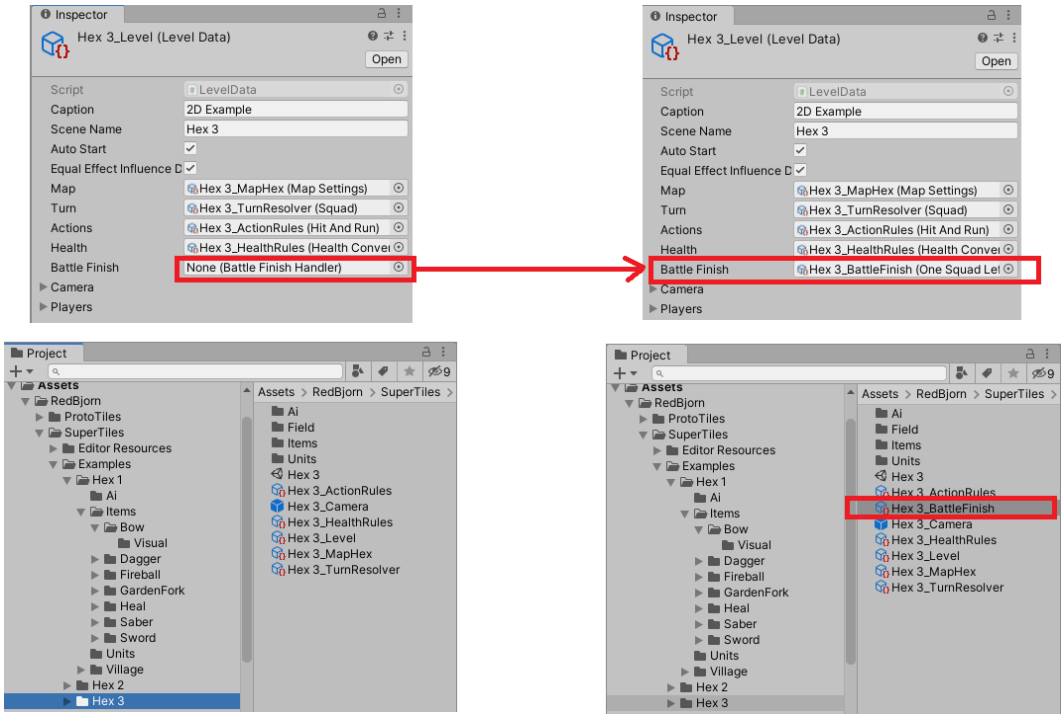
If “Not done” label appears, you could fix it, by clicking Setup button



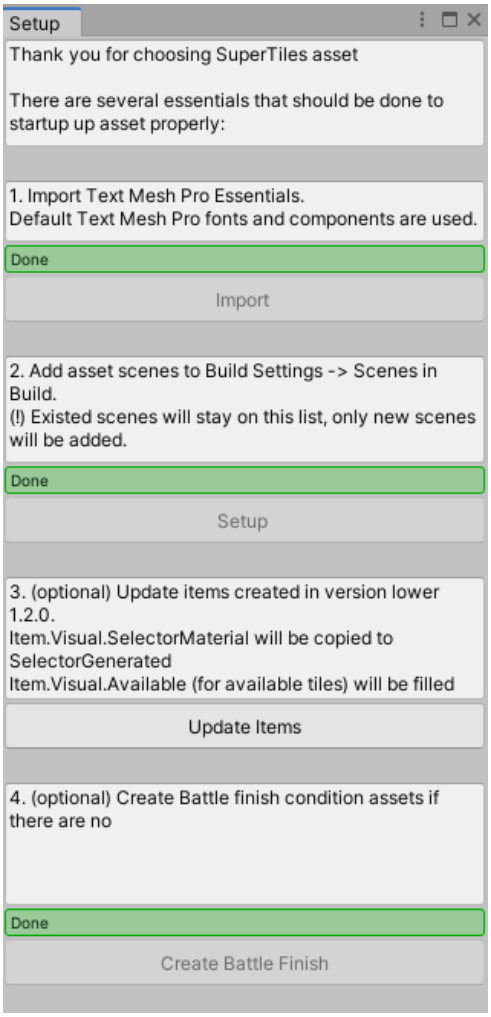
3.(optional) Copy obsolete field value to the newly created field at ItemData assets.



4.(optional) Create Battle finish condition, if there is no one.

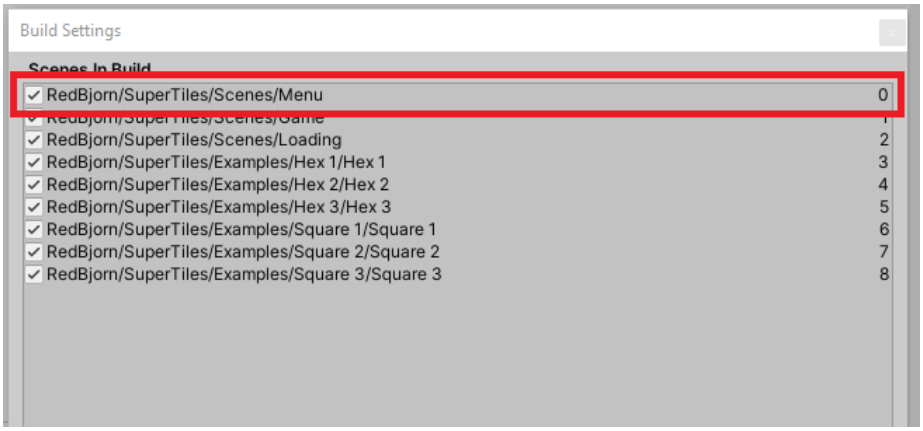


When the setup process is completed correctly then the Setup window becomes fully “green”

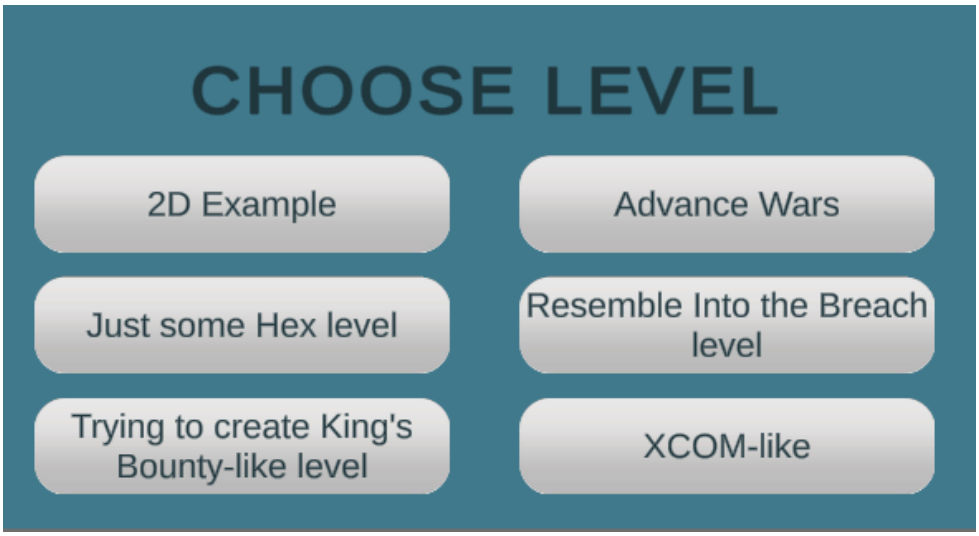

Setup
Thank you for choosing SuperTiles asset
There are several essentials that should be done to startup up asset properly:
1. Import Text Mesh Pro Essentials.
Default Text Mesh Pro fonts and components are used.
Done
Import
2. Add asset scenes to Build Settings -> Scenes in Build.
(!) Existed scenes will stay on this list, only new scenes will be added.
Done
Setup
3. (optional) Update items created in version lower 1.2.0.
Item.Visual.SelectorMaterial will be copied to SelectorGenerated
Item.Visual.Available (for available tiles) will be filled
Update Items
4. (optional) Create Battle finish condition assets if there are no
Done
Create Battle Finish

How to run the asset

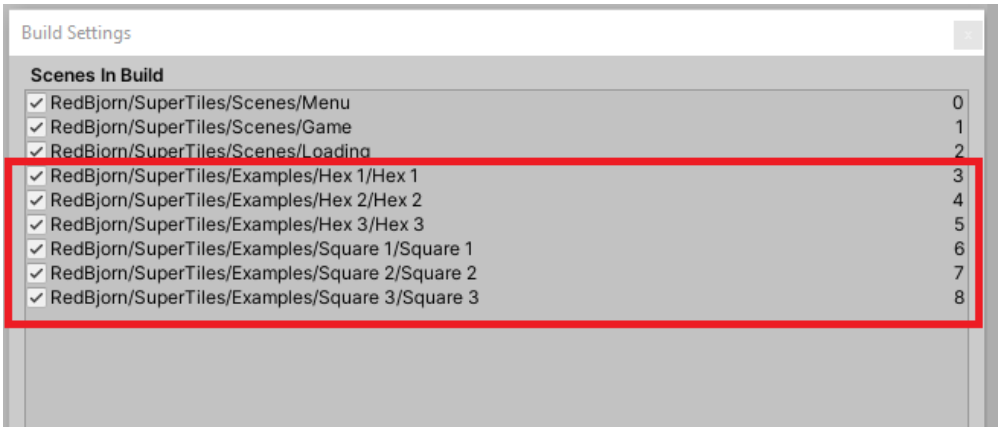
First, you should load the Menu scene. It is the initial scene for the asset.



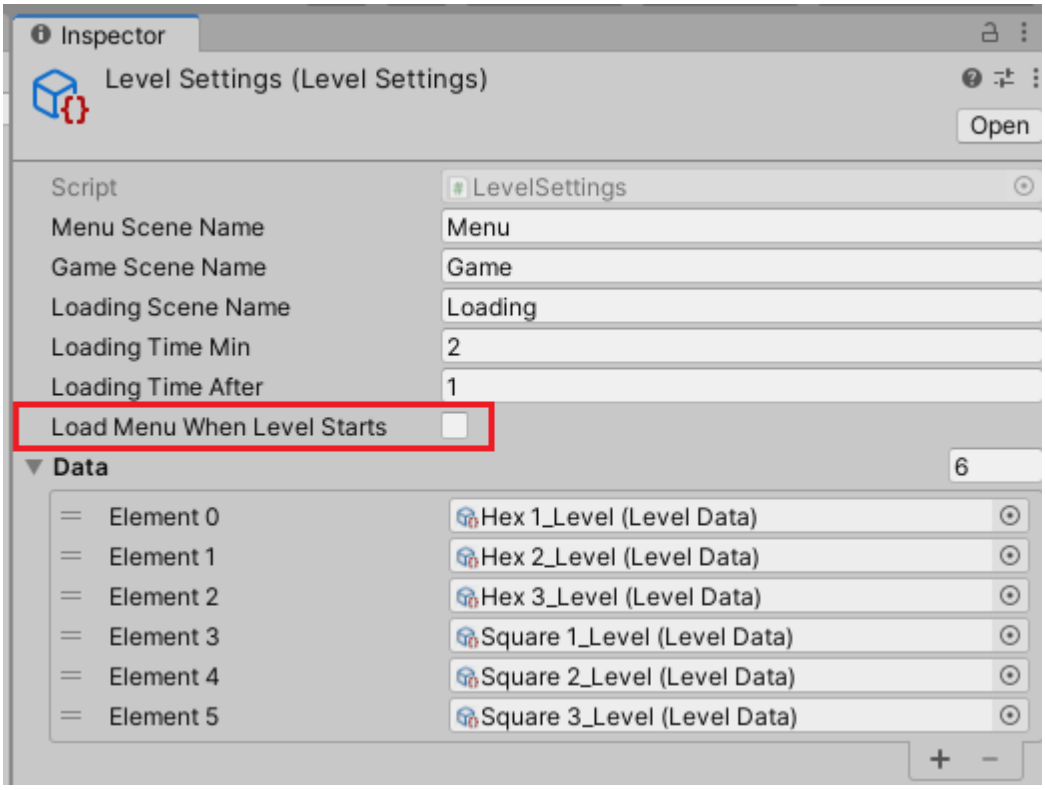
Here you can see six level load buttons. Click on any of them will load the game immediately



Also, you could start from any of the example scenes directly.



If loading from the example scene redirects you to the Menu scene and you doesn't need this redirection you could disable this logic in the LevelSettings asset

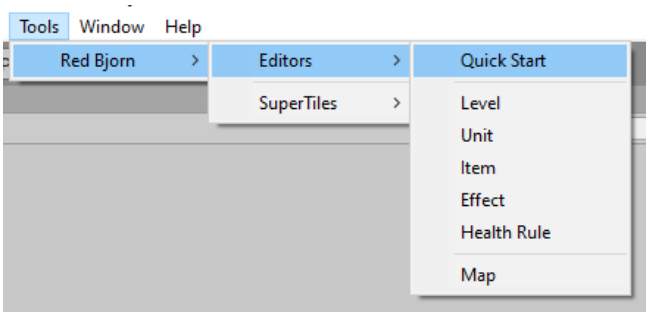


The rest two scenes (Game and Loading) are helper scenes. Loading from them will not lead you to a proper game state.

If **2D Example, Advance Wars** scenes don't load properly you need to check that **Hex 3, Square 3** scenes are added to the **Build Settings -> Scenes In Build**

Quick Start

To create your own level you need to select **Quick start** from Tools/Red Bjorn/Editors submenu.



You will see the Quick Start window where fields are already filled.

1. Folder is the root folder of level related to the Assets folder

2. Scene Filename is the name of scene file without .unity extension

3. Map Grid could be Square or Hex.

4. Map Axis defines which plane will be used for map creation (XZ, XY, ZY)

5. Action Rules define the way unit could make its actions

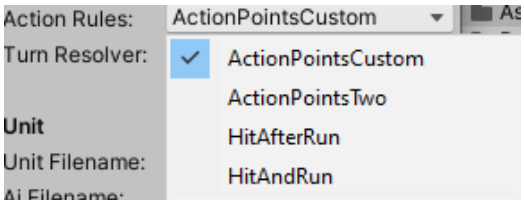
6. Turn Resolver defines the way turn completion is handled

7. Battle Finish – finish battle conditions

8. Unit Filename is the name of the asset file which will be created for storing information about unit (at least one unit should be created to start the level)

9. Ai Filename is the name of the asset file which will store information about unit artificial intelligence

10. Ai Logic is the type of Ai behavior. Right now, there are 2 types: SimpleLogic and Universal



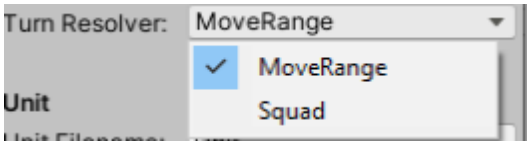
ActionPoinstCustom allows to spend X Points (2 by default), where 1 move action cost 1 Point and 1 item action cost also 1 Point. Move action after Item action is prohibited.

ActionPoinstTwo = ActionPoinstCustom where X = 2 constanlty (most Common)

HitAfterRun allows

- 1 move action or
- 1 item action or
- 1 item action after 1 move action

HitAndRun allows 1 move action and 1 item action in any order

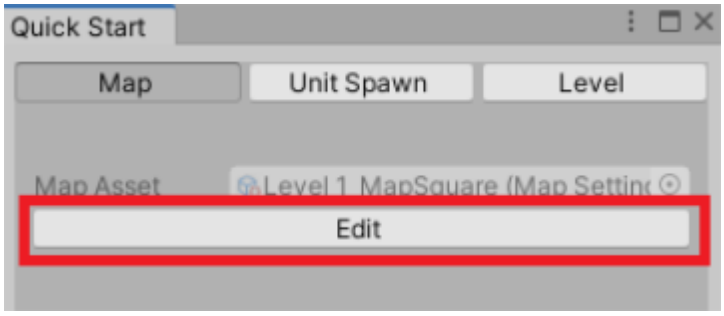


MoveRange: after the unit will do its' actions the turn will be moved to the next unit (it can be in other squad) in list ordered by move range value.

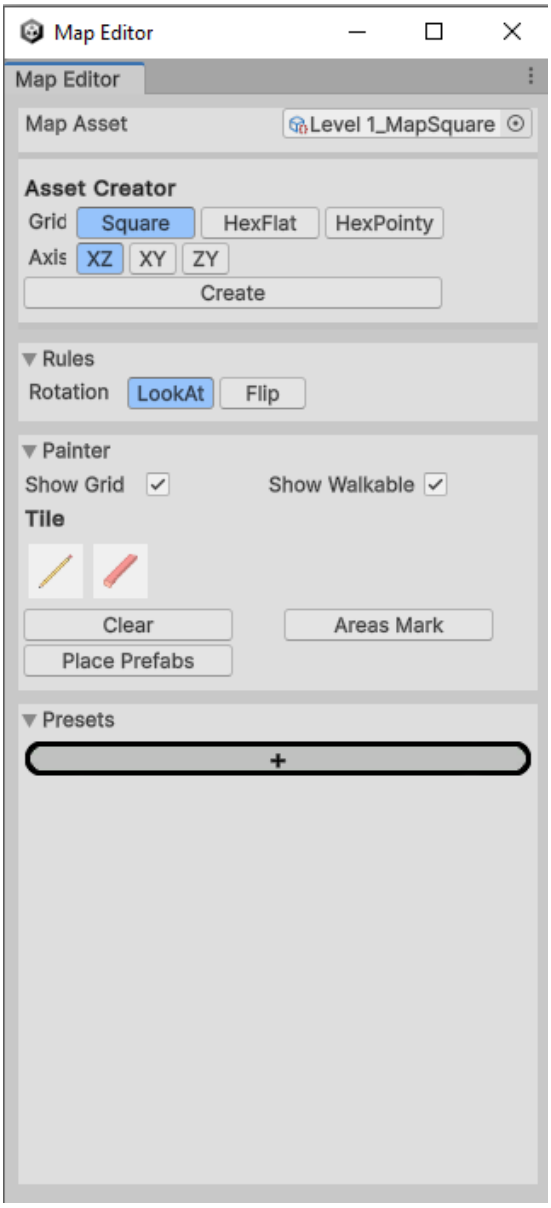
Squad: all units in squad could do its' actions and then turn will be moved to other squad

After specifying the desired values and clicking Create button you will see next three steps

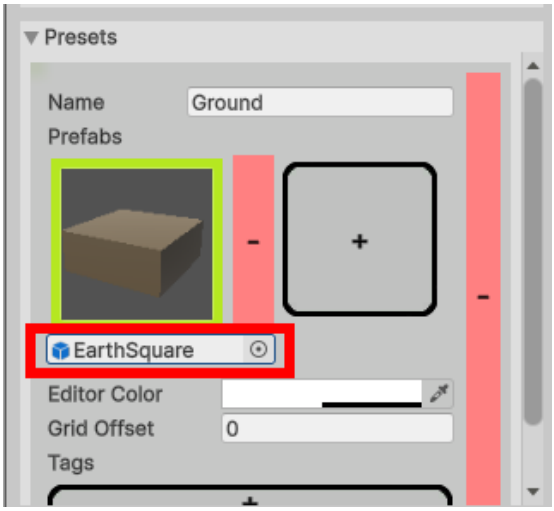
Let us start from map creation like it is conceived and click Edit button



You will see Map Editor window (more information about Map Editor is inside ProtoTiles documentation)

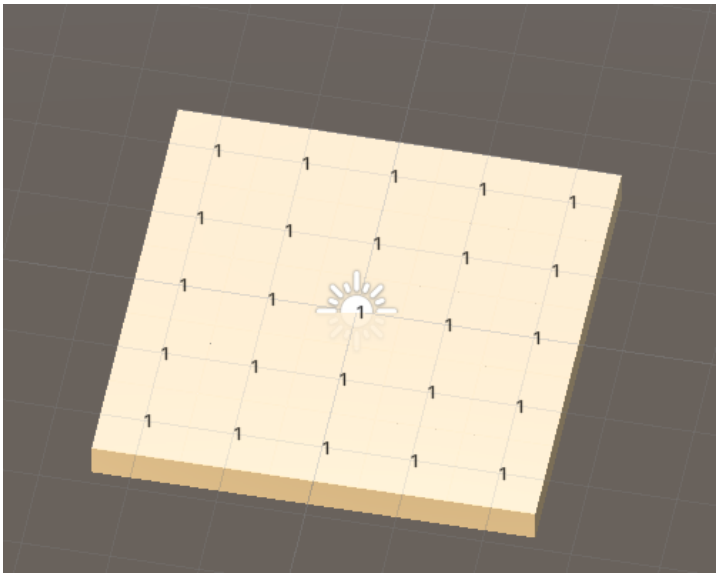


Click Plus button to add initial tile preset and specify prefab field with the cube of 1 unit height and width, for example, EarthSquare

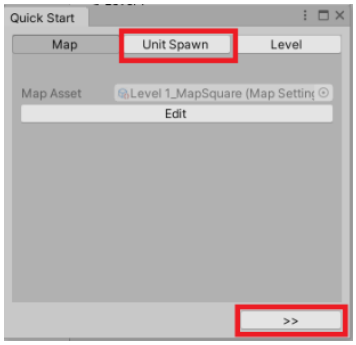


Then click the Pencil button to draw the map at the Scene view tab.

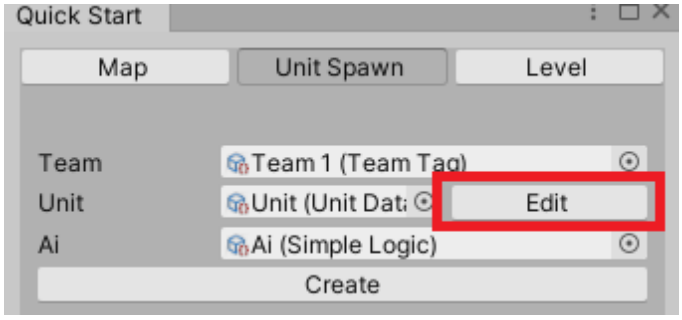
Let us draw 5x5 tile map (or any map you can imagine)



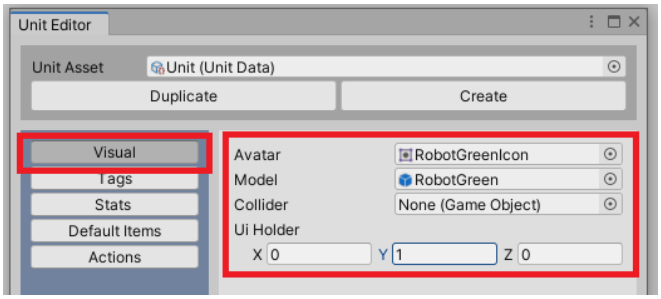
Close Map Editor window and return to the Quick Start window and move to the next step - UnitSpawn



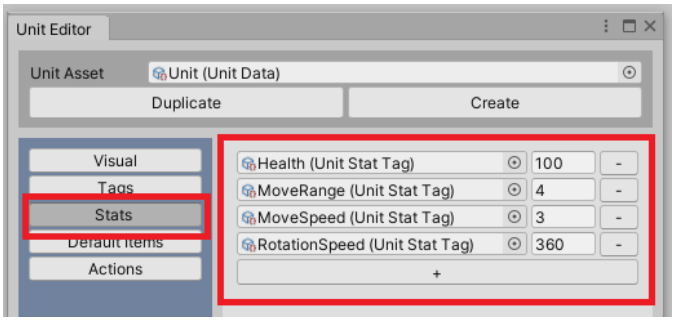
Here you will see already filled fields for Unit spawn point creation. But do not hurry, first Click Edit button to configure our blank Unit asset. It will be the template for other units.



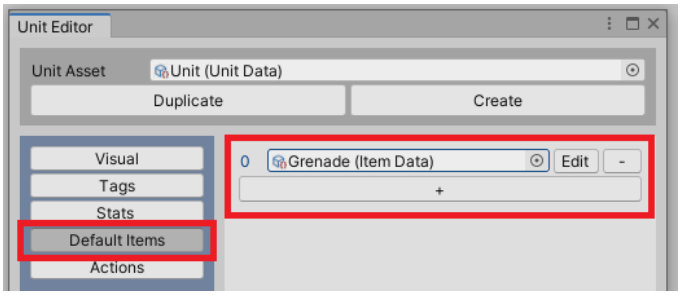
New window (Unit Editor) will open. Define Visual parameters



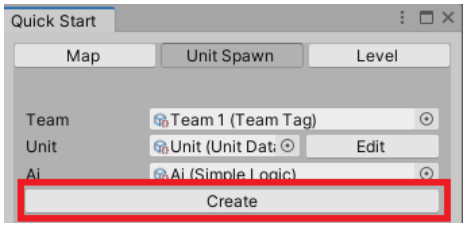
Then define Stats parameters



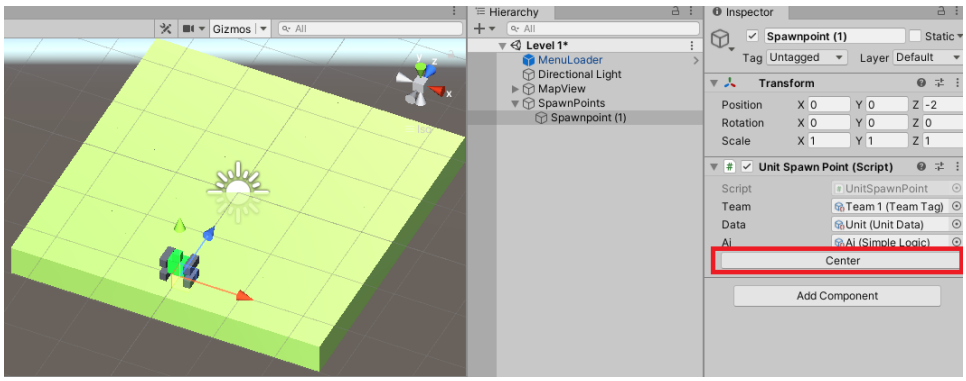
And add at least one default item. You could also create new item here, clicking Edit button will open Item editor, but for now, let us choose from already created items, for example, Grenade and return to Quick Start window.



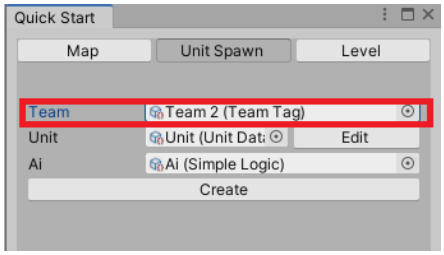
Click Create button to add Unit Spawn point to the Scene View tab



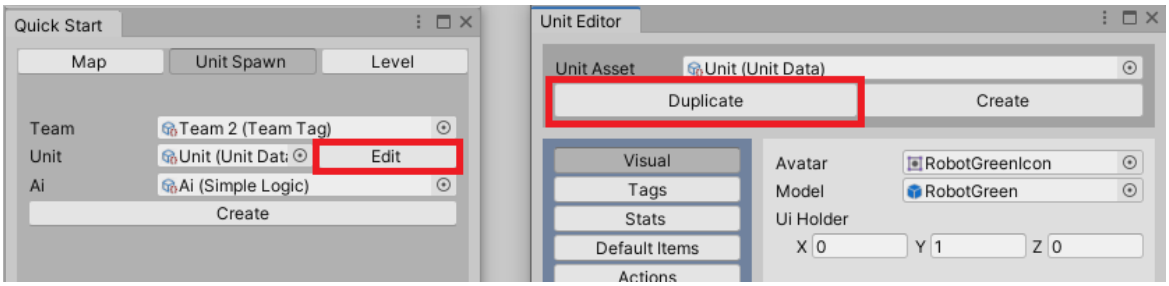
Spawnpoint (1) will be created at Vector3.zero position and you can move it to any map position and click Center button to align with the tile center



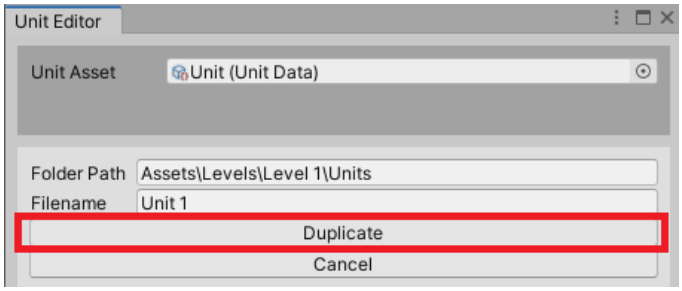
Return to Quick Start window and change unit team to Team 2



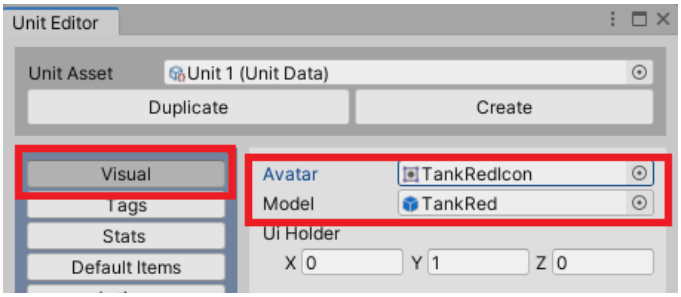
Let us create another unit to prevent the enemy unit from looking like initial unit. Click Edit button than Duplicate button at Unit Editor



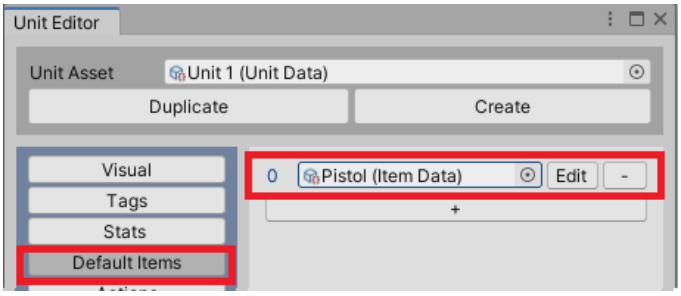
After specifying the desired filename click the Duplicate button.



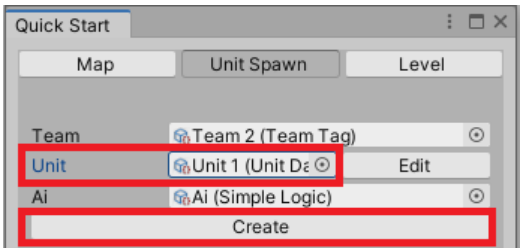
After asset creation, a new unit will be selected at the Unit Editor window. Let us change its model.



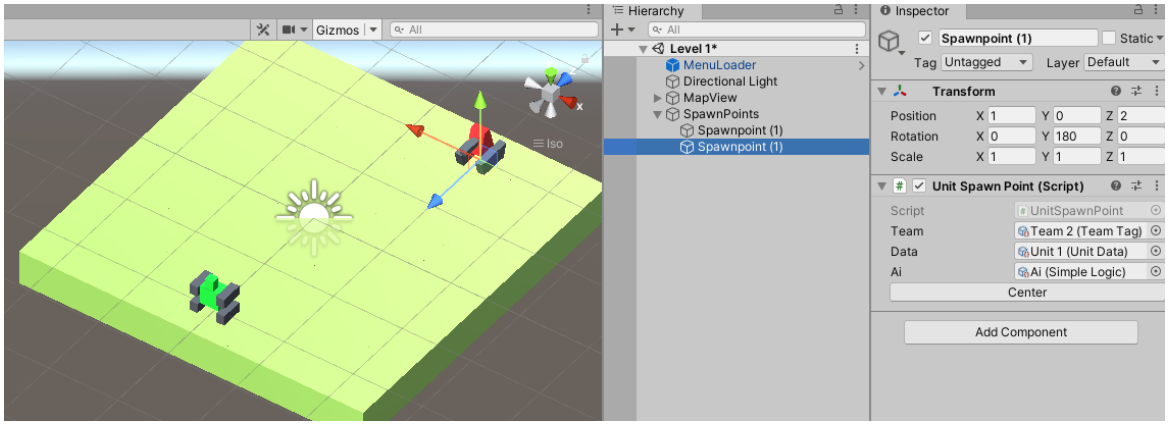
And change its default item



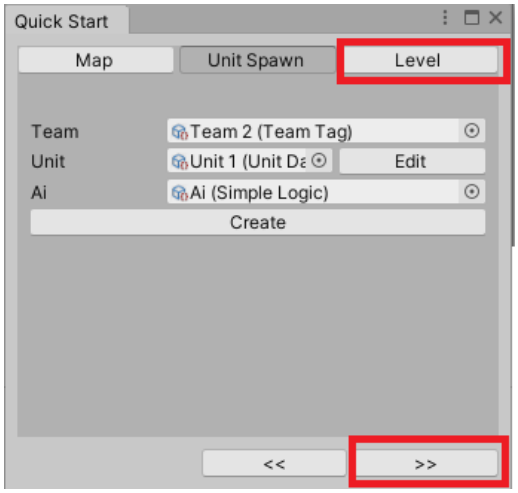
Return to Quick Start window and define new unit inside Unit field and click Create button



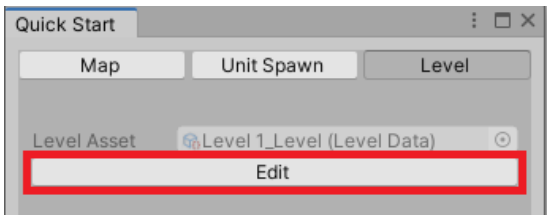
Move the new unit to the correct position. Do not forget about Center button to align spawnpoint position



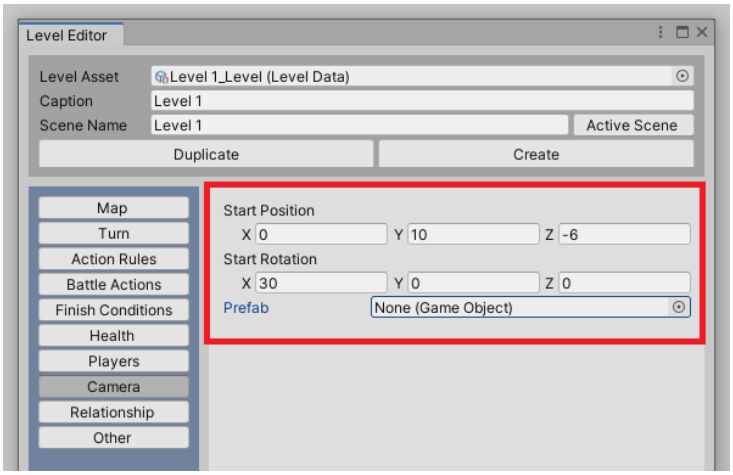
In Quick Start window move to the next step – Level



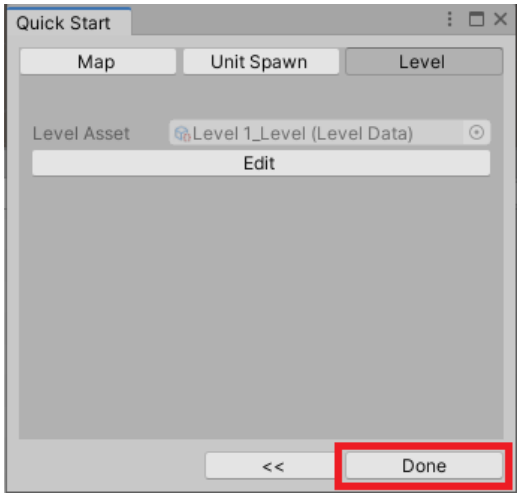
Click Edit button to open Level Editor window



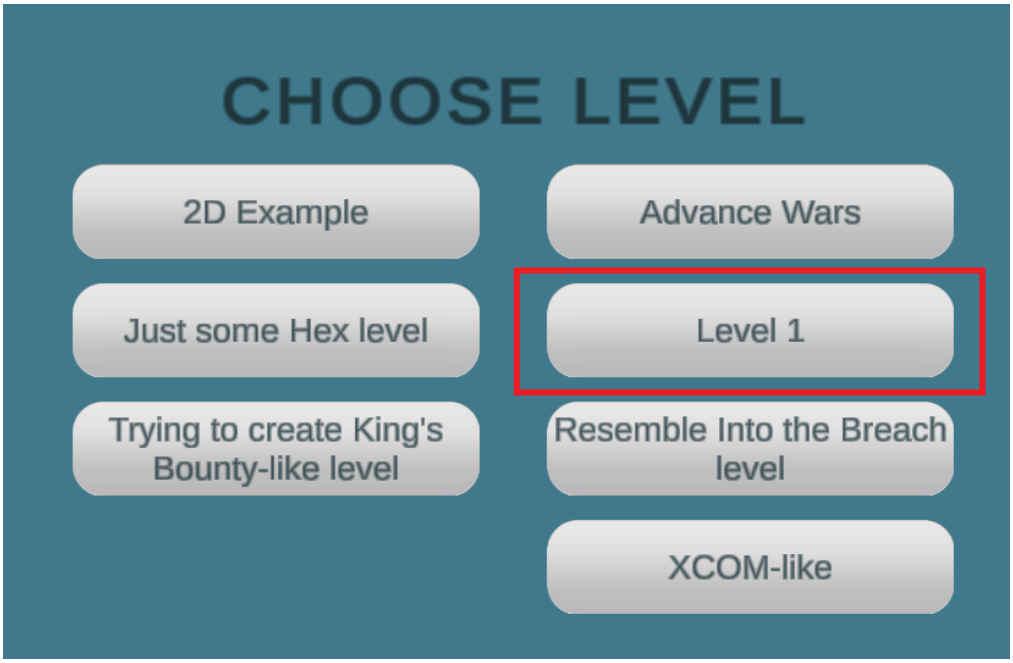
At the Camera tab specify Start Position. For our map (0, 10, -6) will be appropriate position.

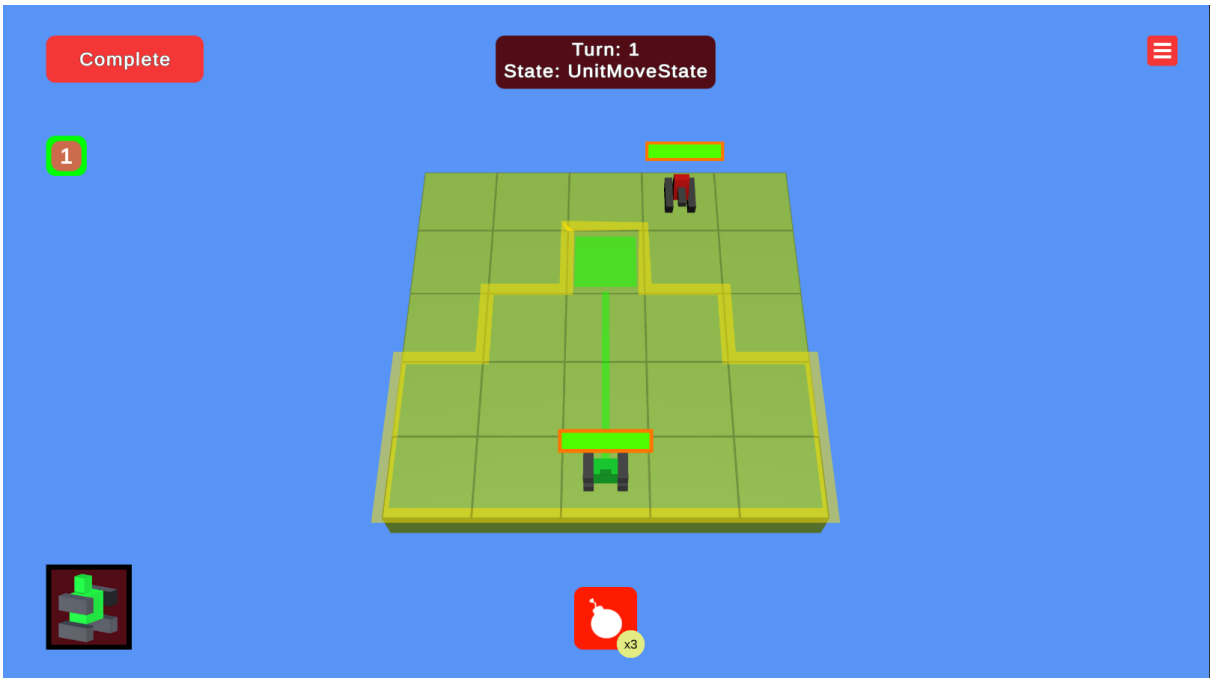


Return to Quick Start window and click Done button



Ta-dam! Congratulations you have created your own level. To check it, click the Playmode button. You will be redirected to Menu scene.


CHOOSE LEVEL
2D Example
Advance Wars
Just some Hex level
Level 1
Trying to create King's Bounty-like level
Resemble Into the Breach level
XCOM-like

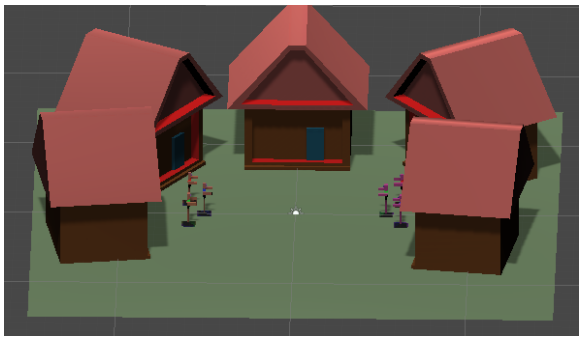

Complete
Turn: 1
State: UnitMoveState
1
x3

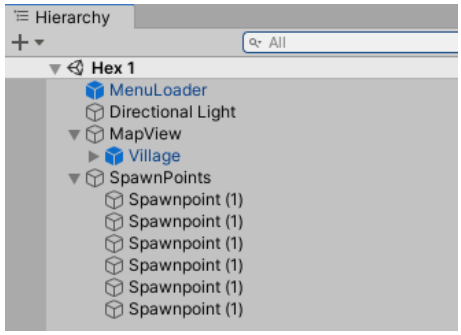
Concept

As an usual game, *SuperTiles* uses a bunch of Data and the Logic which handles its Data.

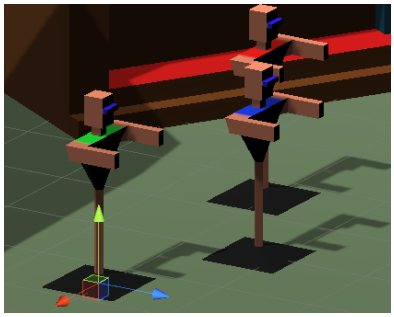
For every important part of Data and Logic there is an individual section which contains more detailed description. Here it will be explained only the basis of *SuperTiles*:

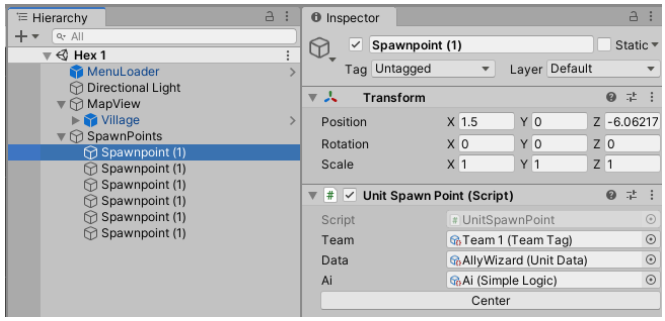
On the bottom of our Data concept there is a well-known unity scene which contains visual information about the level. Map representation is located under MapView gameobject (in our example it is the Village prefab).



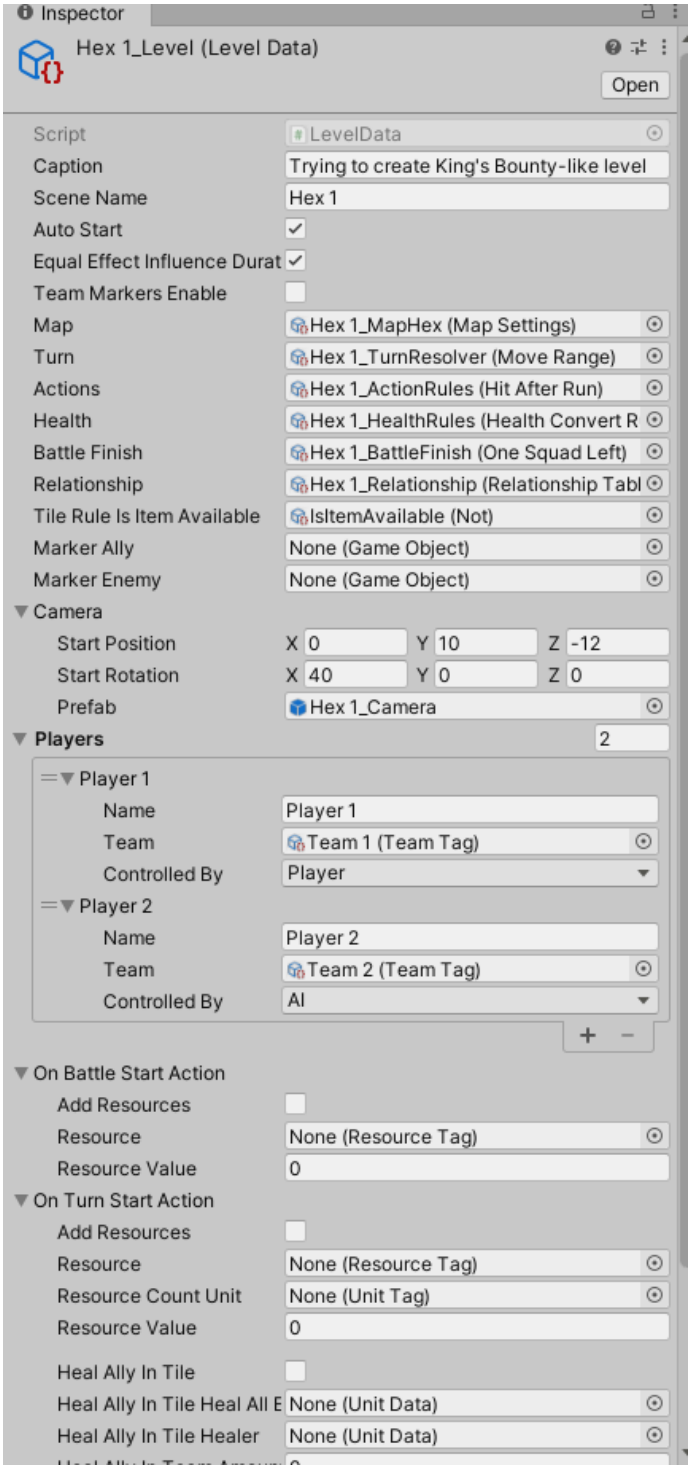


Also, scene contains some unit data. It is located inside spawnpoint gameobjects. UnitSpawnPoint class defines which Team the unit will belong to and what kind of unit (Data field) will be spawned at the corresponding position and what UnitAiData asset will be used if this unit will be controlled by Ai player





Further, the reference to this scene is stored inside LevelData, which is another key point of the concept.



Apart from the scene reference (as Scene Name field), LevelData keeps MapSettings link which defines what grid settings will be used at this scene (which tiles are movable and which are not).

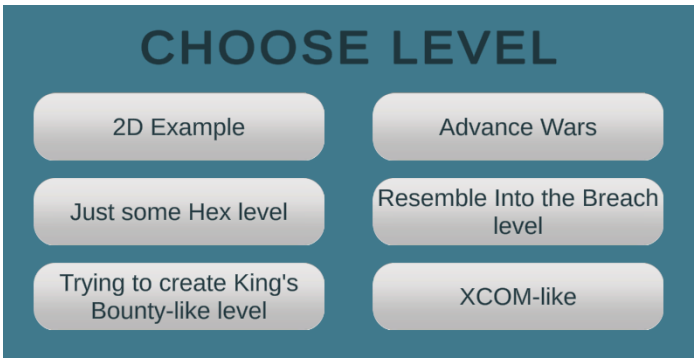
Also, LevelData defines the way the turn is moving from one unit to another, by specifying Turn field.

Actions field is neccesary for counting how many move and item actions can be done by a single unit and describes the valid order of these actions.

Besides, we need to know who will controll every team. This information is also a part of LevelData and can be found at the Players field.

After doing a brief Data overview, let's move on to the Logic description.

Based on the information above it is known that Menu scene is the initial scene to be loaded, but there is no direct reference to any LevelData asset inside scenefile. So where is it?

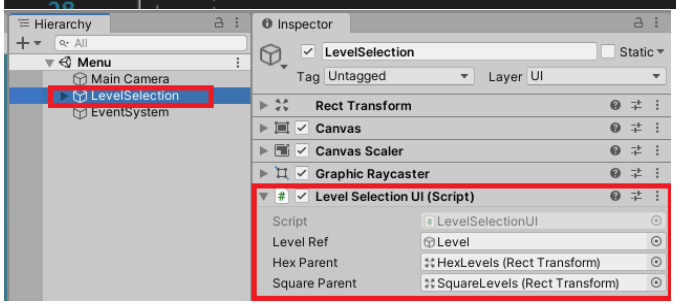


It is worth mentioning that we have something like a root Data object called GameSettings. It is located inside the Resource folder and has lazy initialization.

```
4   namespace RedBjorn.SuperTiles
5   {
6       public class S
7       {
8           static GameSettings CachedGame;
9           static GameSettings Game
10          {
11              get
12              {
13                  if (CachedGame == null)
14                  {
15                      CachedGame = Resources.Load<GameSettings>("GameSettings");
16                  }
17                  return CachedGame;
18              }
19          }
```

Exactly such a reference is used inside the LevelSelectionUI script which is attached to the UI gameobject of the Menu Scene.

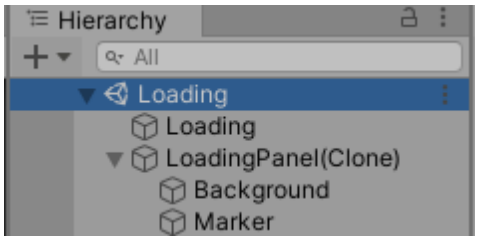
```
21          void Start()
22          {
23              foreach (var p in S.Levels.Data.OrderBy(p => p.Caption))
24              {
25                  CreateLevelButton(p, ButtonParent(p.Map.Type));
26              }
27          }
```



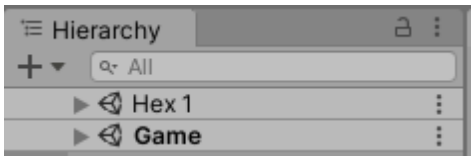
Click on any level button invokes LoadLevel method

```
void LoadLevel(LevelData level)
{
    //Create GameEntity state to pass through the scene loading
    GameEntity.Current = new GameEntity
    {
        Creator = new GameTypeCreators.Singleplayer(),
        Loader = new GameTypeLoaders.Singleplayer(),
        Restartable = true,
        Level = level,
        RandomSeed = UnityEngine.Random.Range(int.MinValue, int.MaxValue)
    };
    //Load level scene through Loading scene
    SceneLoader.Load(level.SceneName, S.Levels.GameSceneName);
}
```

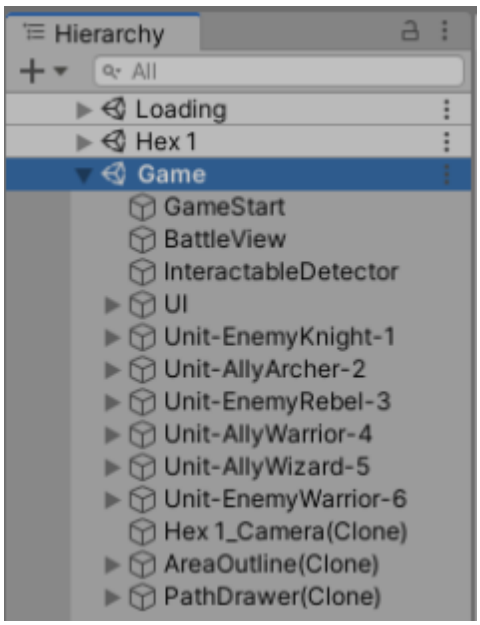
Level loading is performed by a transitioning to an almost empty scene (Loading) to prevent ugly render stuck when a fat scene file should be loaded. It was said “almost”, because there are several objects representing loading process



After the level will be loaded, it is needed to add common stuff (UI, Main camera etc.) And another helper scene (called Game) will be loaded.



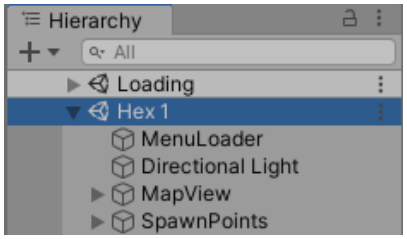
Game scene contains the starter script called GameStart where essential classes are created. Usually essential classes will contain Entity suffixes (GameEntity, UnitEntity, ItemEntity etc.). It indicates that these classes are plain c# classes and could be easily serialized. The main essential class is the GameEntity class. It stores the full state of the current level and also is used to recreate game from the savefile.



Before the process of essential entity creation will start, it should be defined as whether the game will be created from scratch or will be loaded from an existing state. To understand what path is right, we check if the GameEntity contains something in Battle field.

```
if (game.Battle != null)
{
    yield return Loading(game);
}
else
{
    yield return Creating(game);
}
```

When the creation process is finished, the Loading scene will be unloaded and you can start playing the game. Playing the game means that you start a Battle, and the description of this process is declared here




Scene
Game
Animator
Display 1
16:9
Scale
1x
Maximize On Play
Mute Audio
Stats
Gizmos
Complete
Turn: 1
State: UnitMoveState
1
2
3

Core

Tags

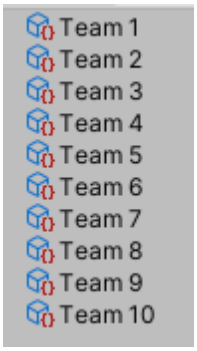
Tags in SuperTiles differ from Unity tags, although they serve the same purpose. There is only one difference – implementation. Unit tag is a single string which is located inside a TagManager asset. SuperTiles tag is a single ScriptableObject which is located somewhere in the Assets folder. Such a tag concept is not an original idea, but seems more comfortable, because of its handy drag-and-drop native feature and flexible refactoring possibilities.

SuperTiles default tag instances (successors of Tag class):

TileTag

(from ProtoTiles)

TeamTag

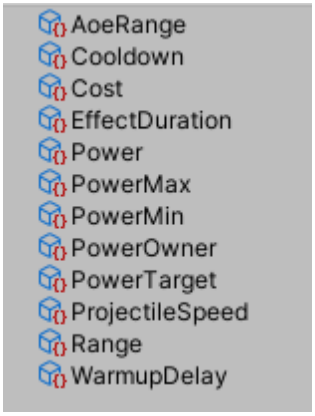


UnitTag

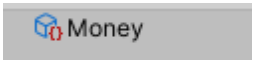
UnitStatTag

ItemTag

ItemStatTag

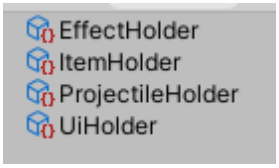


ResourceTag



EffectStatTag

TransformTag

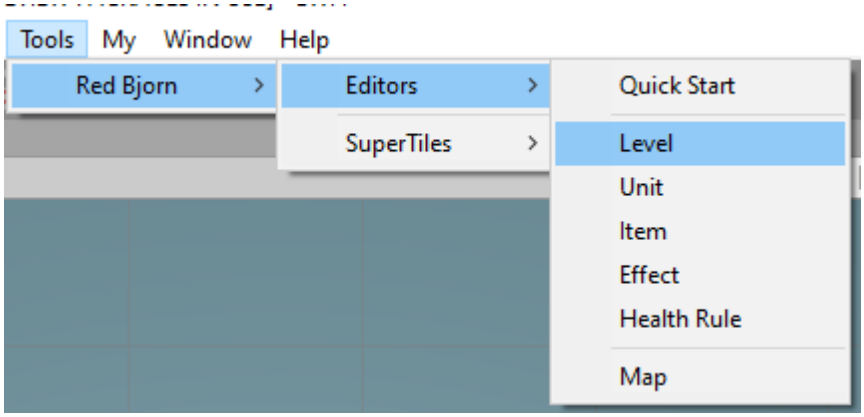


Existed tag list can be easily extended by duplicating the desired instance and giving it a new appropriate name. Also, you could create your own tag type by creating a new Tag class successor.

```
public class TeamTag : Tag
```

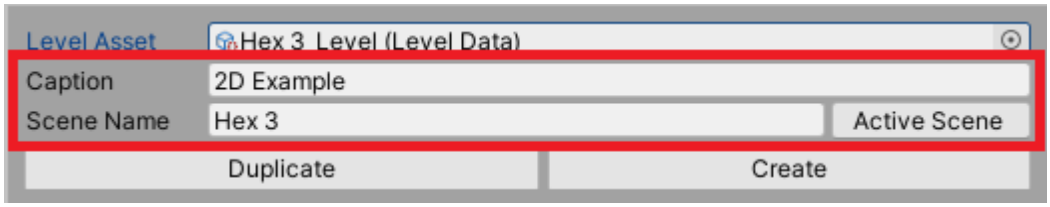
Level

The key point of current turn-based game implementation is LevelData class. Level Editor window is designed for convenient editing of exactly this class.

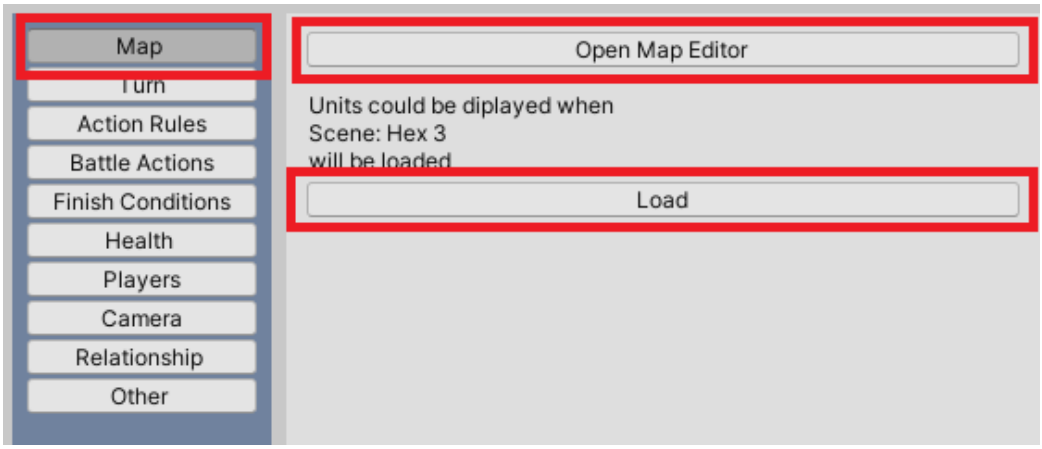


It contains information about:

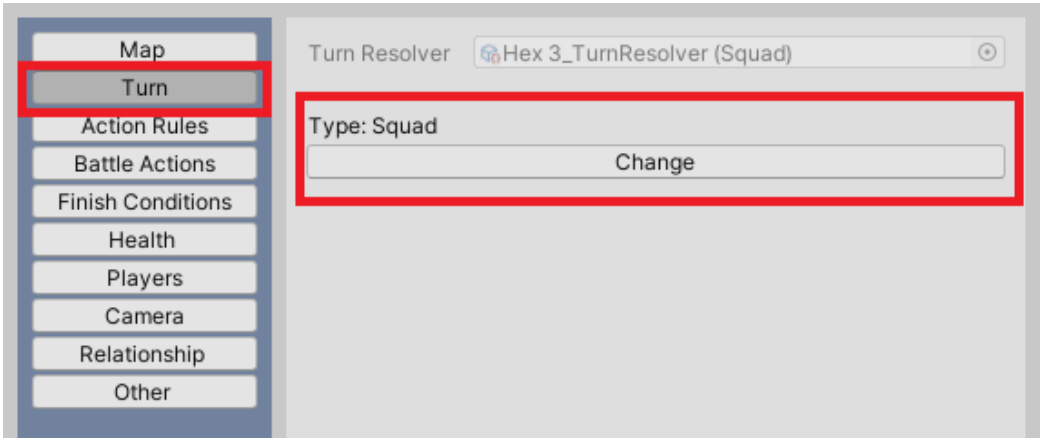
What Caption will be displayed and what Scene should be loaded?



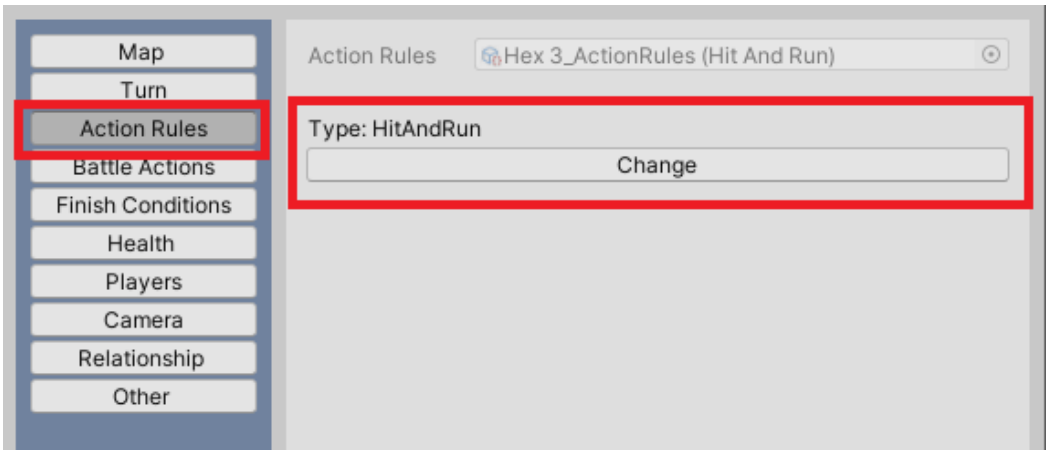
Buttons for loading appropriate scene and opening Map Editor window



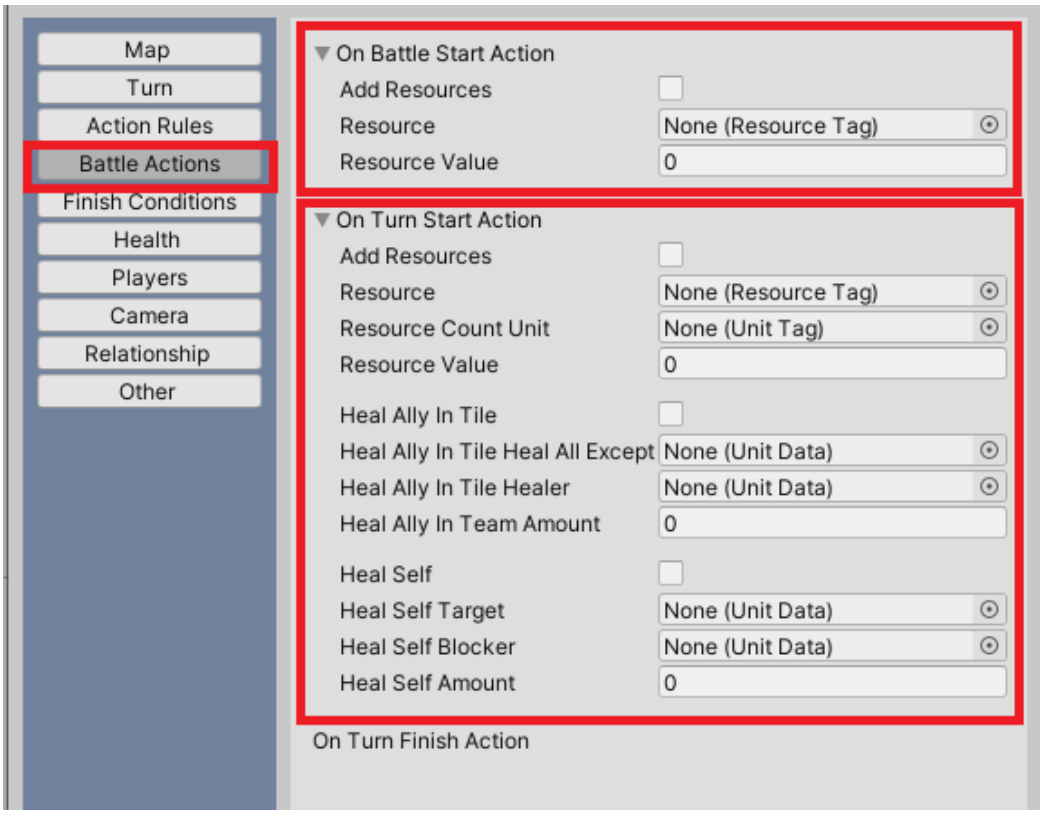
What kind of Turn Resolver is used and button to change the type



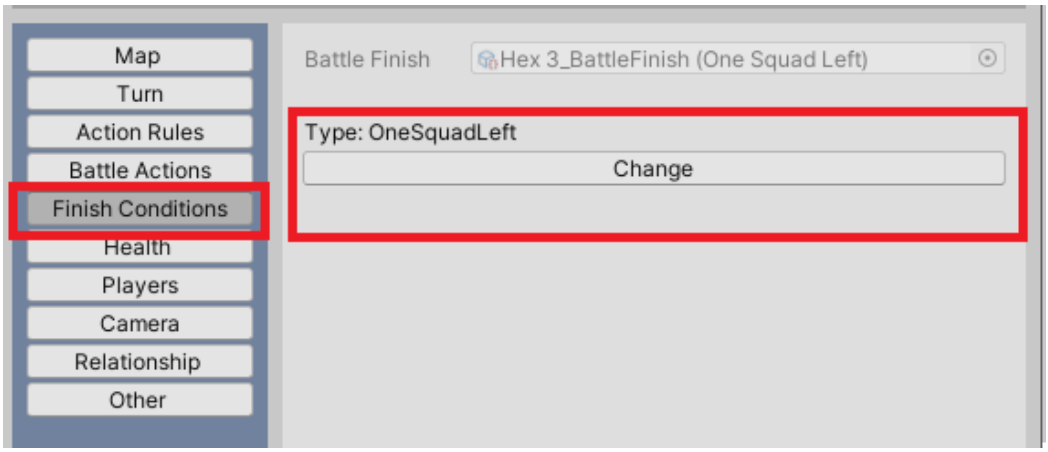
Which Action rules should unit conduct during its turn?



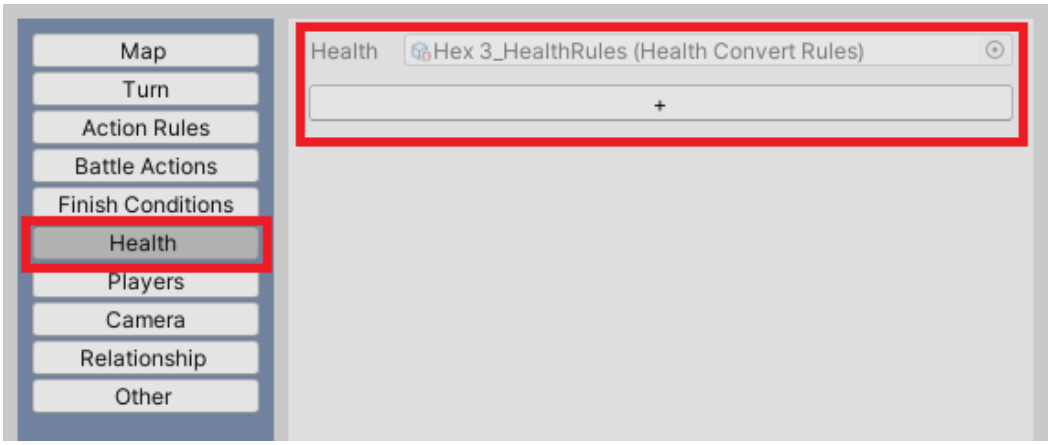
Custom Battle actions: On Battle Start or On Turn Start (On Turn Finish is under development)



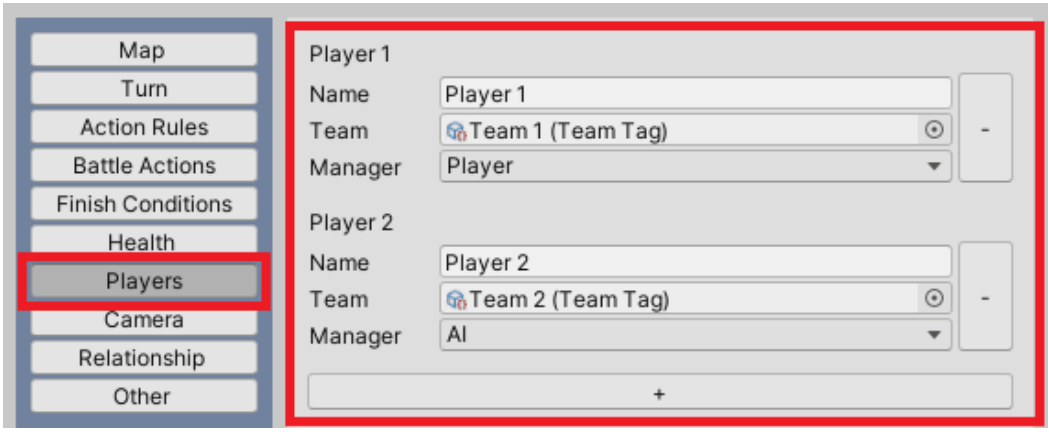
What conditions should be used to check Battle_Finish?



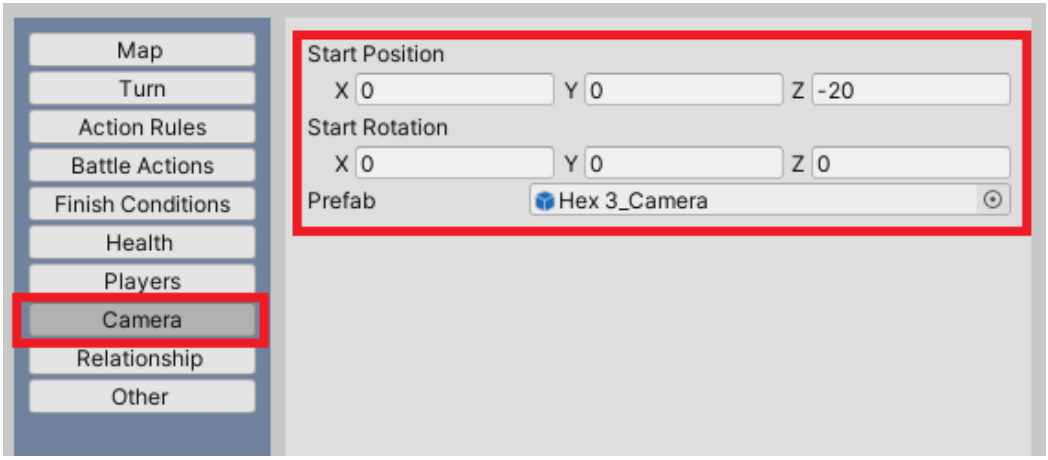
Health Rules which are checked on every health change action



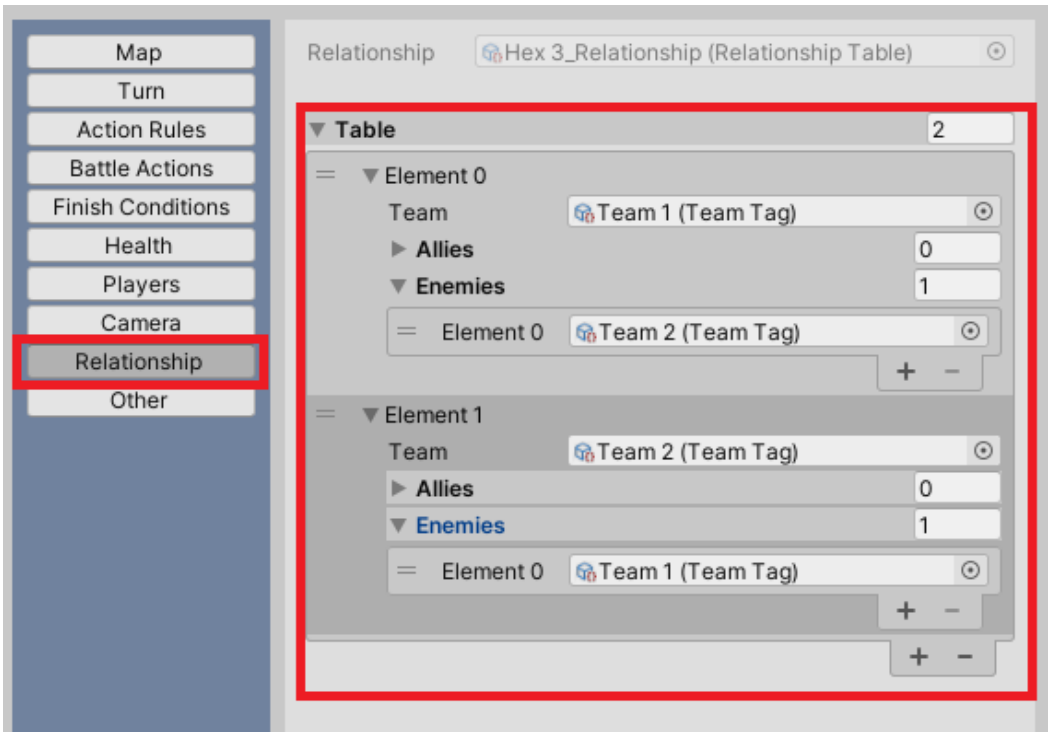
Which team is controlled by Player or by AI?



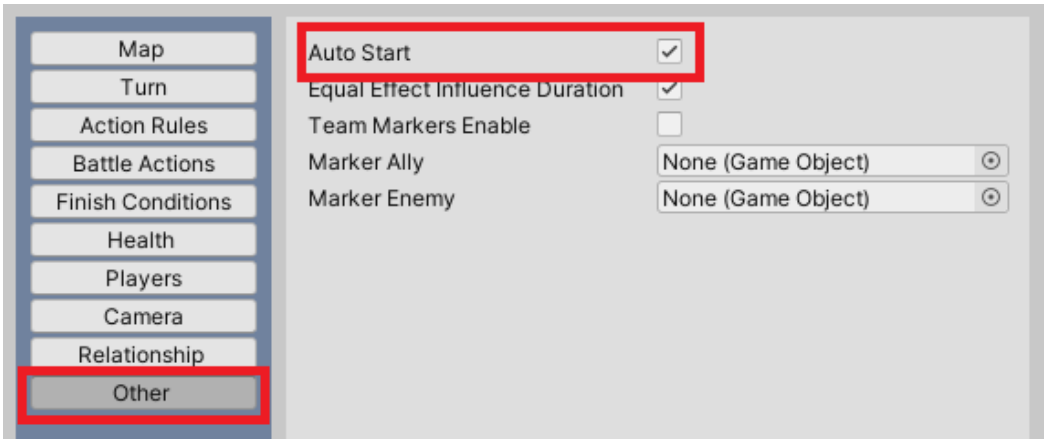
Start position and rotation for Camera gameobject. Prefab which contains Camera component (if none default prefab will be used)



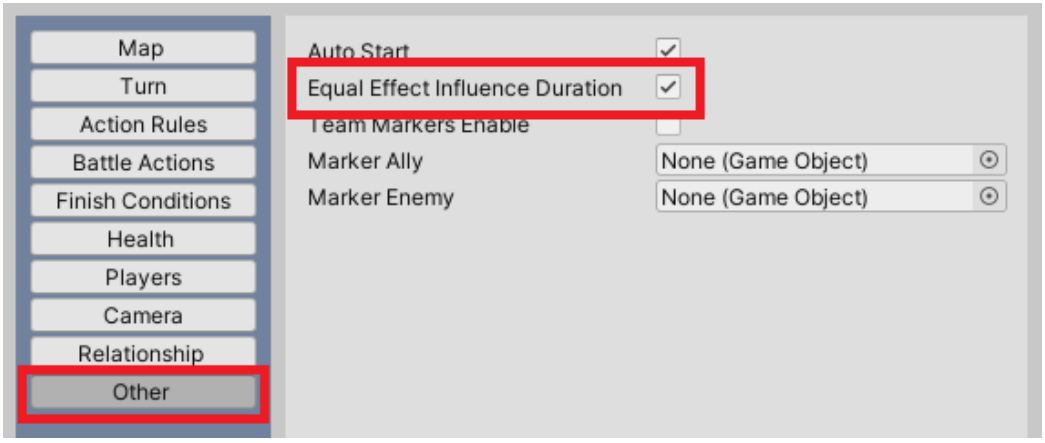
Relationship between teams (allies, enemies)



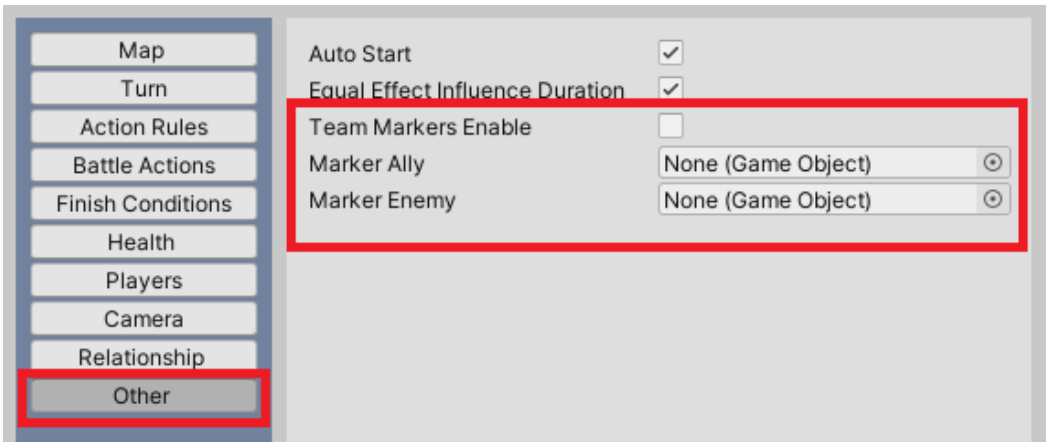
Should the battle start immediately after loading the scene?



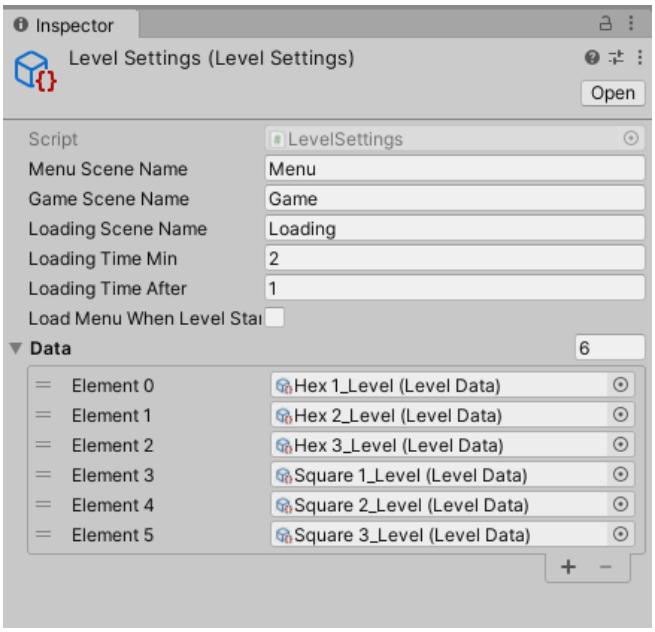
Should effects be stacked by duration increase?



Should team unit markers be enabled? (Works when Marker Ally and Marker Enemy is added)



There is a list of all LevelData assets which could be loaded through Menu scene. Such a list is located inside the LevelSettings asset.



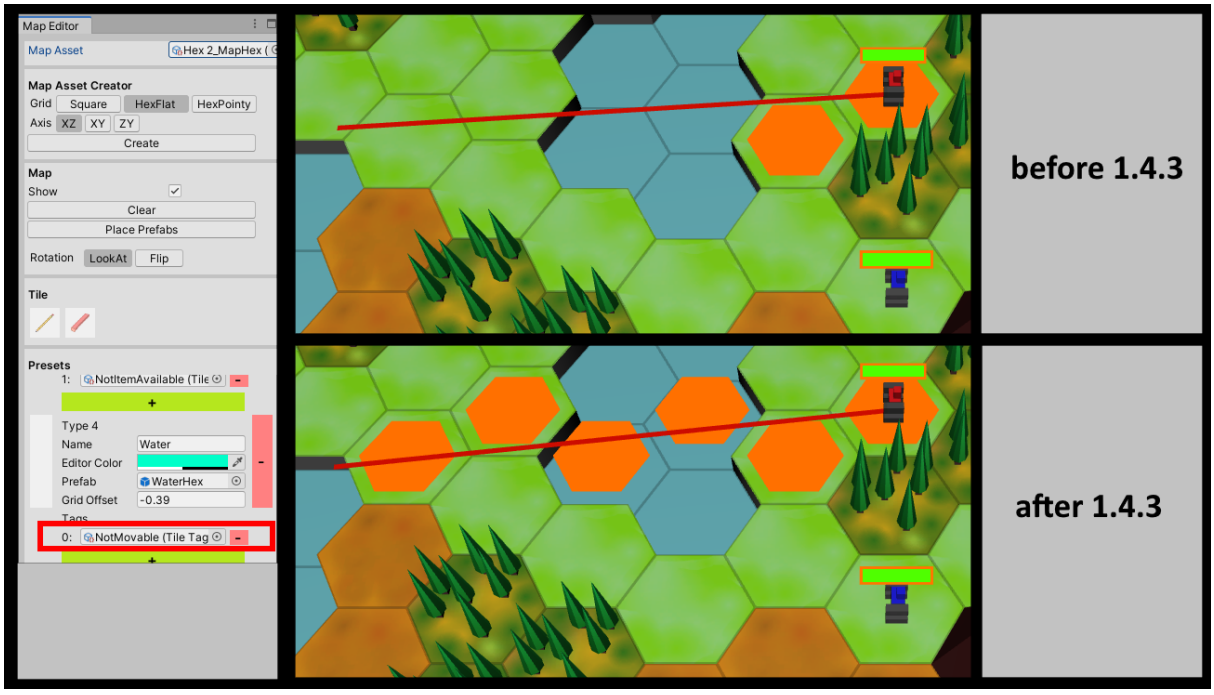
LevelSettings will be used by Menu scene to create load level buttons

```
21      void Start()
22      {
23          foreach (var p in S.Levels.Data.OrderBy(p => p.Caption))
24          {
25              CreateLevelButton(p, ButtonParent(p.Map.Type));
26          }
27      }
```

Default rule which defines tiles that can be selected by items = all tiles which don't contain NotItemAvailable tag

In versions prior to 1.4.3, this rule was hardcoded and instead of using its own tile tag it relied on the Vacant property of tile. In the picture below we could see that in older versions item couldn't not be applied through NotMovable tile (Water), but in newer versions it could, because it is not marked as NotItemAvailable

In the next picture we see the difference in the tile preset tag list which allow us to achieve item ban and move ban of the tile (Forest) in different versions




My turn
Type 2
Name
Forest
Editor Color
Prefab
ForestHex
Grid Offset
0
Tags
0: NotMovable (Tile Tag
before 1.4.3
Type 2
Name
Forest
Editor Color
Prefab
ForestHex
Grid Offset
0
Tags
0: NotMovable (Tile Tag
1: NotItemAvailable (Tile
after 1.4.3

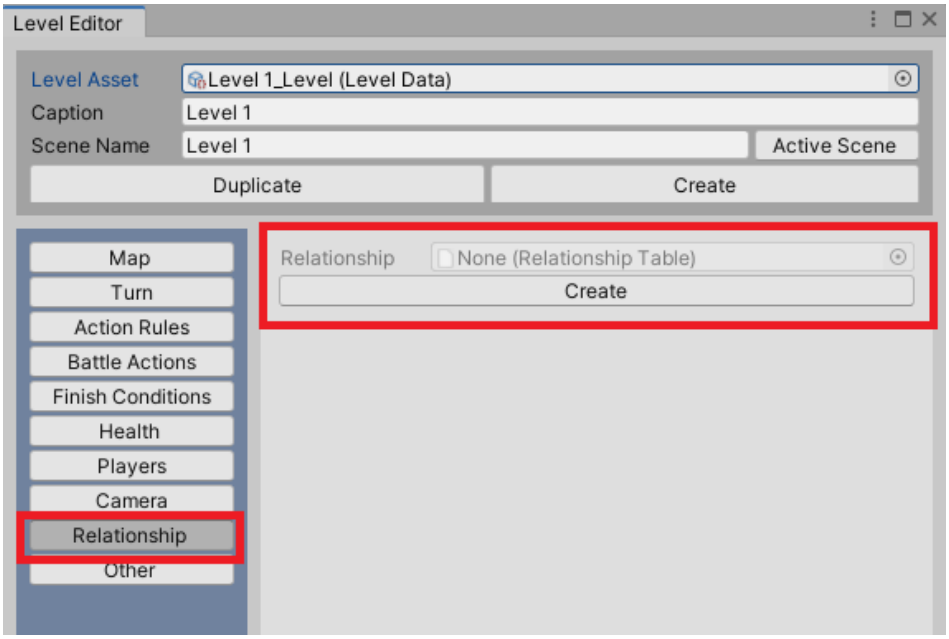
BattleFinish

This handler is about conditions which define should the game be finished or not. They are checked at the end of every turn. Also, this handler determines the winners. Current version has two options:

1.OneSquadLeft defines finish of the game when less than or equal to one squad left
2.OneAllyTeamLeft defines finish of the game when less than or equal to one ally left. This means that we divide the whole number of players into ally sides and if only one ally side left and all of its enemies are dead then the game finishes.

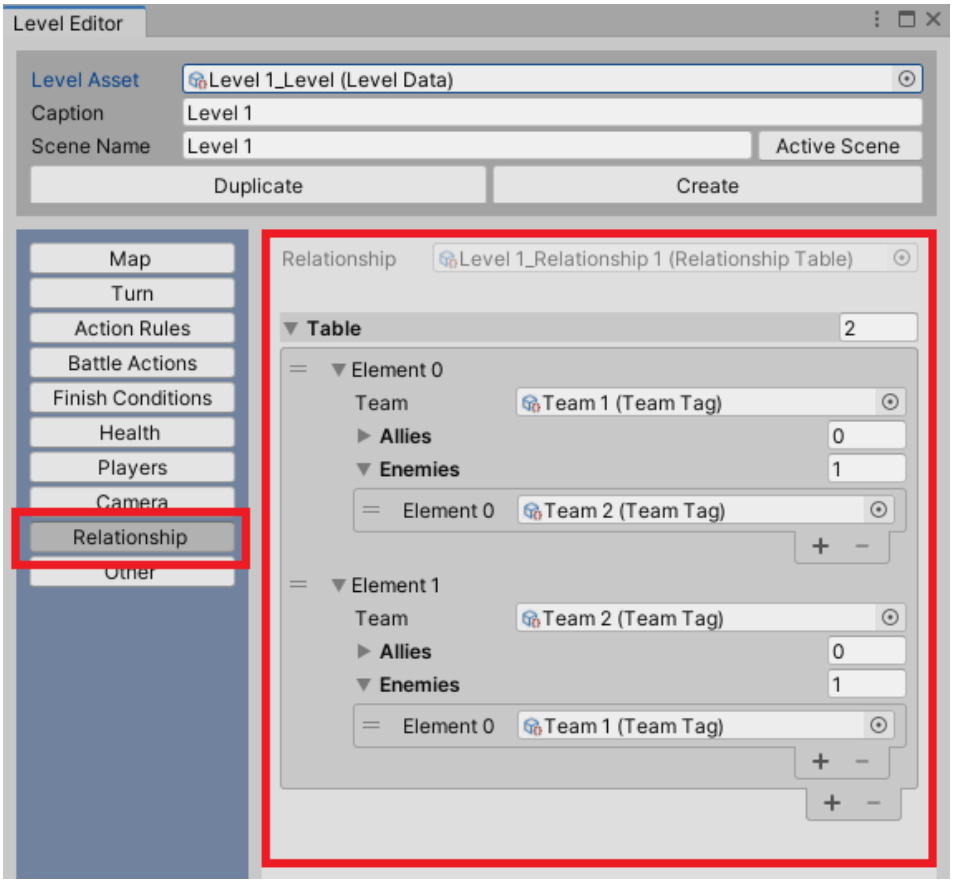
Relationship

If the level was created in SuperTiles version before 1.4.5 and SuperTiles Multiplayer before 1.1.6 this field would be null.



If Relationship is equal to null (like in the screenshot above) then every team has no allies and any other team is considered as an enemy.

Create button will help you to create an instance of RelationshipTable class with the mentioned algorithm (no allies, all opponents are enemies)



As you can see in the screenshot above the basic element of the relationship is TeamTag. So to make units allies or enemies you should specify them with different TeamTag. You could look up through example level Square 1, where Team 1 and Team 3 are allies and fight against Team 2 and Team 4 (which are also allies among themselves)

On Battle Start Action

SuperTiles 1.4.7 supports only adding resources action on the start of the battle

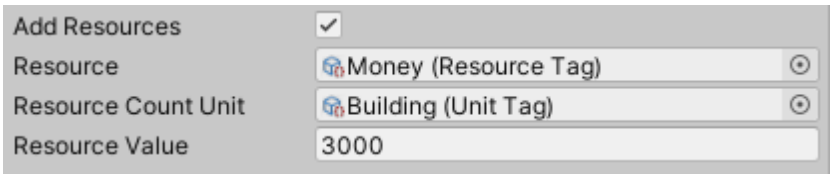
- Add Resources - enables/disables action
- Resouce - the resource type which should be added
- Resource Value - the amount of Resouce that should be added

On Turn Start Action

Action which is executed every turn at the start of turn

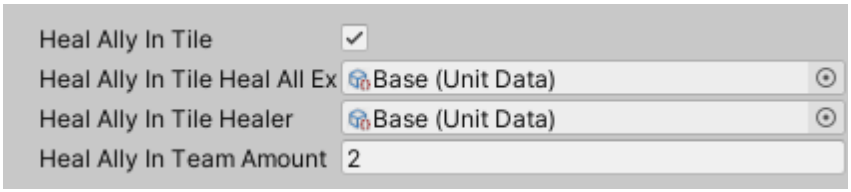
SuperTiles 1.4.7 supports only three action type

1. Adding resources - add resources using formula: The amount of units that contain specific tag multiplied by the given value
Example: Add (3000 x Amount of Building units which is contolled by current player) money to the player



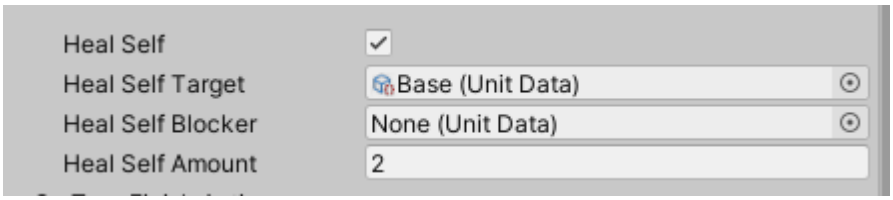
- Add Resources - enables/disables action
- Resource Count Unit - the amount of units that contain specific certain tag
- Resource Value - the value which would be multiplied by previous value

2. Healing unit which shares the tile with the specific ally unit
Example: All Base units heal 2 health points to ally units which share the owner's tile and their UnitData is not equal Base



- Heal Ally In Tile - enables/disables action
- Heal Ally in Tile Heal All Except - units which are NOT heal targets
- Heal Ally In Tile Healer - units which are healers
- Heal Ally In Team Amout - the amount of the heal

3. Self heal
Example: All Base units heal 2 health points to themselves when there is no any other unit in their tile



- Heal Self - enables/disables action
- Heal Self Target - what units should be self healed
- Heal Self Blocker - what kind of units block self healing proccess if they occupy the same tile as the heal target unit
- Heal Self Amount - the amount of the heal power

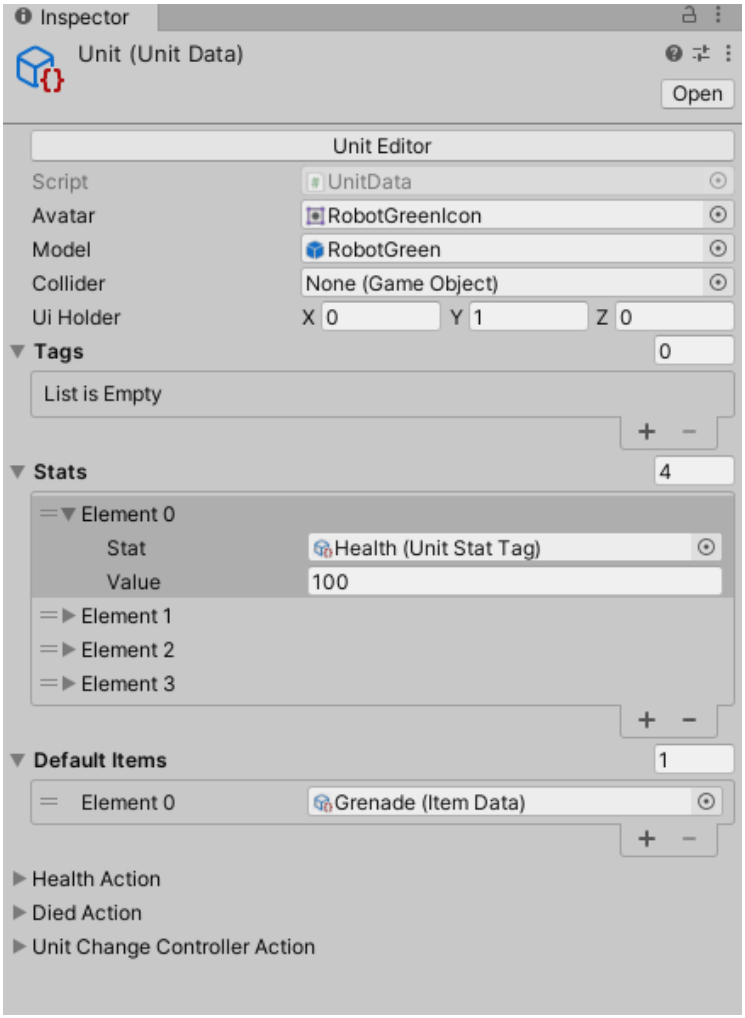
On Turn Finish Action

in development

Unit

Unit in *SuperTiles* consists of three main components: UnitData, UnitEntity and UnitView

UnitData is an information storage with default values like Health or MoveRange.



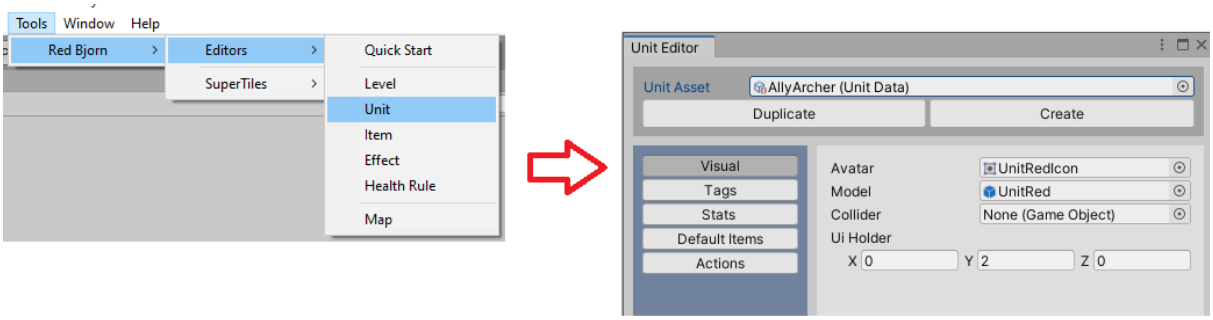
UnitData contains:

- Visual data (Avatar, prefab Model, custom Collider prefab, UI Holder position)
- Tags which would be used for gameplay logic (for example Capture item affects only units with the Building tag)
- Stats information (Health max value, current Move Range value, etc.)
- List of Default items which would be added to unit at the Game start
- Health Action - should be default health bar be replaced with custom one?
- Died Action
    - Rebirth - should be the unit resurrected after the death?
    - Killer Take Control - should the unit change his squad controller to the same as the killer

has?

- Unit Change Controller Action
    - Should be the color of unit's model changed when the unit's squad controller is changed?

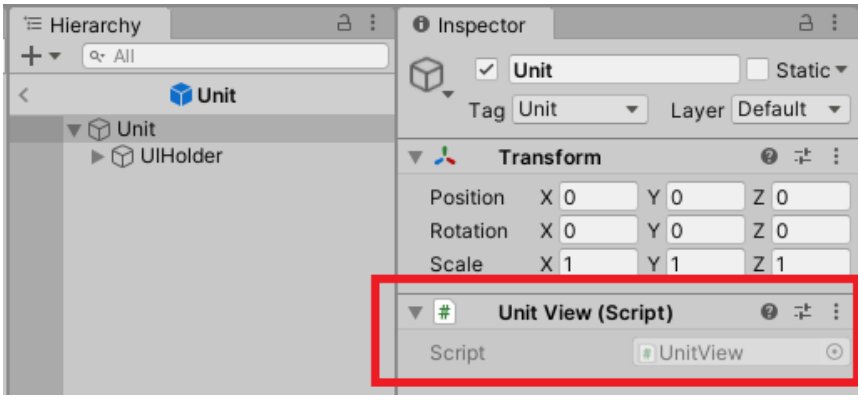
Unit Editor window was designed to simplify Unit duplication and editing



UnitEntity is a wrapper of UnitData which contains actual unit state

- Current unit id
- Values of unit stats
- Is this unit incorporeal or not?
- Is it alive or not?
- Effects which are on the unit, etc.

UnitView is a visual representation of UnitEntity. It inherits MonoBehaviour class and attached to the root gameobject of Unit prefab.

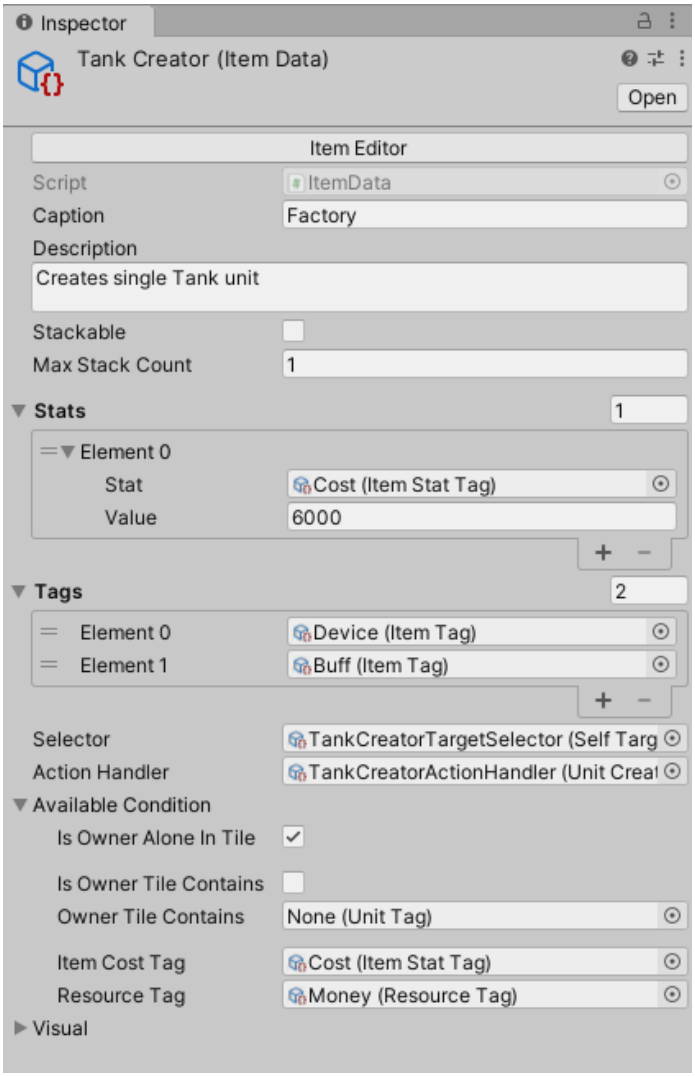


UnitView view contains information about

- The reference to the model gameobject that was instantiated for the Unit
- Unit world space position and rotation

Item

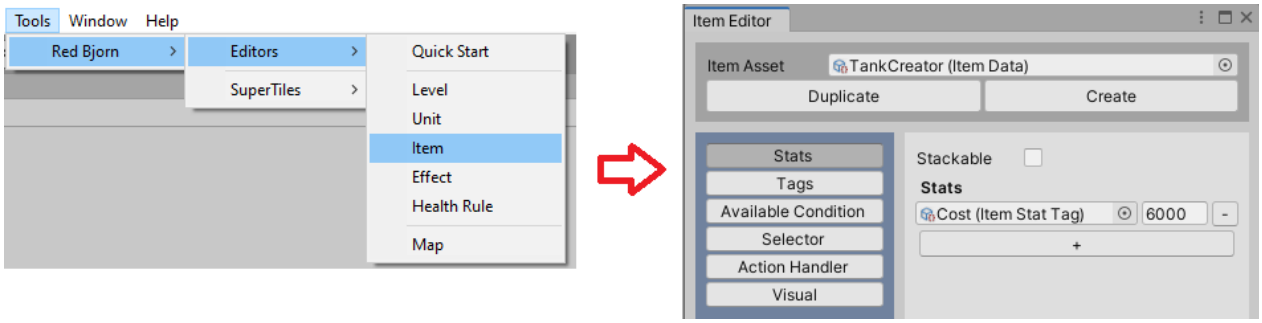
Item in *SuperTiles* consist of three main components: ItemData, ItemEntity and ItemAction.



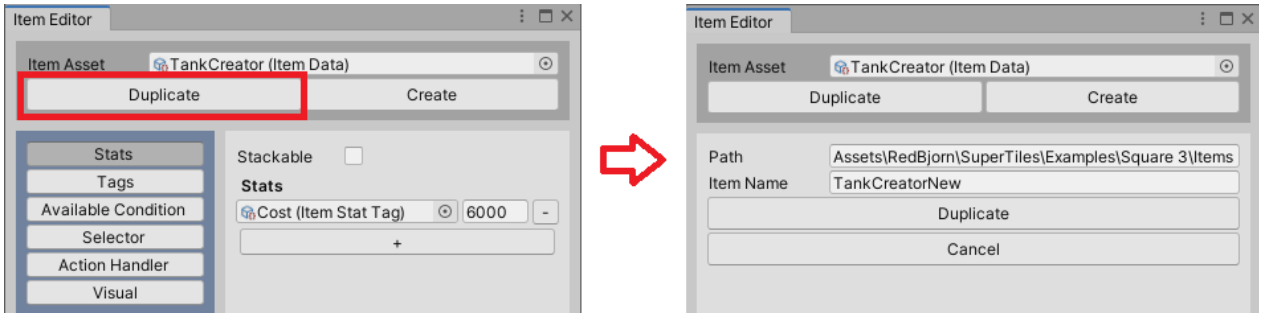
ItemData is an informational storage with default values like Maximum Cast Range or Power. Besides Stats information, ItemData contains visual data (Caption, Icon, Color, and Selector material) and rules:

- Tags – item markers for gameplay logic (Health Rules)
- Selector field – the logic of creating ItemAction
- ActionHandler field– the way ItemAction should be played

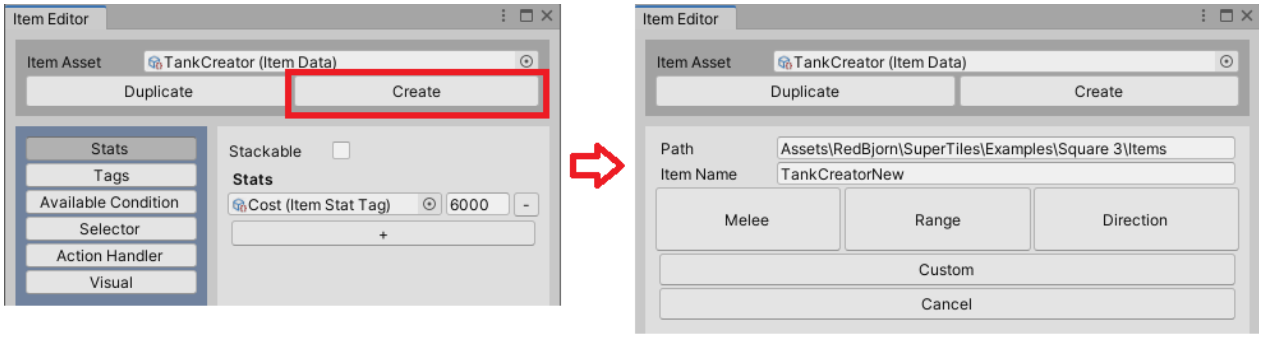
ItemData consists of nested ScriptableObjects and the Item Editor window was designed to simplify the edit process.



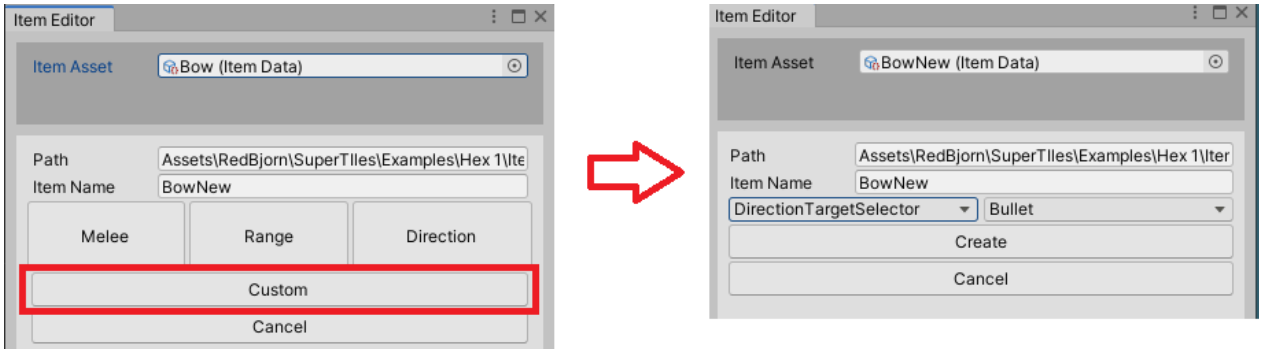
Here you can duplicate selected Item



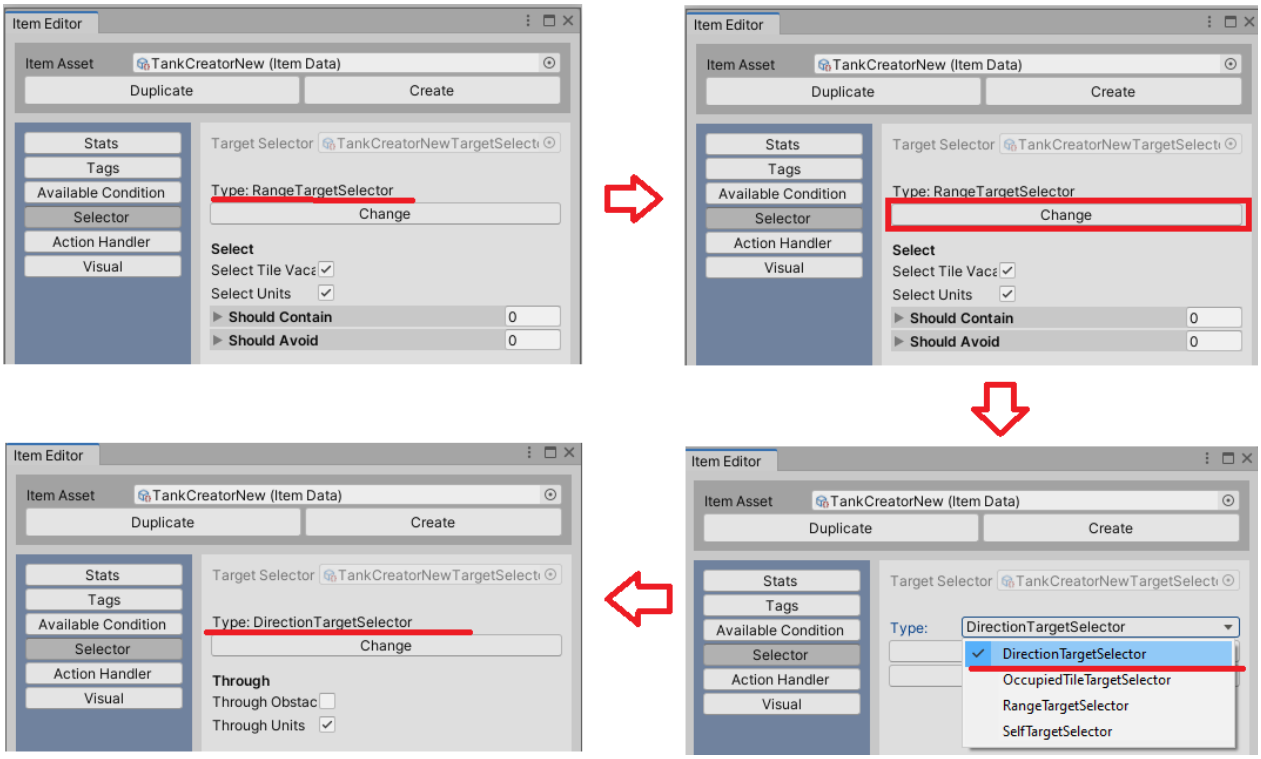
Or create a new one from common templates: Melee, Range and Direction



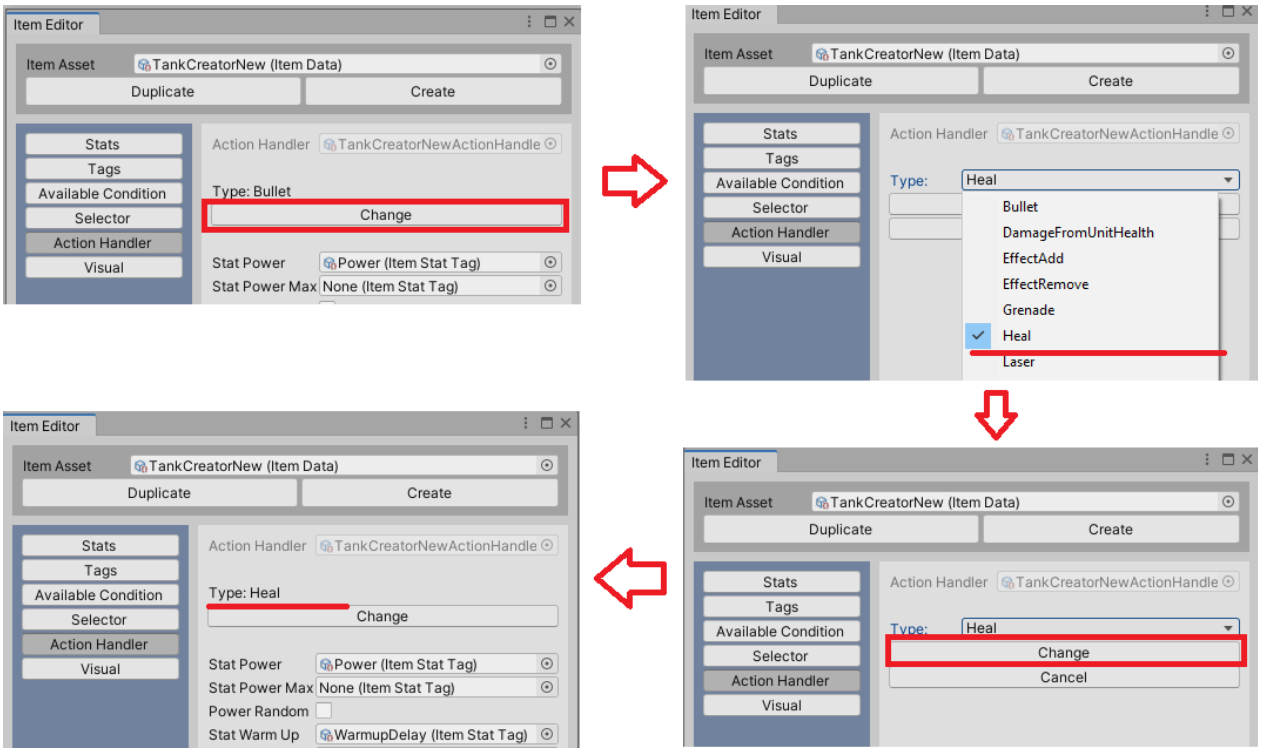
Or you can create an item in manual manner by clicking Custom field and specifying Selector and ActionHandler types



If you need to change Selector type of existed ItemData you should do steps shown in the picture below

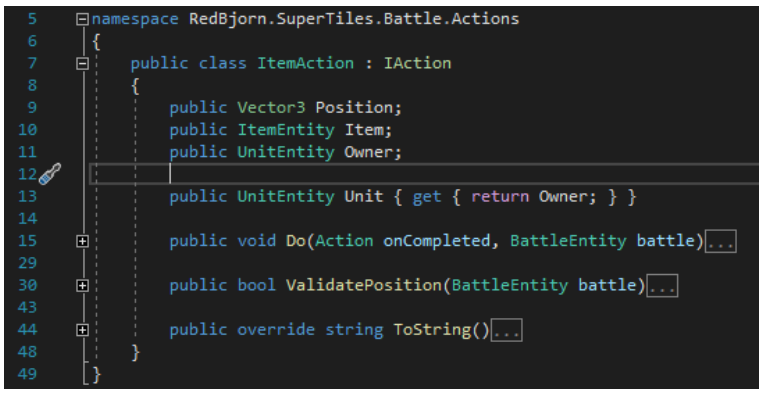


Also, if you need to change ActionHandler type you should do steps shown on the picture below



ItemEntity is a wrapper for an ItemData and handles the data and also contains the item state:

- CurrentCooldown
- CurrentStackCount
- Was it used or not?
- Etc.



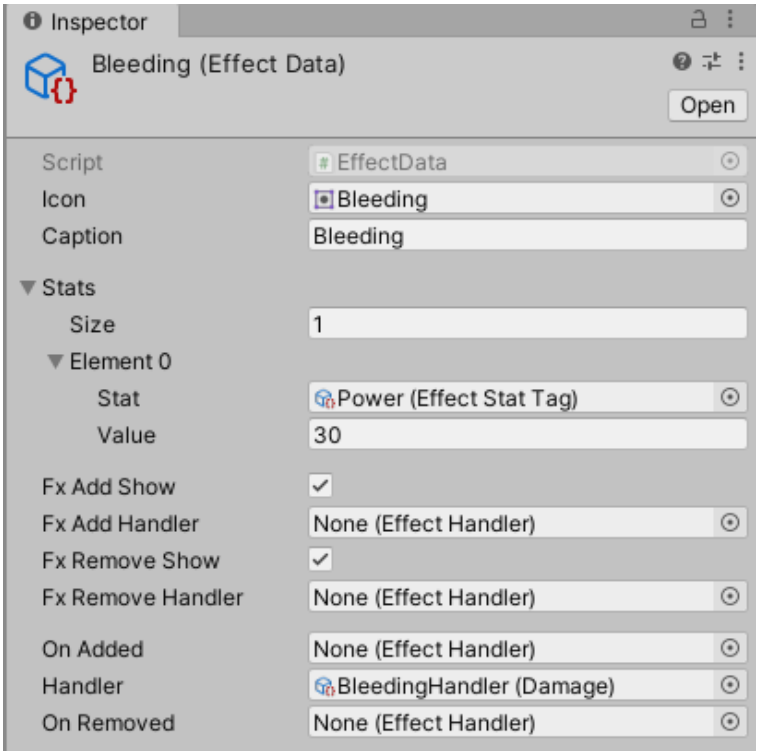
```
namespace RedBjorn.SuperTiles.Battle.Actions
{
    public class ItemAction : IAction
    {
        public Vector3 Position;
        public ItemEntity Item;
        public UnitEntity Owner;

        public UnitEntity Unit { get { return Owner; } }

        public void Do(Action onCompleted, BattleEntity battle)...

        public bool ValidatePosition(BattleEntity battle)...

        public override string ToString()...
    }
}
```

ItemAction is the result of the ItemData -> Selector work. It is easy to read ItemAction as

Unit uses Item at the Position.

Effect

Effect in *SuperTiles* consist of 2 main components: *EffectData*, *EffectEntity*.

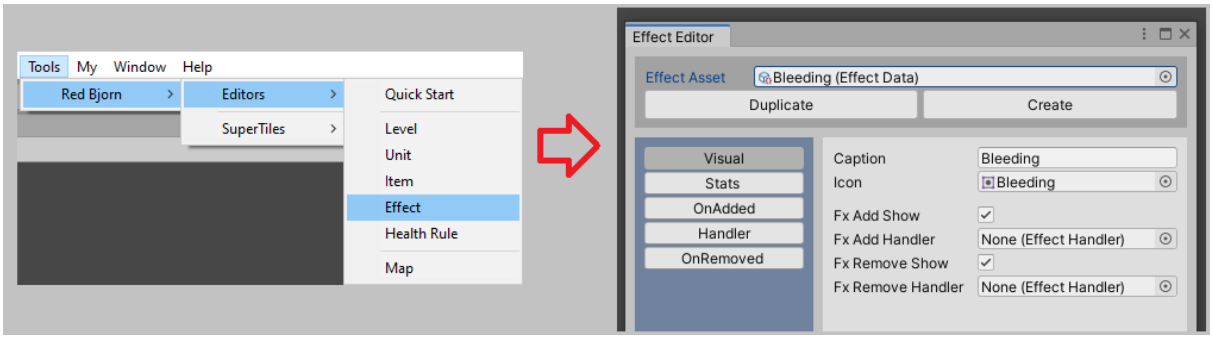


First one is *EffectData* which is an information storage with Stats values, Visual info, and the logic. Logic consists of 3 parts:

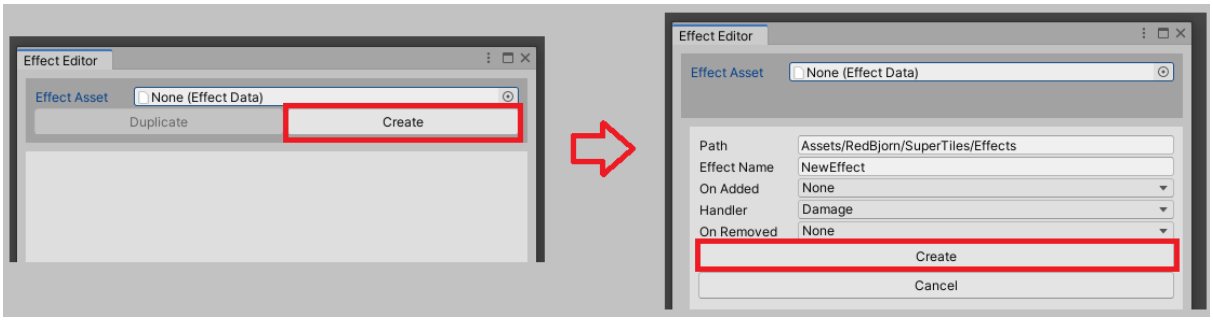
- **OnAdded** executes when effect was added to Unit
- **Handler** executes every owner's turn, before all actions
- **OnRemoved** executes when the effect was removed from the Unit.

For example, **Bleeding** effect in the screenshot above haven't got **OnAdded** and **OnRemoved** logic. Consequently, effect only deals damage (**Handler** has **Damage** type). Also, you could play Fx when the effect will be added or removed by checking corresponding toggles. One moment: If **Fx Add (Remove) Handler** is **None** – Default Fx (located inside asset **EffectFxAddDefault** (**EffectFxRemoveDefault**)) will be played

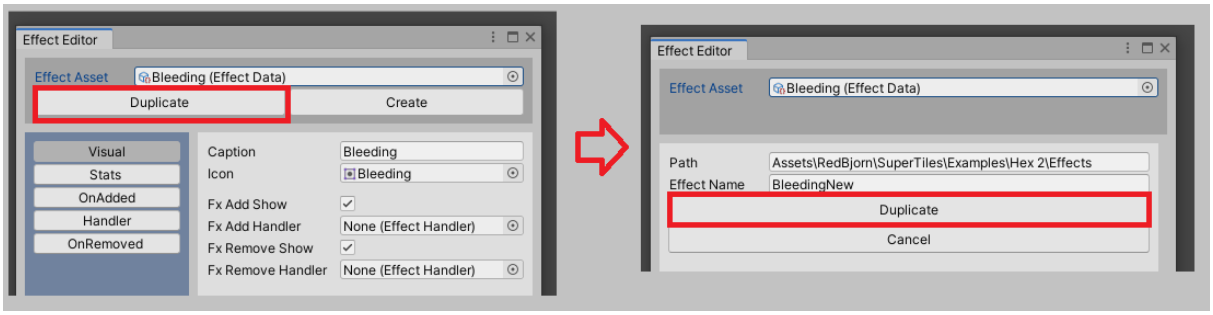
EffectData consists of nested ScriptableObjects and Effect Editor window was designed to simplify edit process



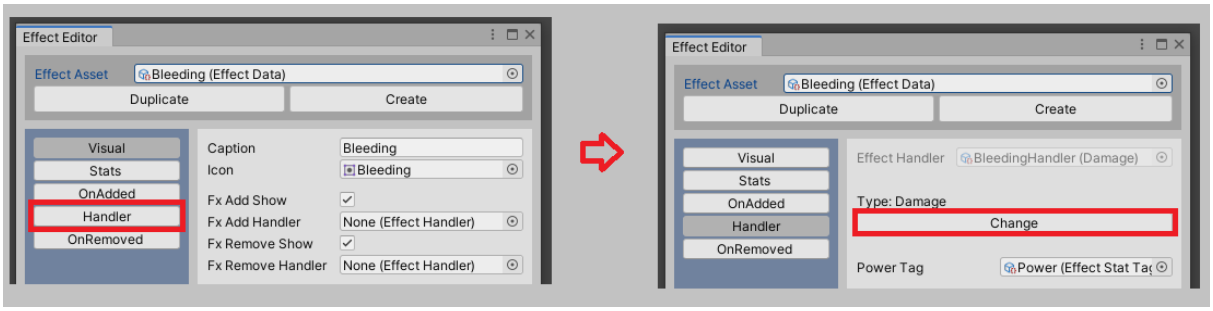
Here you can create a new one



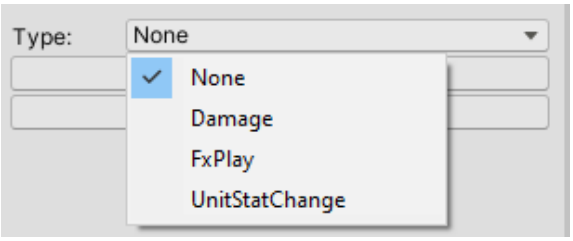
Or duplicate selected *EffectData*



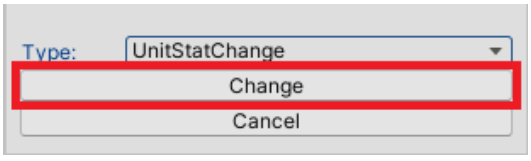
If you decide to change any logic (OnAdded, Handler, OnRemoved), just select corresponding tab and Click Change button



Select new type from popup. If you select **None** than existing logic will be deleted.



And click Change to submit your decision



If you create Effect without any logic, Effect transforms into StatusEffect. It does nothing but could be a marker for any other logic. By other logic, we mean Health Rules

There are 3 types of Handler logic:

1.Damage – deal damage every unit turn. Power is defined as a stat in Stats list

2.FxPlay – Spawn object for specified duration and then destroy it
3.UnitStatChange – change specified unit stat

List of Handlers will be extended in future updates, but you could easily implement any logic by yourself. For example, you decide to create an effect that will kill a unit when it is removed. Of course, you could create effect that deal -9999999 damage, but sometimes this way could lead to unexpected results (input damage could be converted to a small value and the unit will survive). Let's assume that we still decide to create our own logic, then we should create a new class **Kill** which should inherit the **EffectHandler** class.

```
namespace RedBjorn.SuperTiles.Effects
{
    /// <summary>
    /// Base class for EffectHandler with initial validation checks
    /// </summary>
    public abstract class EffectHandler : ScriptableObjectExtended
    {
        public IEnumerator Handle(EffectEntity effect, UnitEntity unit, BattleEntity battle)
        {
            yield return DoHandle(effect, unit, battle);
        }

        protected abstract IEnumerator DoHandle(EffectEntity effect, UnitEntity unit, BattleEntity battle);
    }
}
```

As you could see, EffectHandler has only one abstract method which just needs to be implemented.

Something like

```
protected override IEnumerator DoHandle(EffectEntity effect, UnitEntity unit, BattleEntity battle)
{
    UnitEntity.Kill(unit, battle);
    yield break;
}
```

Method **Kill** is underlined because there is no such method, so I wrote it just for example. There is no doubt that you could implement it by yourself.

And one more step, you should create a **Kill** class creator. To do this you need to create a new class which will implement the **EffectHandlerCreator** interface. It is needed for correct Effect Editor work.

```
public interface EffectHandlerCreator
{
    EffectHandler Create(EffectData effect, string type);
}
```

Examples could be found inside the **EffectHandlerCreator** file.

Important note: Creator class name should be made in the following form:

<Classname>Creator ⇨ KillCreator

Second important Effect component is *EffectEntity*.

```
namespace RedBjorn.SuperTiles
{
    /// <summary> Effect state
    [Serializable]
    public class EffectEntity
    {
        public Dictionary<EffectStatTag, StatEntity> Stats = new Dictionary<EffectStatTag, StatEntity>();

        [NonSerialized] UnitEntity Owner;

        public int Duration { get; private set; }
        public EffectData Data { get; private set; }
        public SpriteRenderer UI { get; private set; }

        public StatEntity this[EffectStatTag stat] { get { return Stats.TryGetOrDefault(stat); } }

        public EffectEntity(EffectData effect)...

        public EffectEntity(EffectData effect, int duration, UnitEntity owner)...

        public override string ToString()...

        public IEnumerator OnAdded(BattleEntity battle)...

        public IEnumerator Handle(BattleEntity battle)...

        public IEnumerator OnRemoved(BattleEntity battle)...

        public void DurationAdd(int delta)...
    }
}
```

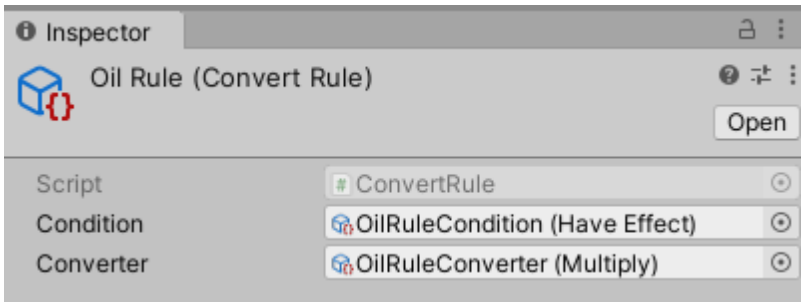
Here we store the Effect state at the runtime. Effect state mainly consists of effect current duration and several wrappers of *EffectData* logic methods.

Usually, Effects could be added from Item -> ActionHandler. More info here.

Effects will be visualized near the unit avatar and there will be an effect mark near the unit model.

HealthRules

It is a storage for a list of rules. Each rule is a single ScriptableObject of *CovertRule* type.



CovertRule consists of two fields: Condition and Converter. It could be read as If **Condition** is true then convert input delta value with **Converter**.

Condition field has a **Condition** class type which is an abstract class with one method.

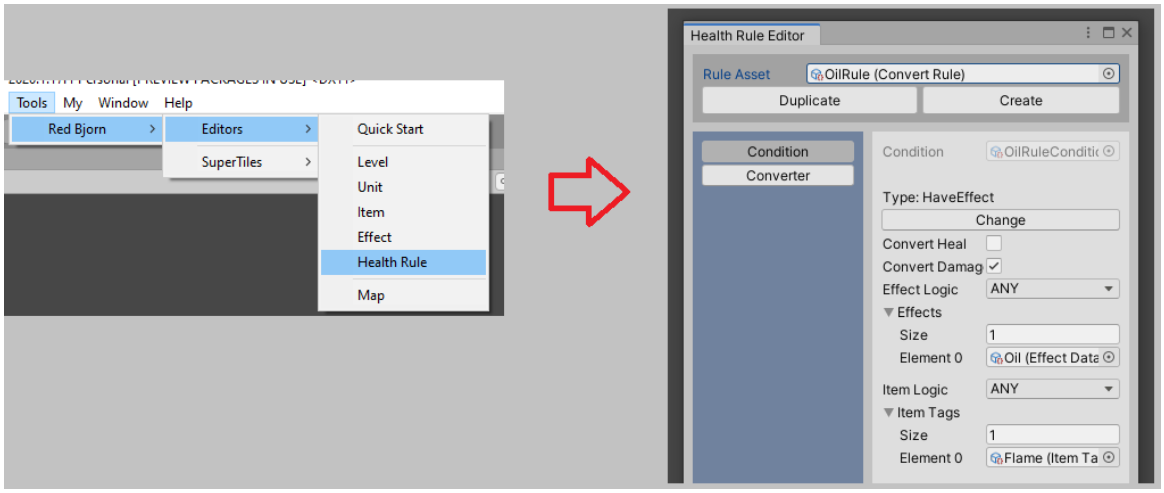
```
namespace RedBjorn.SuperTiles.Health
{
    public abstract class Condition : ScriptableObjectExtended
    {
        public abstract bool IsMet(float delta, UnitEntity victim, UnitEntity damager, ItemEntity item);
    }
}
```

You can see that condition could be checked with any of the victim, damager and item or all of them at once.

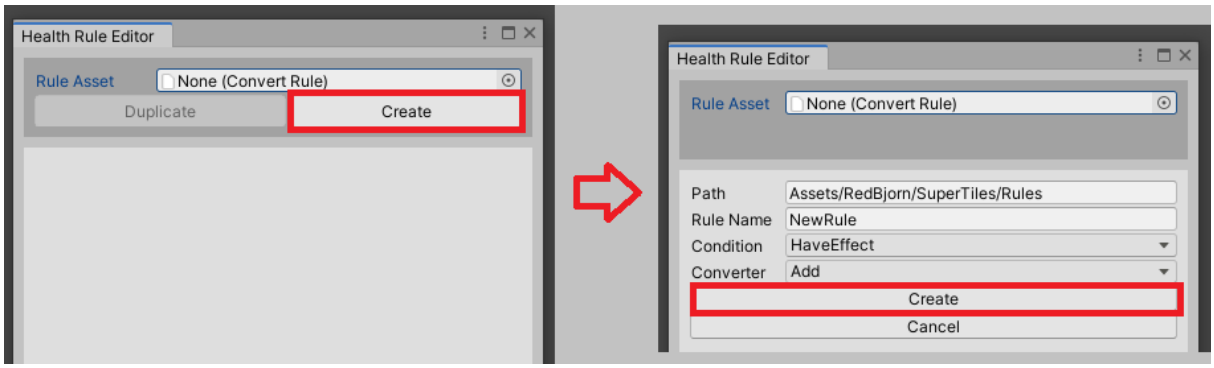
Converter field has a **ValueConverter** class type which is also an abstract class with one method.

```
namespace RedBjorn.SuperTiles.Health
{
    public abstract class ValueConverter : ScriptableObjectExtended
    {
        public abstract float Convert(float val);
    }
}
```

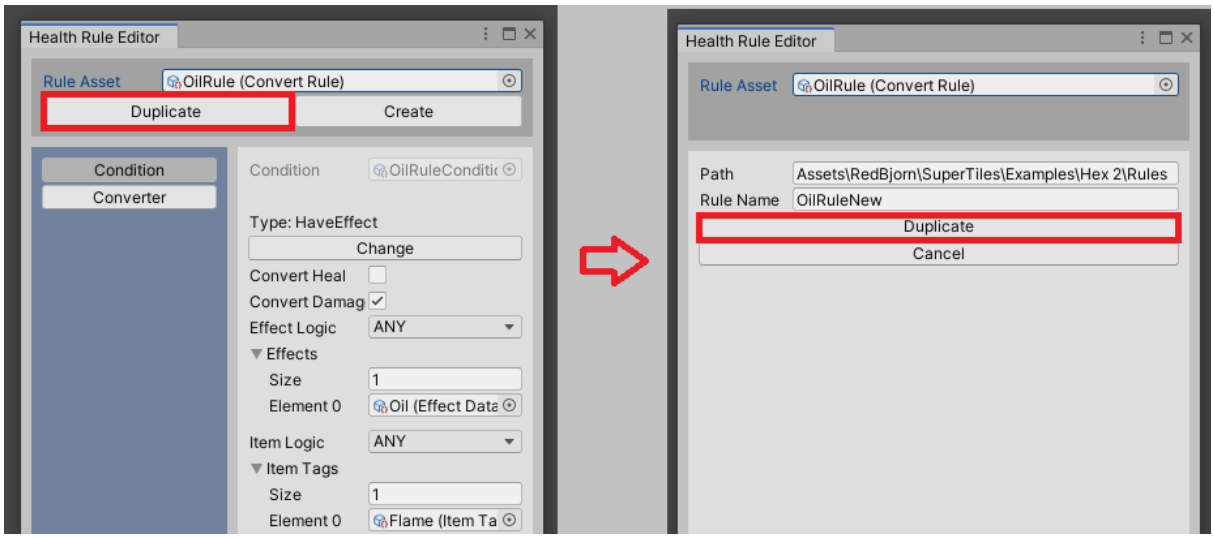
Health Rule Editor window was designed to simplify the process of editing single rule.



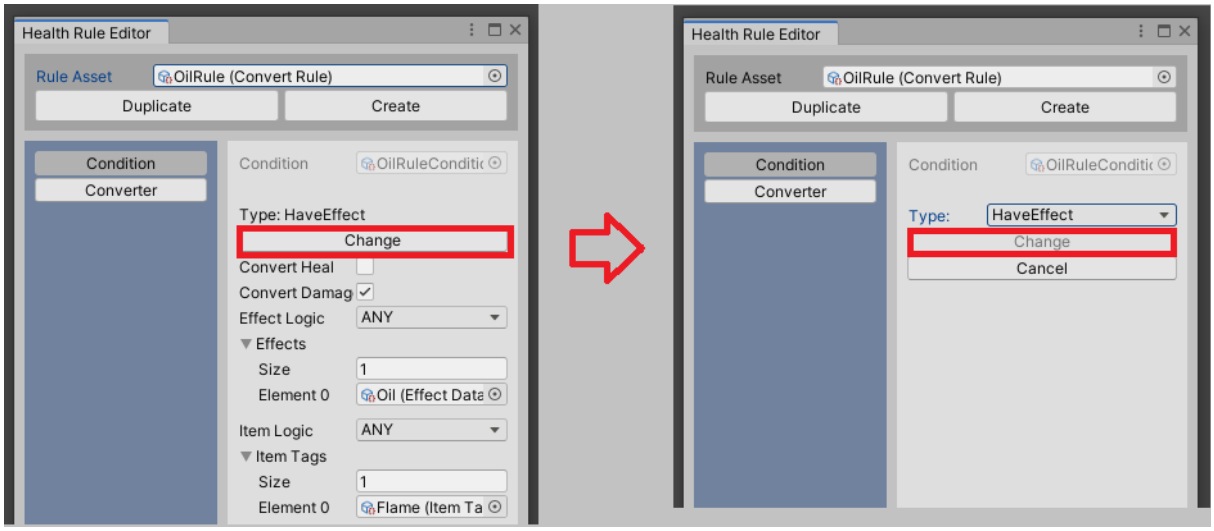
Here you can create a new one



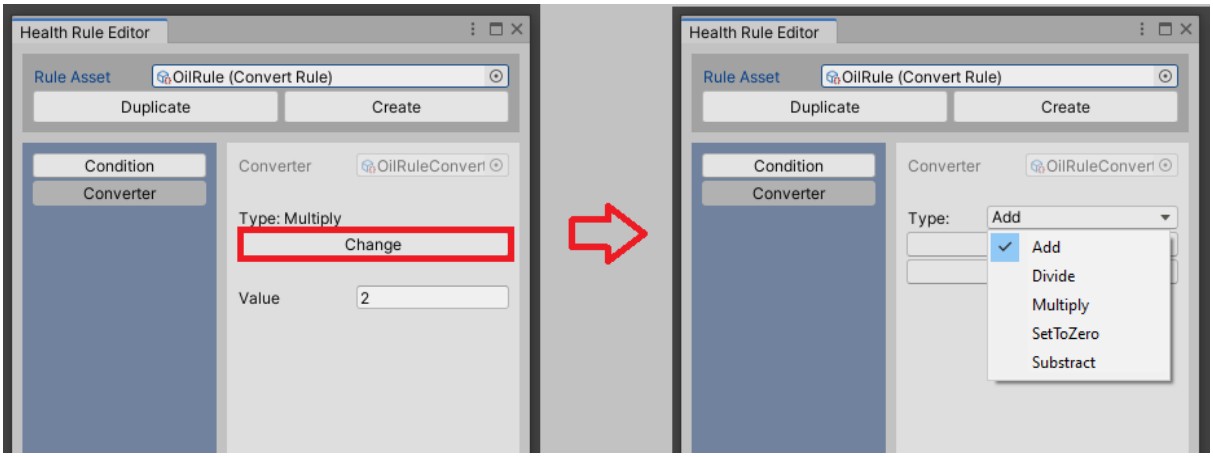
Or duplicate selected CovertRule



Also, you could change **Condition** type by clicking Change button



And you could change **Converter** type by clicking Change button



To implement your own **Conditions** and **Converters** you should create corresponding Creator classes besides:

```
public interface ConditionCreator
{
    Condition Create(ConvertRule rule, string type);
}
```

```
public interface ValueConverterCreator
{
    ValueConverter Create(ConvertRule rule, string type);
}
```

It is similar to EffectHandler creation.

AI

AI in *SuperTiles* consist of 2 main components: *UnitAiData* and *UnitAiEntity*

UnitAiData represent abstract class which successors represent Ai logic by implementing `TryNextAction` method

```
10  namespace RedBjorn.SuperTiles
11  {
12      public abstract class UnitAiData : ScriptableObjectExtended
13      {
14          public abstract bool TryNextAction(UnitAiEntity ai, BattleEntity battle, out IAction action, int index);
15
16          public static UnitAiData FindDefault()...
37
38  #if UNITY_EDITOR
39          public static UnitAiData Create(string folderPath, string name, string type)...
66  #endif
67      }
68  }
```

UnitAiEntity is a wrapper of *UnitAiData* containing the runtime state of Ai:

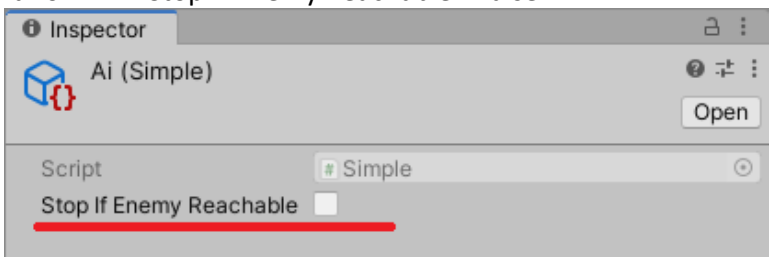
- unit it controls
- actions count, which were already scheduled for current AI.

```
3   namespace RedBjorn.SuperTiles
4   {
5       public class UnitAiEntity
6       {
7           public UnitAiData Data;
8           public UnitEntity Unit;
9           public int TurnActionCount { get; private set; }
10
11          public bool TryNextAction(BattleEntity battle, out IAction action)...
20
21          public void OnEndTurn()...
25
26          public override string ToString()...
30      }
31  }
```

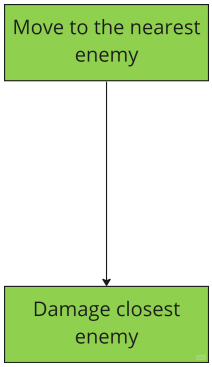
Current version of *SuperTiles* contains two AI behaviors: Simple and Universal

Simple

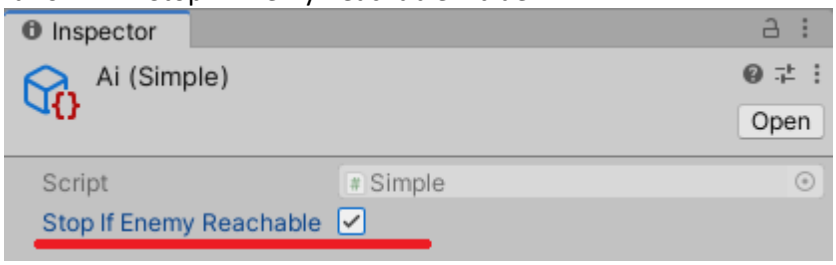
Simple behaviour implement two scenarios
Scenario 1. Stop If Enemy Reachable = false



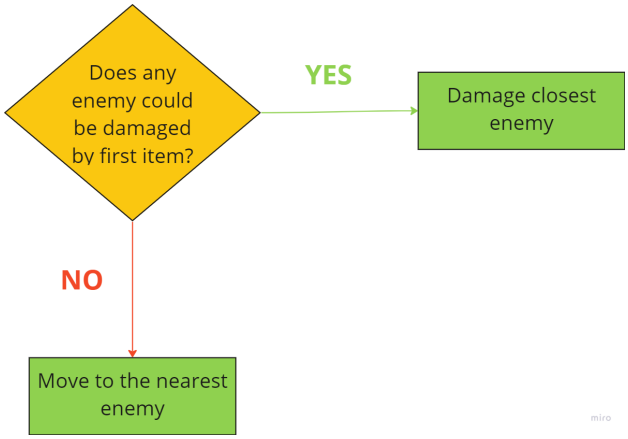
Simplified behavior diagram



Scenario 2. Stop If Enemy Reachable = true



Simplified behavior diagram

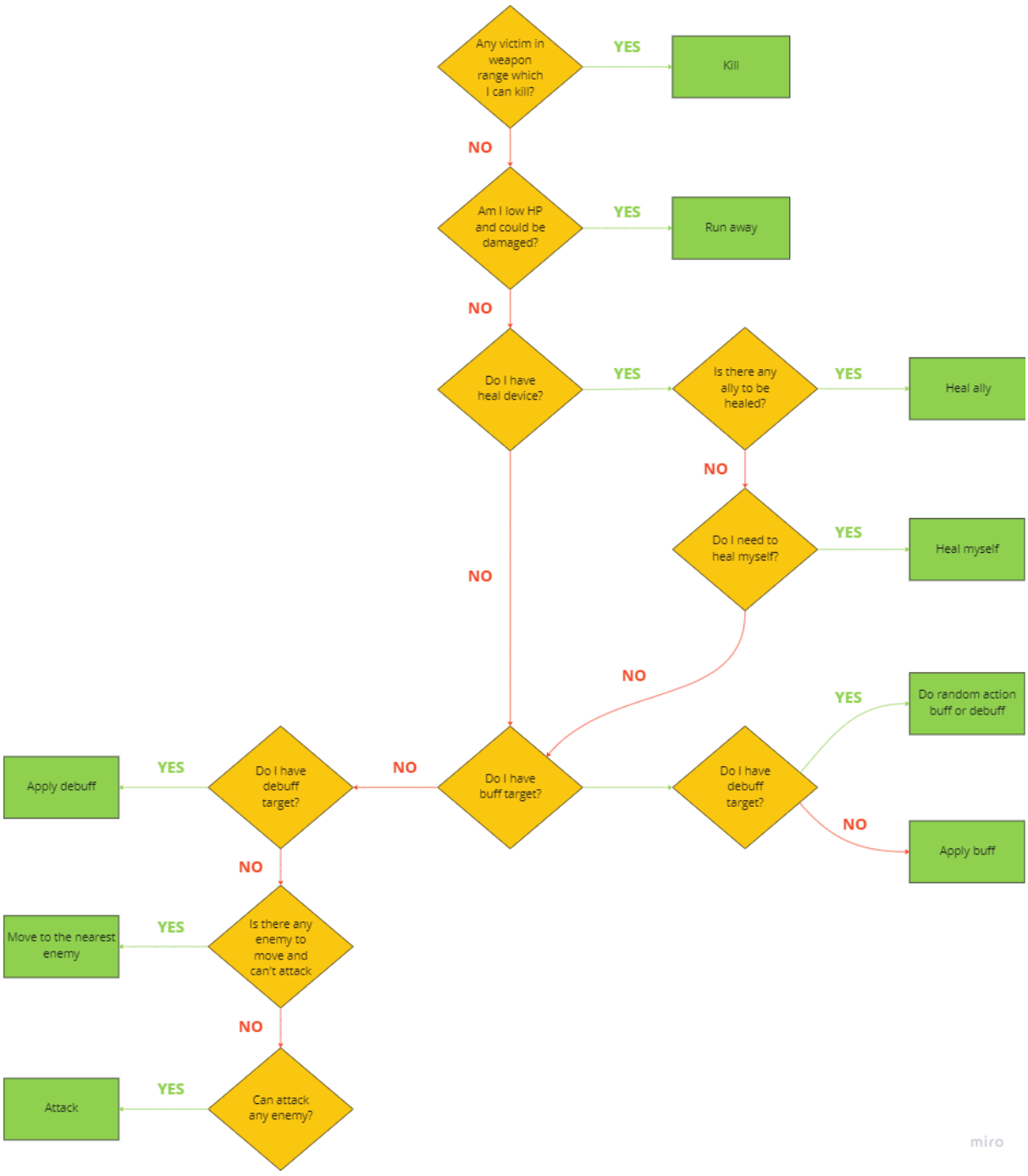


Universal

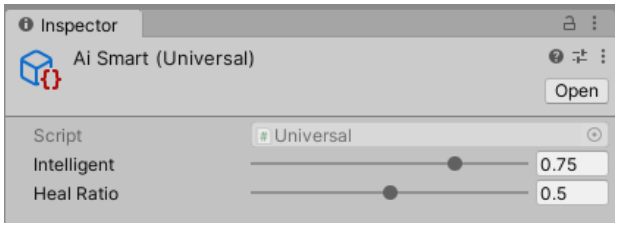
Universal AI behavior requires item tags to be specified on ItemData.

- Weapon
- Device
- Heal
- Buff
- Debuff

Simplified behavior diagram:



Tunable options



- **Intelligence** is a change (**Random(0, 1) < 0.75**) to make best solution
- **Heal Ratio** is a percentage of maximum health value. If the current health value of the unit is below this value, then the unit accepts the decision to run away and heal

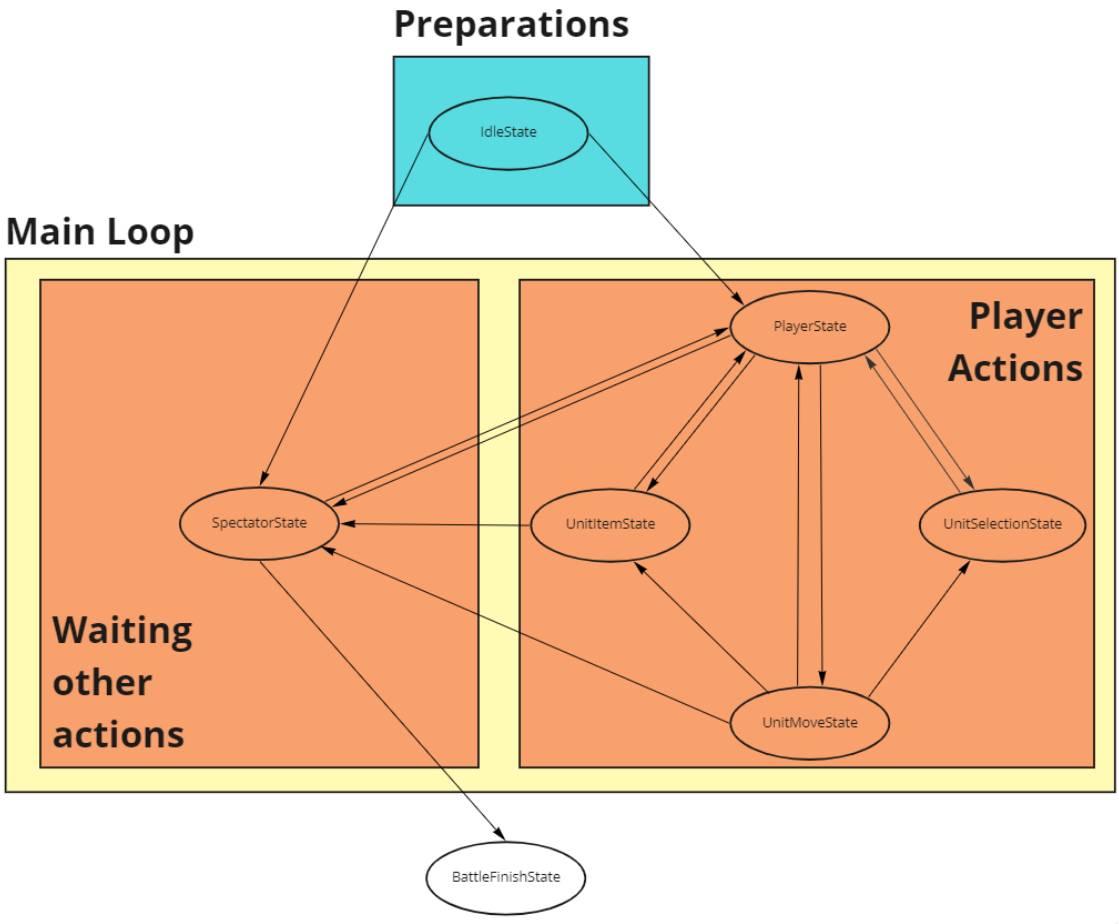
Battle

Battle in *SuperTiles* consist of 3 main components: *BattleSettings*, *BattleEntity* and *BattleView*

Information about *BattleSettings* is here.

BattleView is the main class which defines communication between all entities and visual game representation (**View** suffix indicates that this class is usually a MonoBehaviour successor and has corresponding entity class, this time it will be *BattleEntity*). This class is an implementation of Finite State Machine (FSM) which controls the Battle state.

In the picture below you could see a diagram of the FSM.



Preparations

BattleView starts from the Idle state where it does nothing. It only waits for internal actions (UI initializing, caching, etc.) to be done. After these actions are completed State Machine will go to the Main Loop.

Main Loop

Main Loop begins in the Spectator state for all players. Players wait at this state until BattleView receives the signal about the player which could make actions. This player could move freely

between available *Player actions*: Move, Item and UnitSelection. Other players still are in the Spectator state (*Waiting other actions*). When current player's turn is finished (at a will or the time is up), current player is sent to the Spectator state and BattleView receives another signal of the next selected player.

BattleFinish state is similiar to Idle state except predefined scripting behaviour (calculating battle winners, spawning finish UI, etc.)

BattleEntity handles business game logic and contains current Battle state including:

- The number of current turn
- Current turn state (Starting, Started, Finishing, Finished)
- Battle state (Started, Finished)
- Players who participate in current Battle
- Dead and alive units
- Ai instances
- and more

Save

Save in *SuperTiles* is organized as a composition on C# plain classes. As Unity doesn't support reference serialization by default, all fields with reference types inherited from *System.Object* should be marked with SerializeReference attribute.

```
namespace RedBjorn.SuperTiles
{
    /// <summary>
    /// Game state
    /// </summary>
    [Serializable]
    public class GameEntity
    {
        [SerializeReference]
        public IGameTypeCreator Creator;
        [SerializeReference]
        public IGameTypeLoader Loader;
        public bool Restartable;
        public int RandomSeed;
        public LevelData Level;
        public BattleEntity Battle;
```

Class which contains whole game info is called *GameSave*.

```
namespace RedBjorn.SuperTiles.Saves
{
    [Serializable]
    public class GameSave
    {
        public string Version;
        public string Timestamp;
        public GameEntity State;
    }
}
```

It consists only of 3 fields:

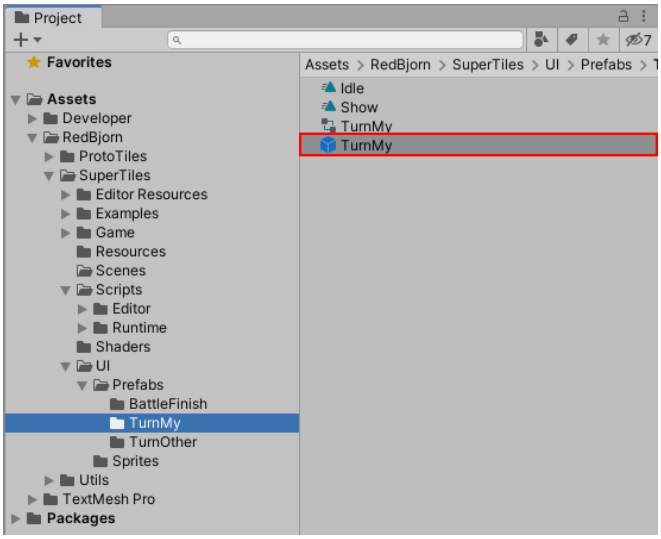
1.Version field to check compatibility between current build and save file. And if versions mismatch then several handlers could be made to update save file (not implemented it this asset)
2.Timestamp could be used to form appropriate save order.
3.GameEntity stores info about level that should be loaded and the battle state

SaveController creates an instance of *GameSave* class inside Save method logic.

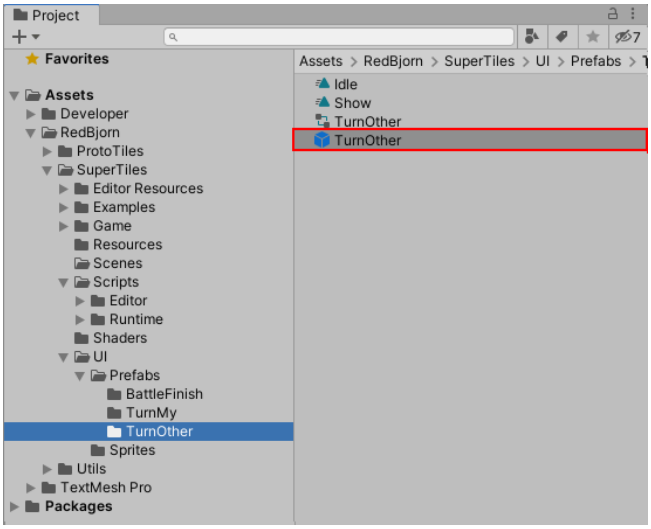
Labels

There are three UI labels which are presented at the beginning of the turn

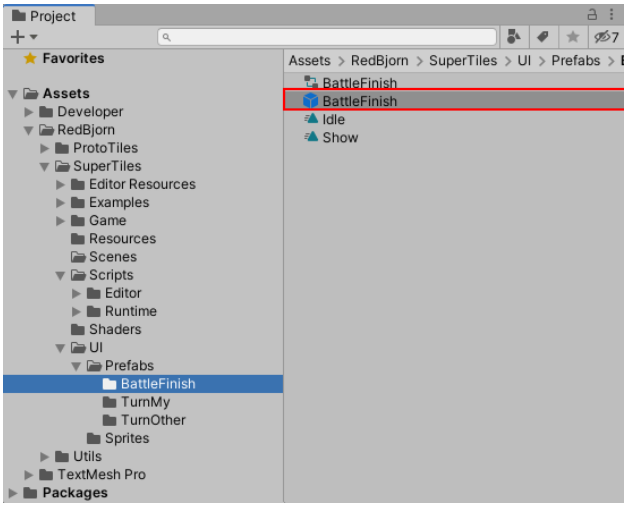
1.**TurnMy** prefab is shown when the player turn starts.



2.**TurnOther** prefab is shown when another player's turn starts.



3.**BattleFinish** prefab is shown when the battle finishes.


Project
Favorites
Assets
Developer
RedBjorn
ProtoTiles
SuperTiles
Editor Resources
Examples
Game
Resources
Scenes
Scripts
Editor
Runtime
Shaders
UI
Prefabs
BattleFinish
TurnMy
TurnOther
Sprites
Utils
TextMesh Pro
Packages
Assets > RedBjorn > SuperTiles > UI > Prefabs
BattleFinish
BattleFinish
Idle
Show

Controllers

In the *SuperTiles*, by **Controller** we mean the logic dedicated to a single aspect of the game logic which is available from everywhere. Frequently term **Manager** is used to define the same thing.

AudioController

Controller which could play any audio clip or already defined button click sound

CameraController

Controller which gets data from InputController and converts it to the movement of Camera

InputController

Controller which is a wrapper to UnityEngine.Input class with some quality-of-life features and custom restrictions.

SaveController

Controller which implements basic Save/Load/Delete logic for the specified slot. By slot we mean unique string value.

Save

To save the game, you should pass an instance of *GameEntity* to this controller and a string which represents a slot name.

```
/// <summary> Save game into a file
public static void SaveGame(GameEntity game, string savename, Action onSaved = null)
```

Also, you could specify an *Action* which will be played after save process will be finished (for example disable UI save icon)

Inside the SaveGame method, *GameEntity* will be serialized to an array of bytes, at first. Then the received array will be written to a file inside **<Application.persistentDataPath>/Saves** folder.

Load

To load the game, you should pass the name of a save file.

```
/// <summary> Load game from file
public static void LoadGame(string savename, Action<GameSave> onLoaded = null)
```

Also, you could specify an *Action* which will be played after load process will be finished (for example disable UI load icon)

LoadGame operates in reverse order in comparison to SaveGame method. At first, it reads bytes from the file. Then it deserializes bytes to an instance of the *GameSave* class.

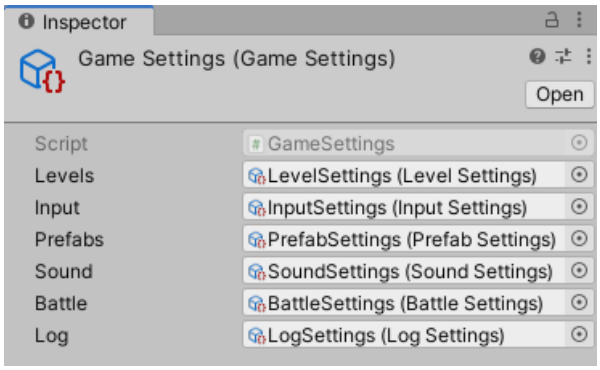
Delete

DeleteGame method is the simplest one. It just deletes the file with the specified name.

```
/// <summary> Delete save file
public static void DeleteGame(string savename, Action onDeleted = null)
```

GameSettings

As mentioned above, there is a root data object named *GameSettings* – an instance of *GameSettings* class.



It specifies how all settings will be loaded. From the screenshot below it can be seen that *GameSettings* asset should be inside the Resources folder and is loaded when it is called for the first time.

```
8           static GameSettings CachedGame;
9           static GameSettings Game
10          {
11              get
12              {
13                  if (CachedGame == null)
14                  {
15                      CachedGame = Resources.Load<GameSettings>("GameSettings");
16                  }
17                  return CachedGame;
18              }
19          }
```

This class is frequently used therefore to simplify reference calls (something like GameSettings.Instance.Levels) *StaticSettings* class was created.

```
namespace RedBjorn.SuperTiles
{
    public class S
    {
        static GameSettings CachedGame;
        static GameSettings Game
        {
            get
            {
                if (CachedGame == null)
                {
                    CachedGame = Resources.Load<GameSettings>("GameSettings");
                }
                return CachedGame;
            }
        }

        public static LevelSettings Levels { get { return Game.Levels; } }
        public static InputSettings Input { get { return Game.Input; } }
        public static PrefabSettings Prefabs { get { return Game.Prefabs; } }
        public static SoundSettings Sound { get { return Game.Sound; } }
        public static BattleSettings Battle { get { return Game.Battle; } }
        public static LogSettings Log { get { return Game.Log; } }
    }
}
```

For example, if you want to get LevelSettings somewhere in the code, you should call S.Levels. S for the first character of *StaticSettings*.

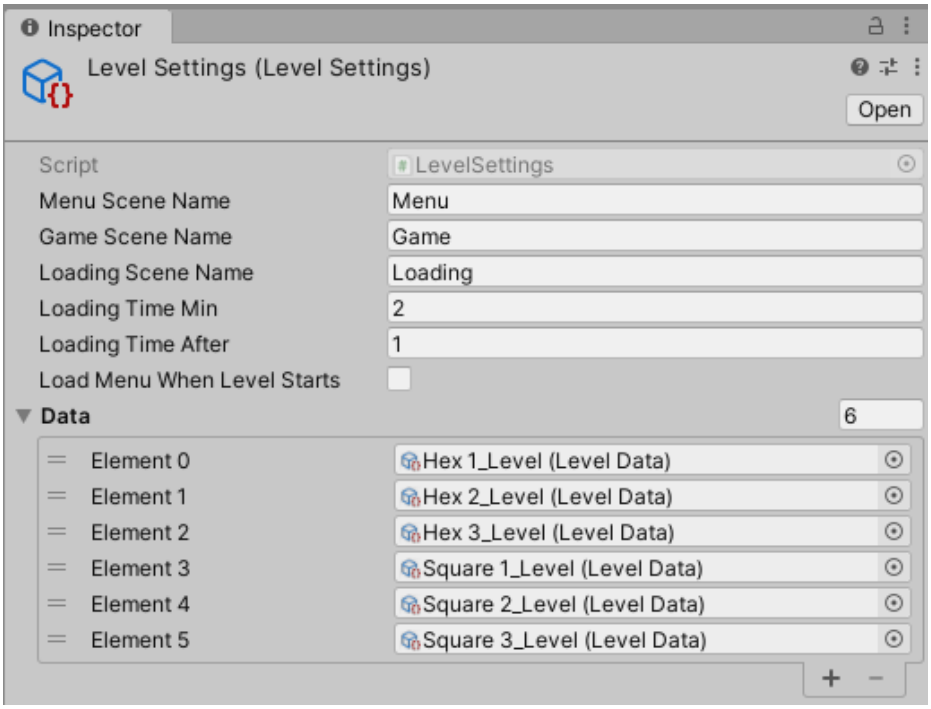
```
void Start()
{
    foreach (var p in S.Levels.Data.OrderBy(p => p.Caption))
    {
        CreateLevelButton(p, ButtonParent(p.Map.Type));
    }
}
```

LevelSettings

Contains data about helper scene names and list of gameplay LevelData

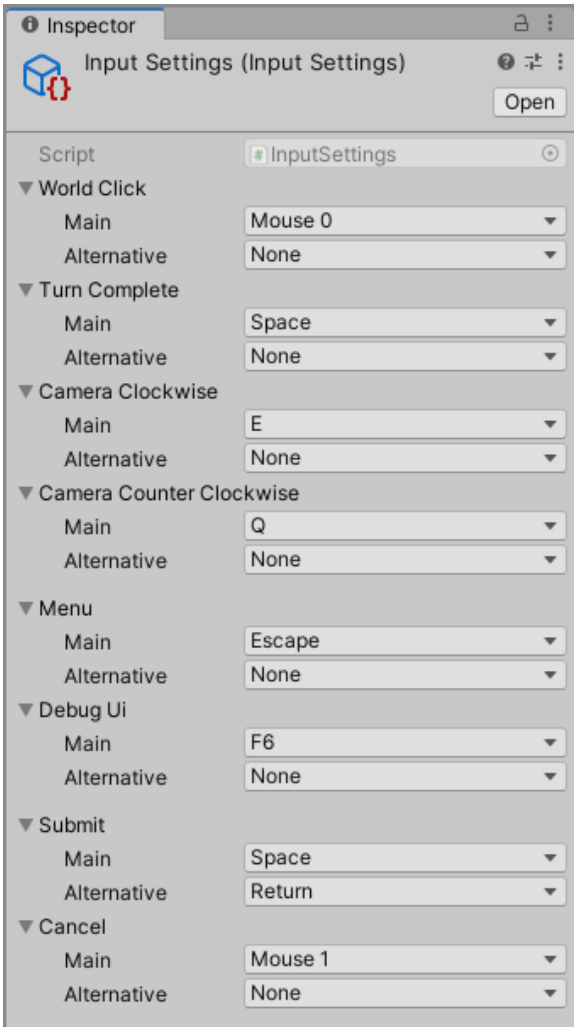
Loading Time Min is a minimal load level duration to avoid fast frame switching when lightweight scene is loading

Loading Time After is a duration between disabling loading screen and enabling game controls.



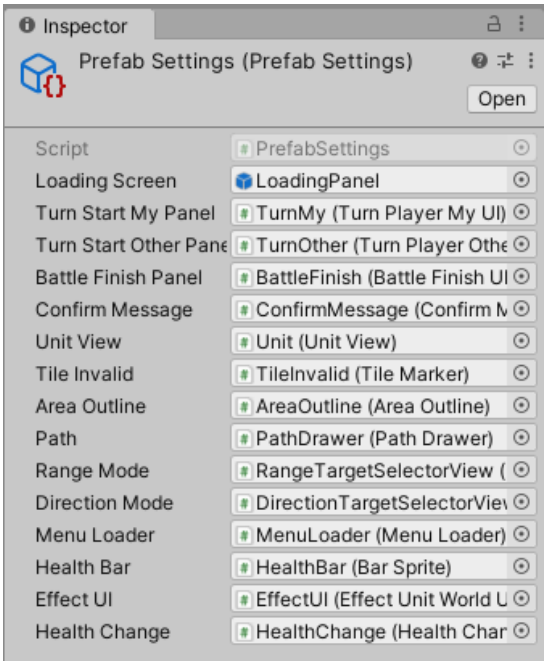
InputSettings

Contains data about keycodes which is paired to corresponding input actions



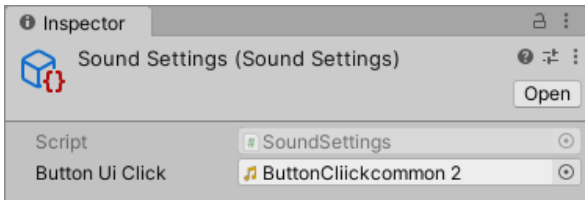
PrefabSettings

Contains reference to prefabs which do not have their own reference holders.



SoundSettings

Contains reference to used audio clips



BattleSettings

Contains information that helps during Battle:

- Could be game autosaved at the end of each turn
- Unit's default interactable material
- Fill Speed of health bar
- Default EffectHandlers which are played when effect will be added (removed)
- Tags mapping

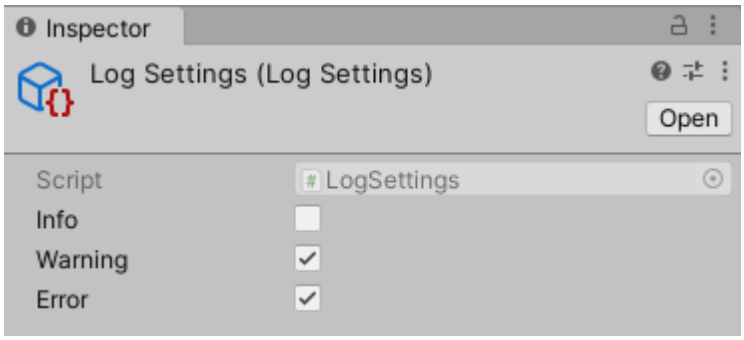
TextSettings

TextSettings is a convenient place to gather texts which could be later used for localization purposes.

1. Status text is displayed at the label at the top of the screen next to the turn counter.
2. Turn Start text is displayed at the start turn label.
3. Turn Finish text is displayed at the confirmation window when player press finish turn button
4. Battle Finish text is displayed at the confirmation window when the battle is over and the player is prompted to exit the menu

LogSettings

Contains Log level toggles

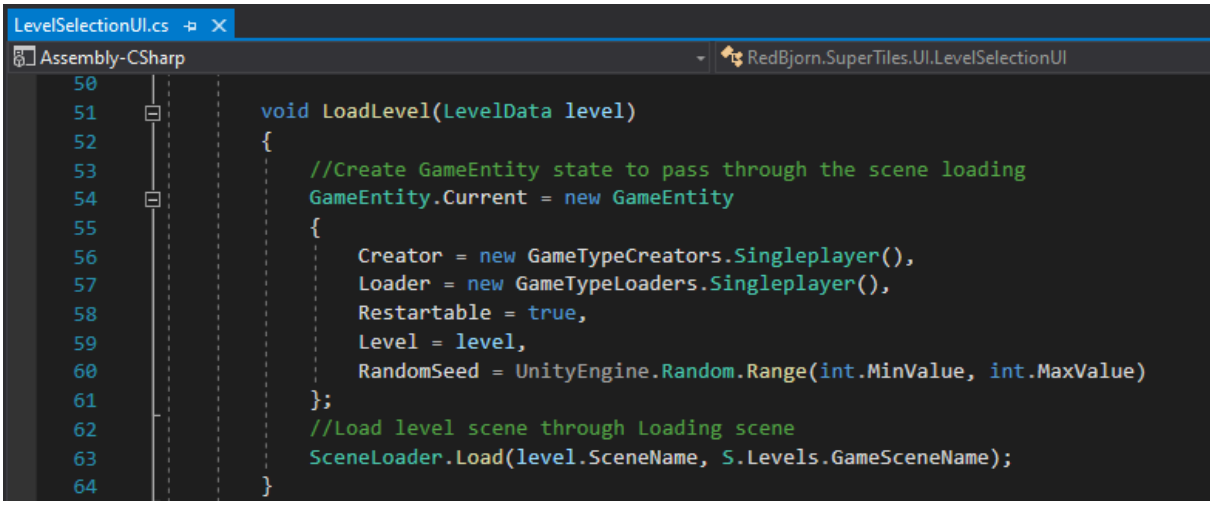


Scenes

Menu

It is a scene which was created only for one purpose: to select *LevelData* which will be loaded.

The core part is located inside `LoadLevel` method



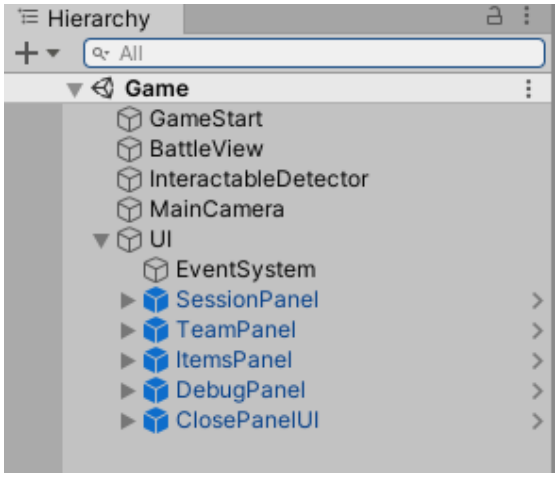
First, we create a new *GameEntity* instance with specified *LevelData*. Then the instance is assigned to a static variable to “survive” through the scene loading process.

Second, we load selected level with additional Game scene through the Loading scene.

Summarizing the above, Menu scene could be customized in any way. The only thing that needs to be left is only these several lines of code.

Game

Game scene includes a bunch of gameObjects which are needed for correct Battle playing.

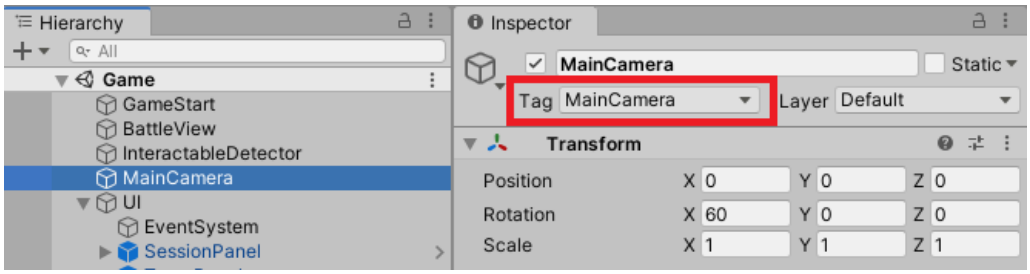


GameStart triggers scene loading and start essential class creating (GameEntity, BattleEntity, UnitEntity, UnitAiEntity and others)

BattleView handles **Player** or **Ai** input and change Battle state respectively

InteractableDetector check If there is any interactable gameObject under the mouse

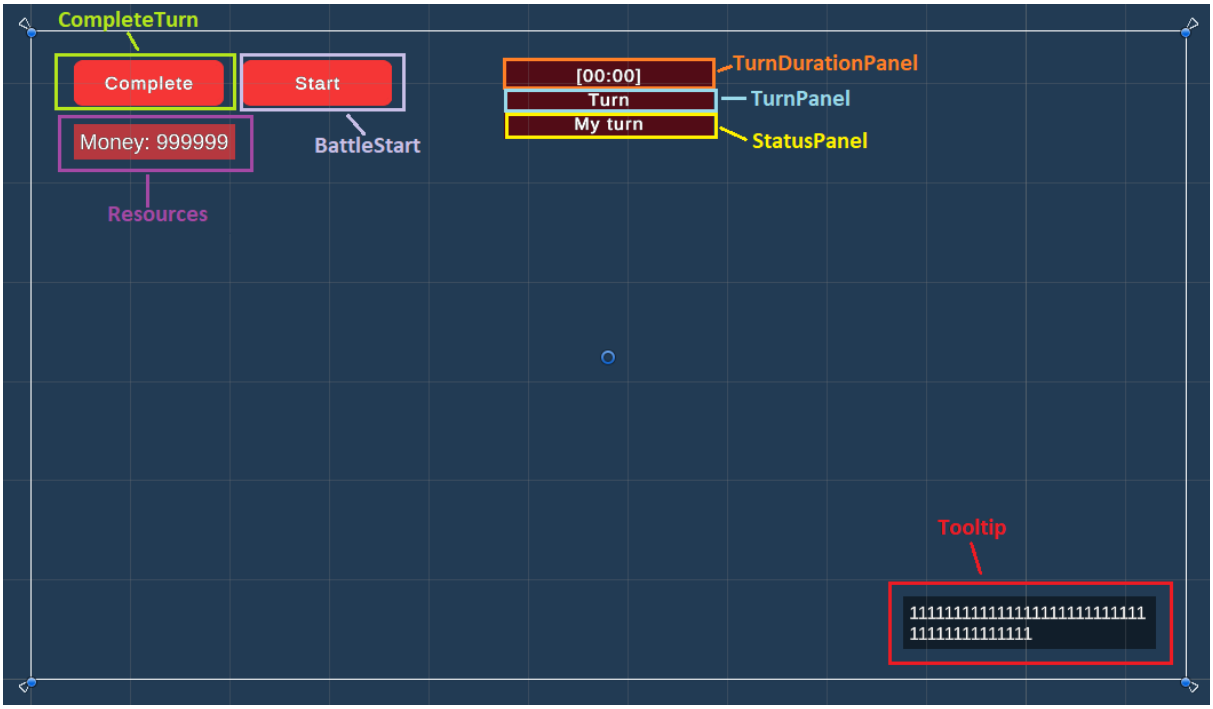
MainCamera is nothing than camera which renders the game. However, it has Unity standard tag Main Camera



Ui is a gameObject with only a CanvasGroup component which is also a parent for Ui related gameObjects

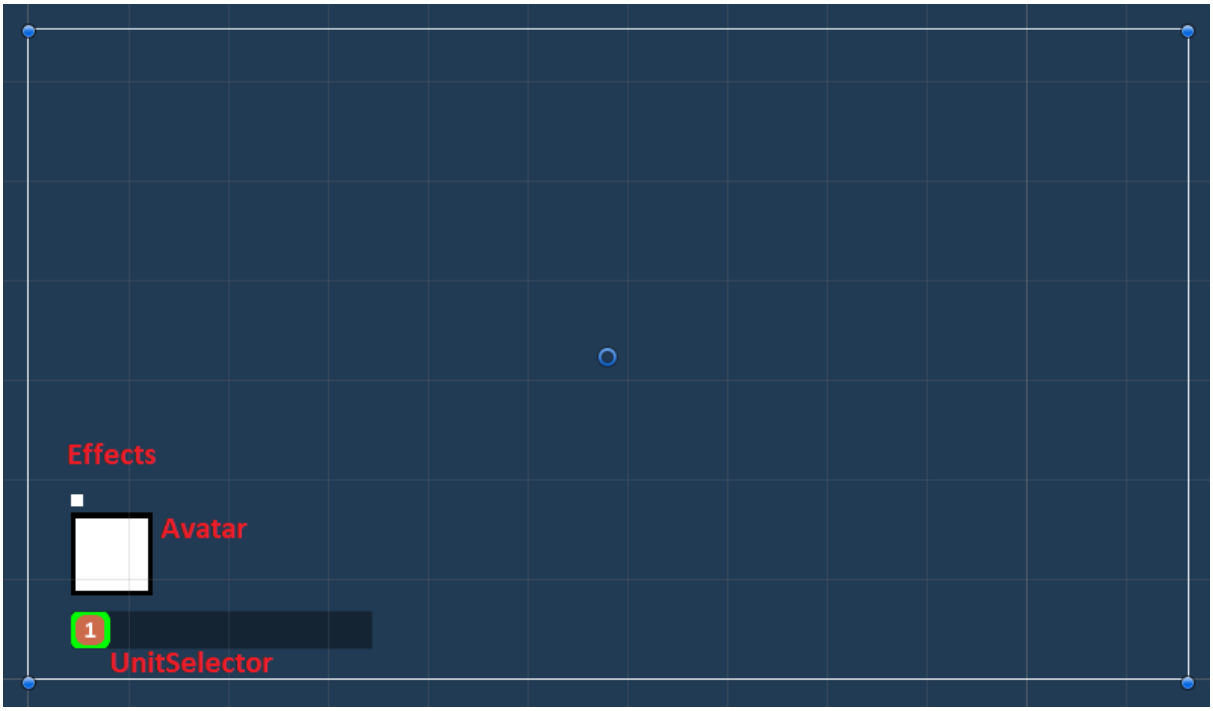
EventSystem – gameObject which is needed for correct input handling in Unity input default pipeline

Session panel represents complete turn button and start Battle button (when Auto Start logic is disabled) and two information texts:

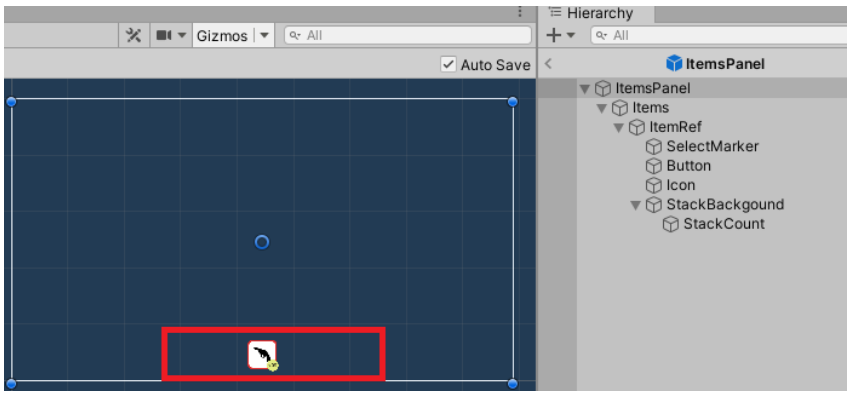


1. **CompleteTurn** - the button to finish turn
2. **BattleStart** -this button is appeared when AutoStart toggle is disabled
3. **Resources** shows the amount of resources of the selected player
4. **TurnDurationPanel** shows the amount of time remaining until the end of turn
5. **TurnPanel** - number of the current turn
6. **StatusPanel** - the status of the current game stage
7. **Tooltip** shows info about unit and item when mouse is pointing such objects

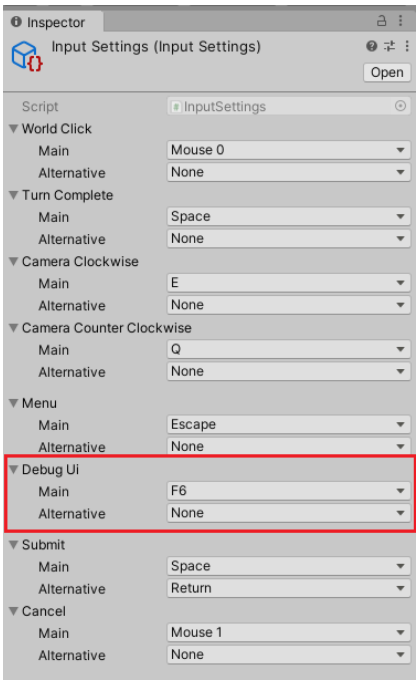
Team Panel shows unit button numbers and selected unit avatar



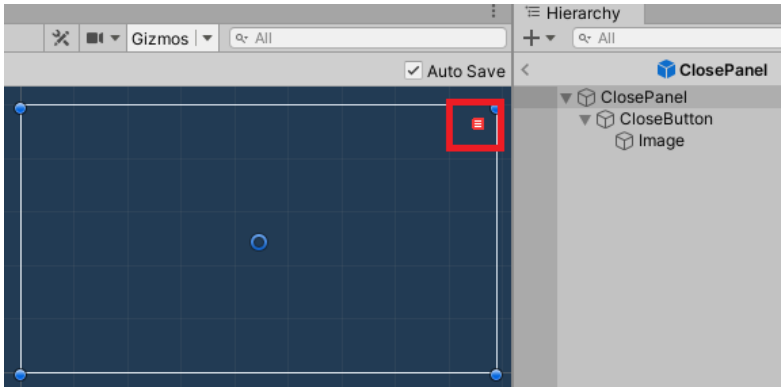
Item panel shows all items for selected unit.



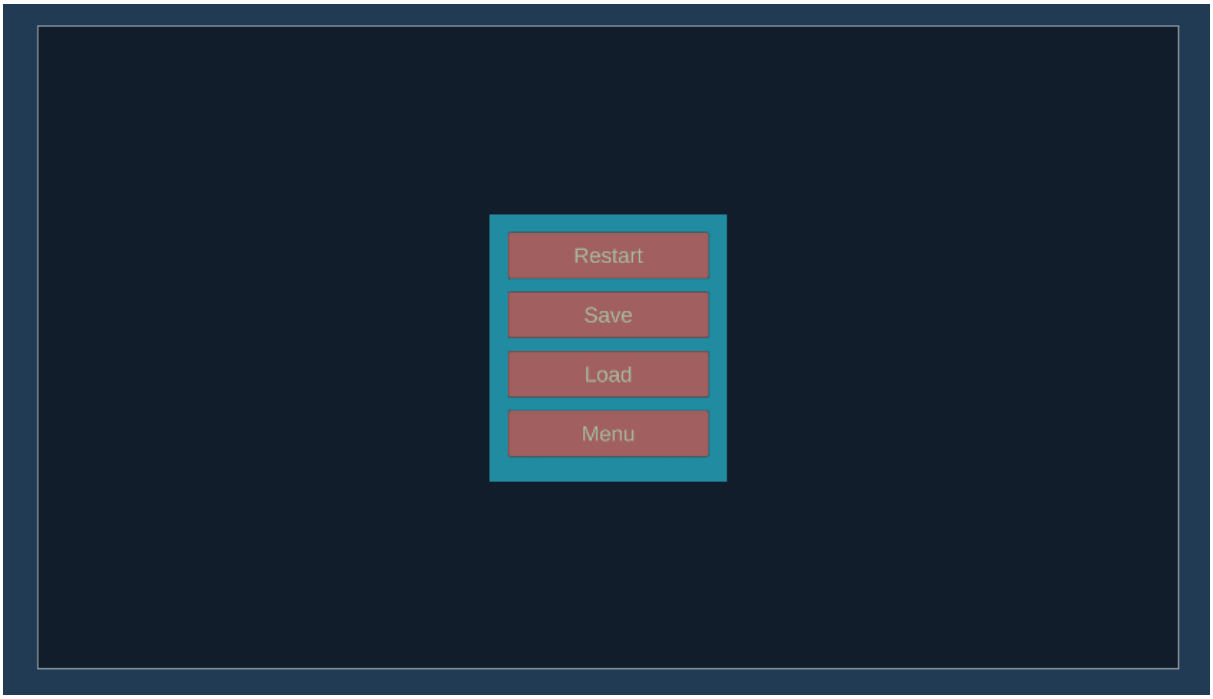
DebugPanel shows debug information about current Battle. It is closed by default and can be opened by clicking Debug UI keycode



ClosePanel represents button which open MenuPanel



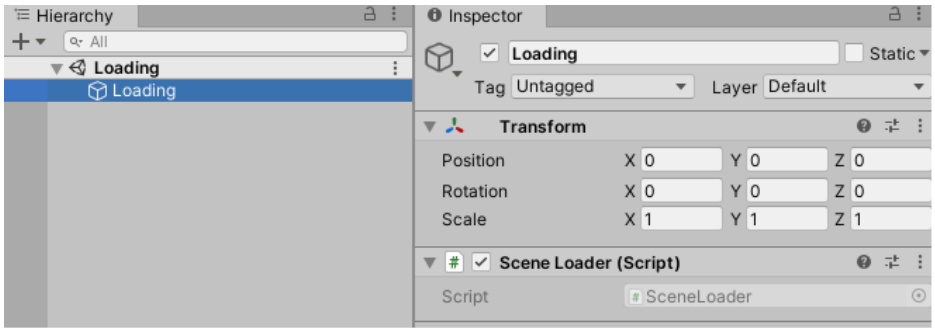
MenuPanel represents in-game menu. Black semitransparent background is a button. Click on this button closes MenuPanel.



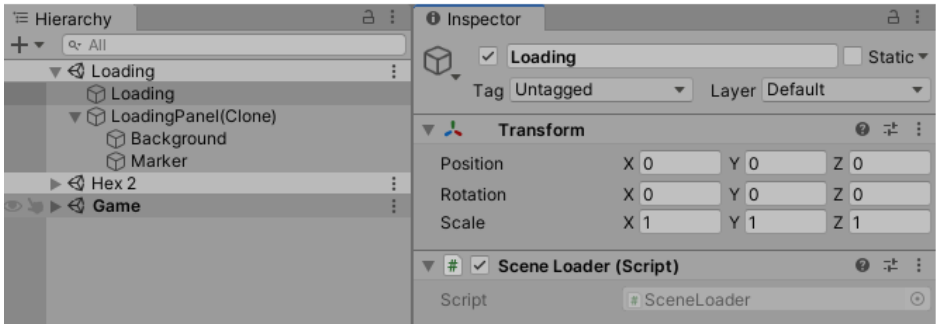
- Restart button restarts current level from the beginning
- Save button saves current session to a file
- Load button loads game state from a file
- Menu button moves the game to Menu scene

Loading

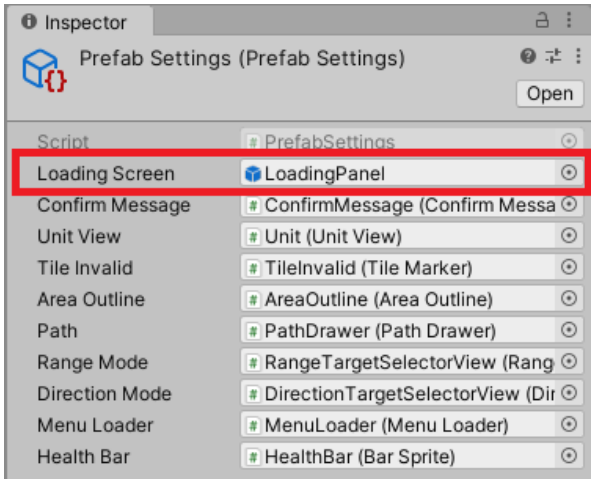
Loading scene is a transitional scene between menu and game scenes. At runtime it instantiates LoadingPanel with black background and white rotating circle



Editor



Playmode



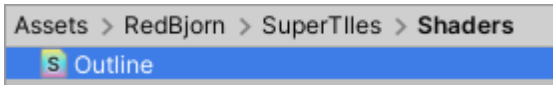
The main logic is encapsulated inside SceneLoader class and handles the order of scene loading.

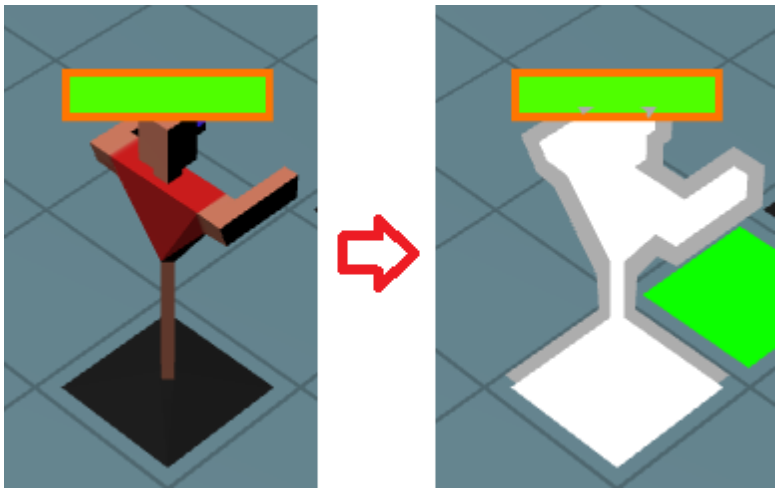
Shaders

There is only one custom shader in SuperTiles: Outline.

Outline

It represents straightforward implementation of outline logic; however, it does its work as expected.





Interactable material is a sole use of current shader.