Tree of Dreams

by

Table of Content:

1. General Information

Engine: Unity 2019.1.14f1
Scripting Language: C#
Version Control: TortoiseHG

Platform: PC
Controls: Mouse & Keyboard

2. Folder Structure

Drive and Project-Share

Drive and Project-Share are Constructed in the exact same way but the Drive is the focused storage.
This is a sample Structure:

```
.
├── Art
│   ├── Assets-2D
│   │   ├── Current
│   │   └── Reference
│   ├── Assets-3D
│   │   ├── Current
│   │   └── Reference
│   ├── Concepts
│   ├── Documents
│   ├── References
│   ├── UI
│   │   ├── Current
│   │   └── Reference
│   └── VFX
│       ├── Current
│       └── Reference
├── Design
│   ├── Balancing
│   ├── Documents
│   └── Scribbles-Wireframes
├── Engineering
│   ├── Builds
│   └── Documents
├── Milestones
│   ├── MS00
│   ├── MS01
│   └── MS03
├── Production
│   ├── Documents
│   └── Marketing
```

└── Sound

Rules:

The Folders ***Art, Design, Engineering, Milestones, Production, Sound*** should not be changed.
No Person-Specific Folders
No double-naming (No second UI Folder, etc….)
There is an ***Current*** folder in some ***Art***-subfolders; Place every „ready for usage“ Asset for the equivalent area int it, so it can be found easier.

Unity Project

The Folder Structure in the Unity-Project should be a mixture of theme based and object oriented Folders. That means no explicit ~~*Scripts*~~, ~~*Sprites*~~ *or* ~~*Materials*~~ Folders.
This is how the Project Structure could look like:

.
├── Environment
│ ├── Buildings
│ └── Nature
├── ~~GUI~~
├── Inventory
├── Menu
├── **Other**
└── Player

This Structure isn‘t fix and can vary. If it isn‘t clear where to put an asset place it in the ***Other*** Folder and inform your local engineer.

3. Naming Convention

Every File or hierarchy element in the Project is named in UpperCamelCase.
Variations are separated by an underscore (_) (Max. one underscore per file!)
Examples:

MainCamera
PlantBase.cs
RecipeScriptableObject
PlayerHouse_02.fbx

Don‘t:

player_movement.cs
greatWall
Wooden_Chest_04_final_XD.fbx

If it is needed to add a Working-state tag to the File place it after the underscore:

PlayerHouse_Current02.fbx

This tag is irrelevant for the unity project so it should be removed when implementing it to the project or placing it in the equivalent ***Current*** Folder.

! ! !
Replace Files in the ***Current*** Folder and the project, do not just add a new one
! ! !
So if there is a *icon.png* Sprite in the project do not just place in there and name it i*con2.png* but override *icon.png*.

4. Repository

General Workflow

The ideal workflow for this project is to keep the default branch up-tp-date and functional. To guarantee that features are created and edited in an extra branch. This extra branch should be merged back into default branch as soon as possible to avoid merge conflicts and errors. If there is a new default commit while on a feature branch the default commit at first gets merged to the this branch to prevent merge errors to be pushed to the default branch. If there are no errors or they all have been solved the feature branch can be merged back to default.
Communication is key! If a commit gets transacted it should be communicated to the team clearly (especially in case of default commits).

Commits

Commits should be done as often as possible. An Ideal commit should be a complete working step. Commits that just save „work in progress content“ should be avoided.
The commit message is a short description of what is changed in this commit. A Commit message consists of one sentence long summary of what was done in this commit and multiple additional enumerations starting with a „-“ followed by a keyword like these: ***added, fixed, changed, removed, ...***
Blocks of text should not be used.

Branches

The default branch contains the latest working state of the game. If there are just minor changes or new assets they should be committed and pushed directly on the default-branch. When working on a new Feature, a new branch for this Feature should be created. The branch should be named after the Feature in UpperCamleCase:

PlayerMovement

On default only the person who creates a branch should work on it. If another person wants to work on it these two must communicate which each other!

5. Coding Convention

In terms of coding this Project follows the offiziel c# coding- :
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions
and naming convention:
https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/general-naming-conventions

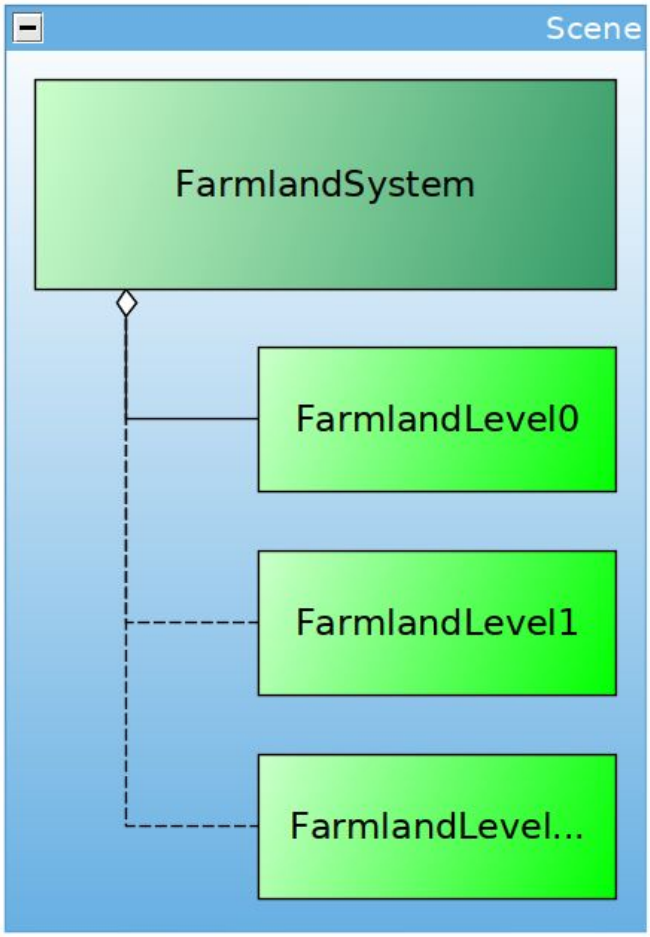
6. Project Structure

This Section describes how the how the scenes and prefabs are structured and work.
Diagrams are used a lot in this Section therefore some conventions should be explained:
[insert Legend here]

Farmland System

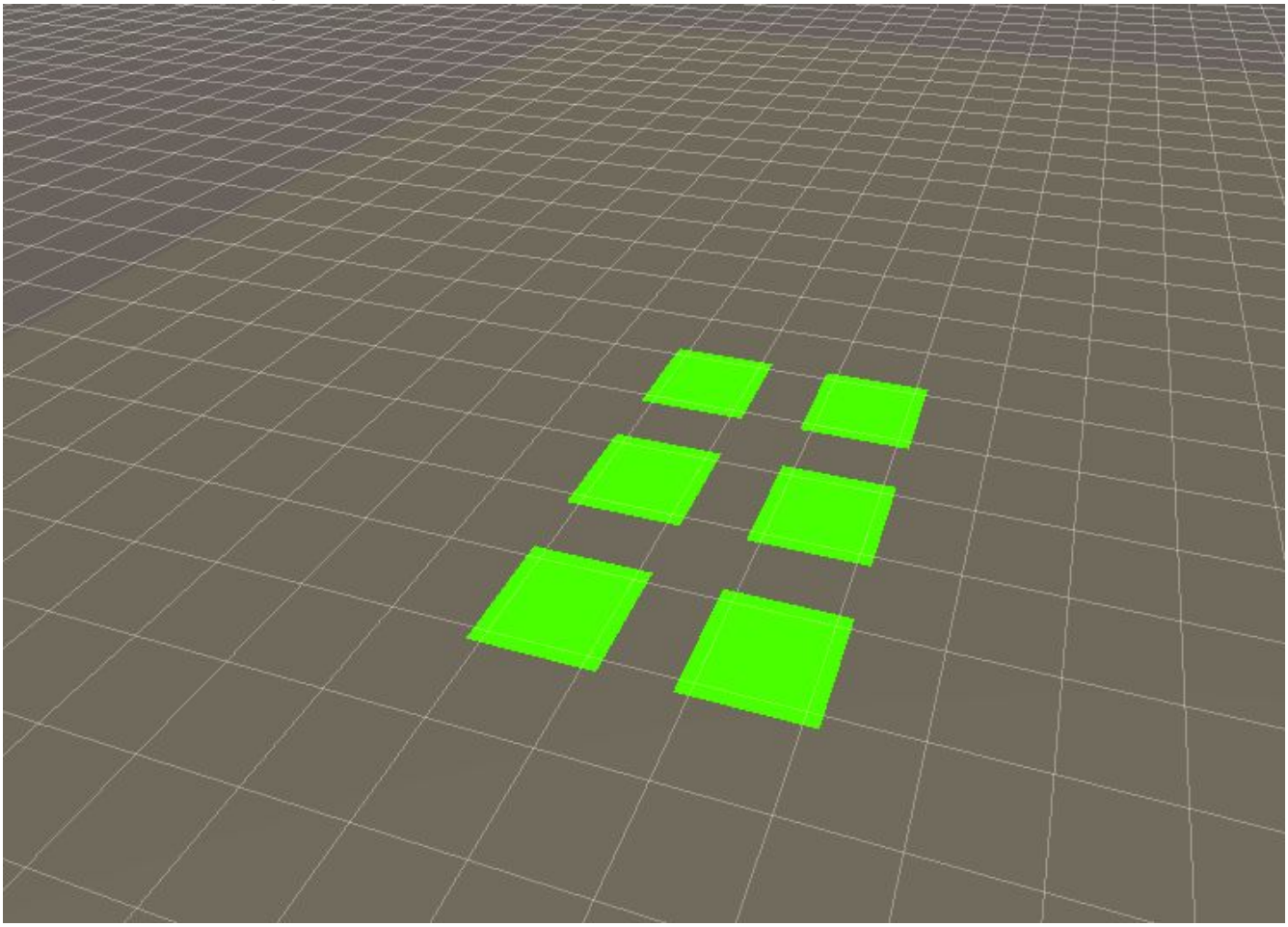
Note: For more information how to work with the Farmland system see Farmland system Workflow Document

The Farmland system is responsible for everything that interacts with or takes place in the 2D grid logic of the scene.

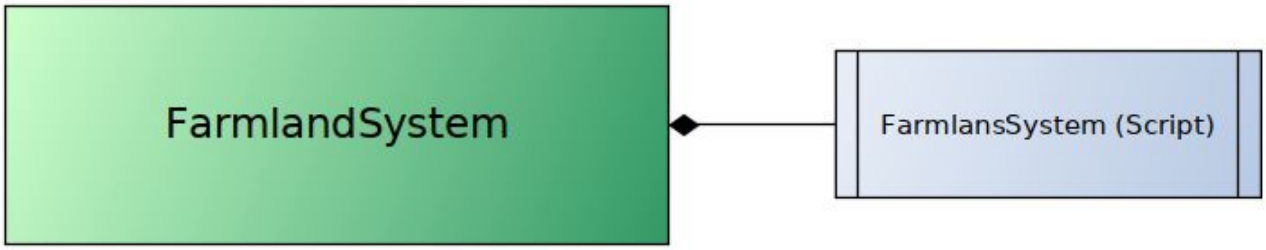


The Farmland system is placed in the scene.

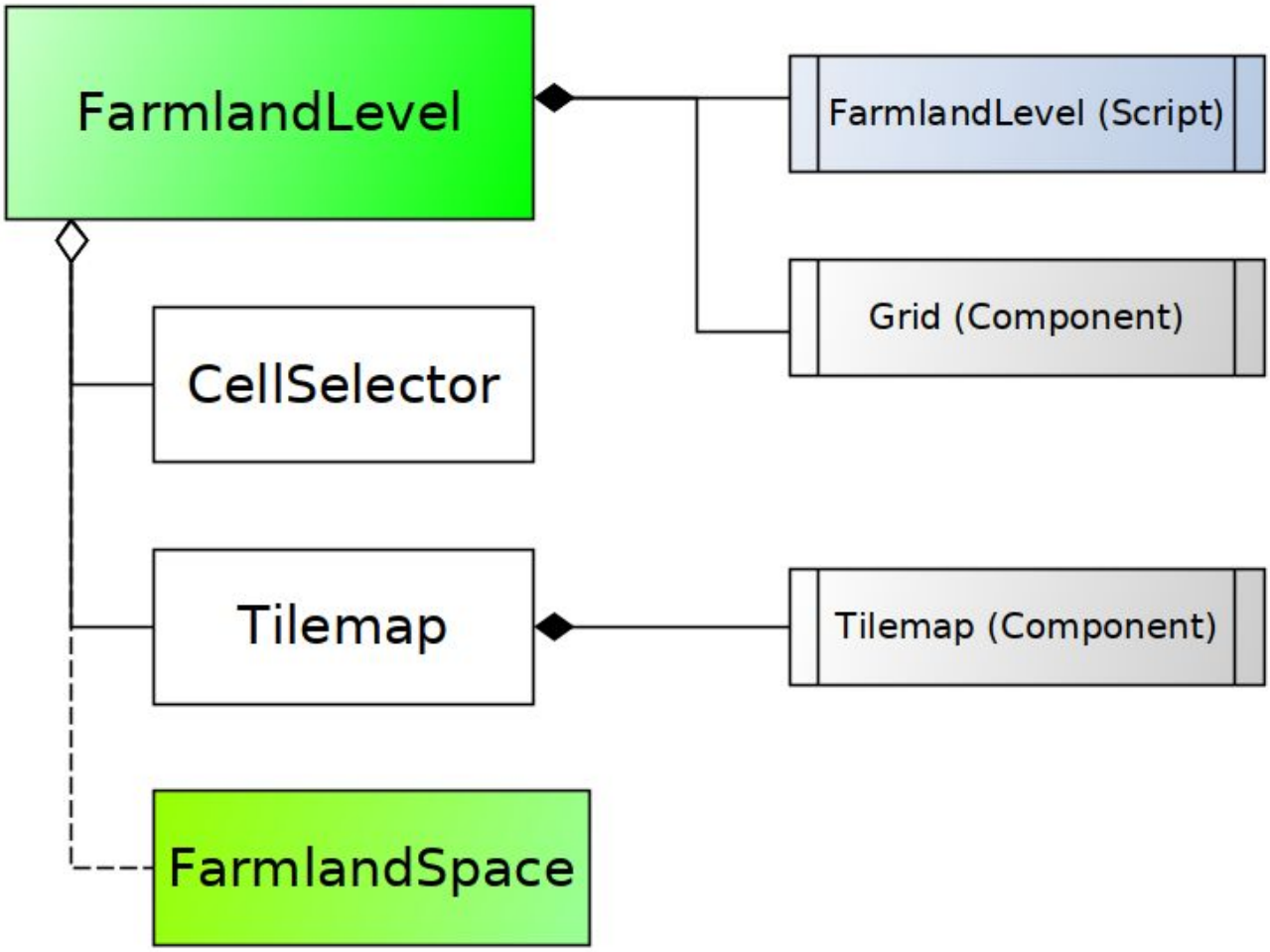
It can have multiple Farmland levels but needs at least one to work. Each Level represents a 2D grid space.



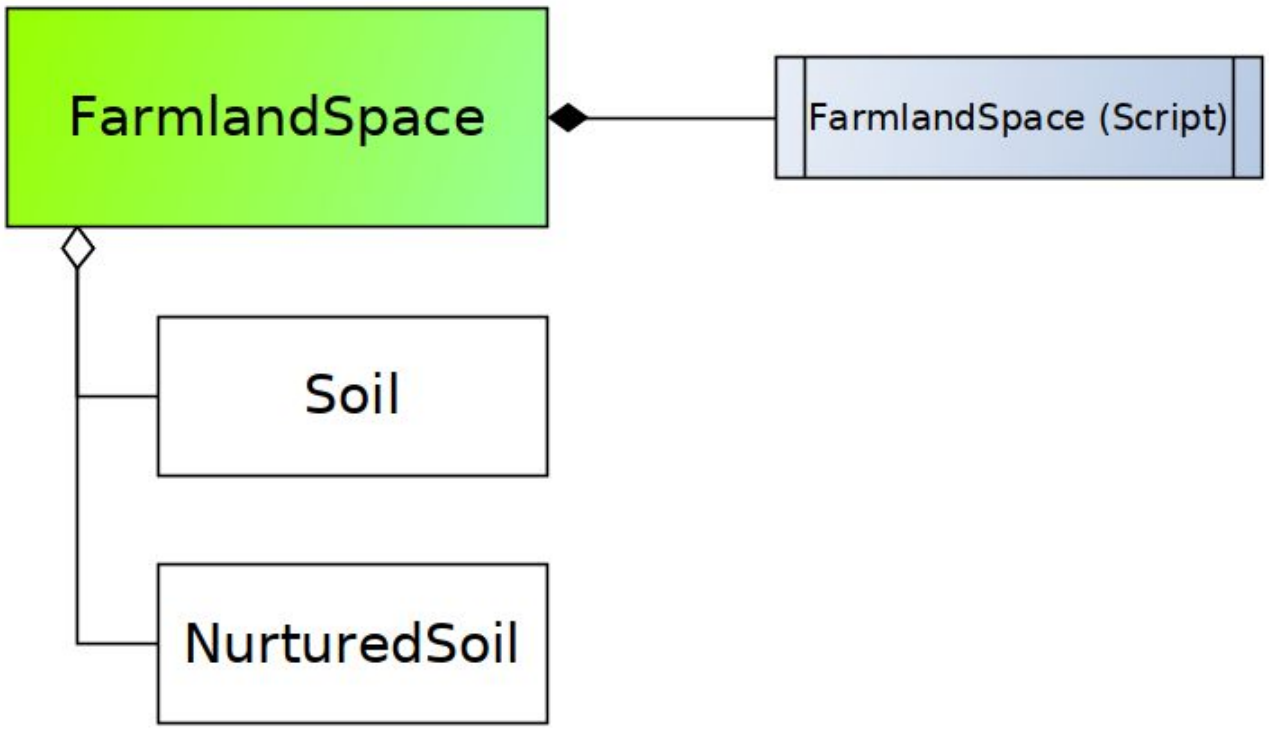
The Player can only interact with certain cells which are set with a tilemap.
When interacting with a level, a Farmland space is created which manages or deletes itself after a day, depending on its state.



The Farmland system prefab has an equevantly named script that manages the these Levels.



The Farmland level prefab has an equivalently named script that manages the interaction with the level and a grid component that is important realize the 2D grid logic in unity.
It also has a Tilemap child object with an equivalently named component that interacts with the grid component and saves which cells are useable for ingame interaction.
The CellSelector is a visualisation for where the player is aiming at on the level. Its position is set on the cell where the mouse points at.

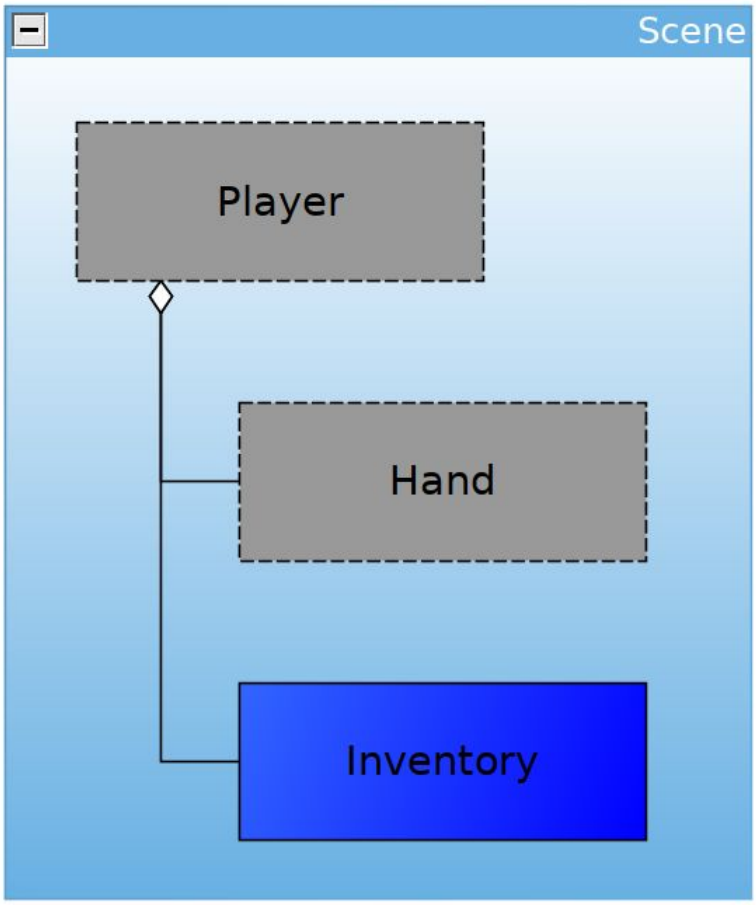


The Farmland space prefab has an equivalently named script that manages its different states.

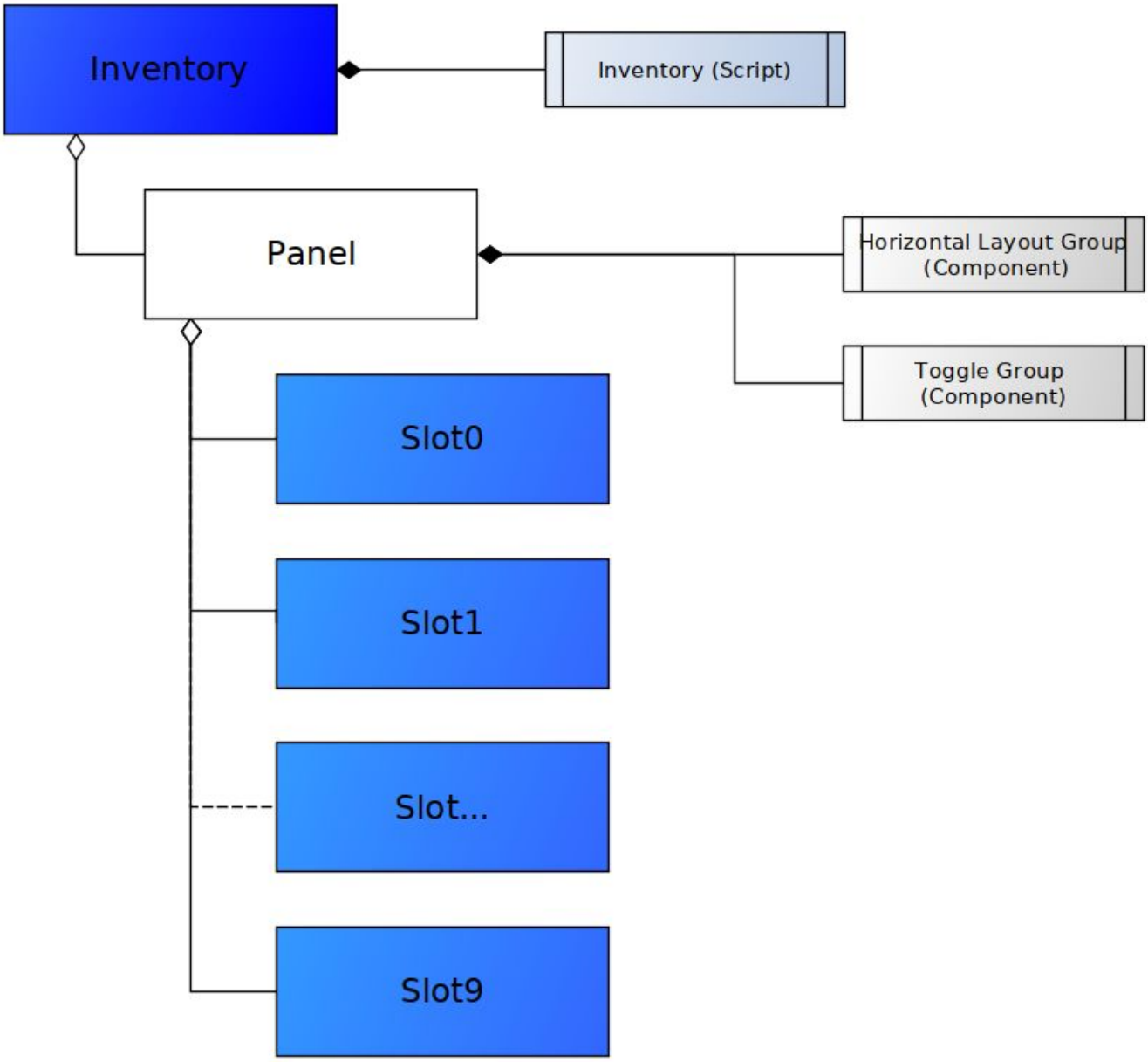
Inventory

Note: For more information on how to work with the Inventory see Inventory Workflow Document

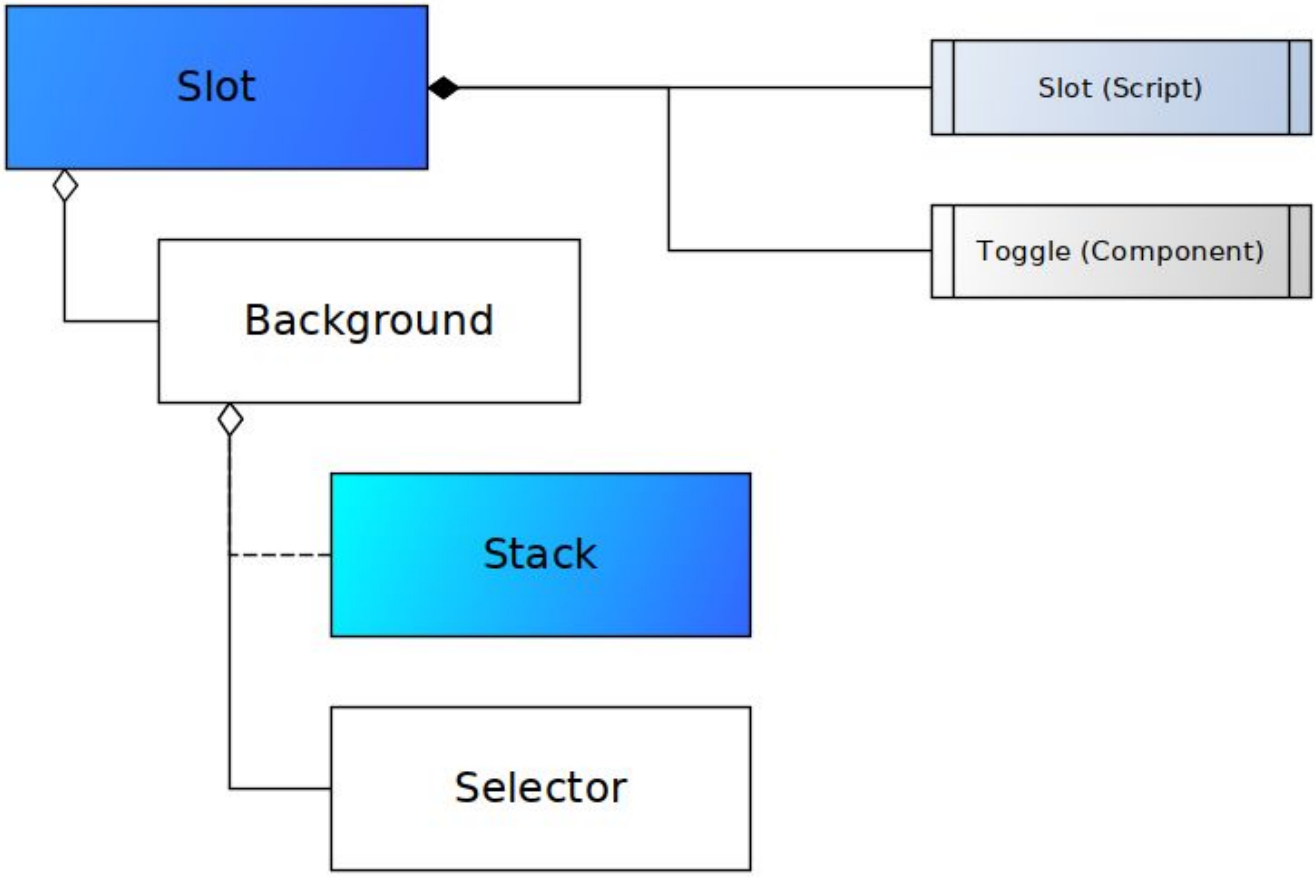
The Inventory is attached to the Player but it can also used on its own.



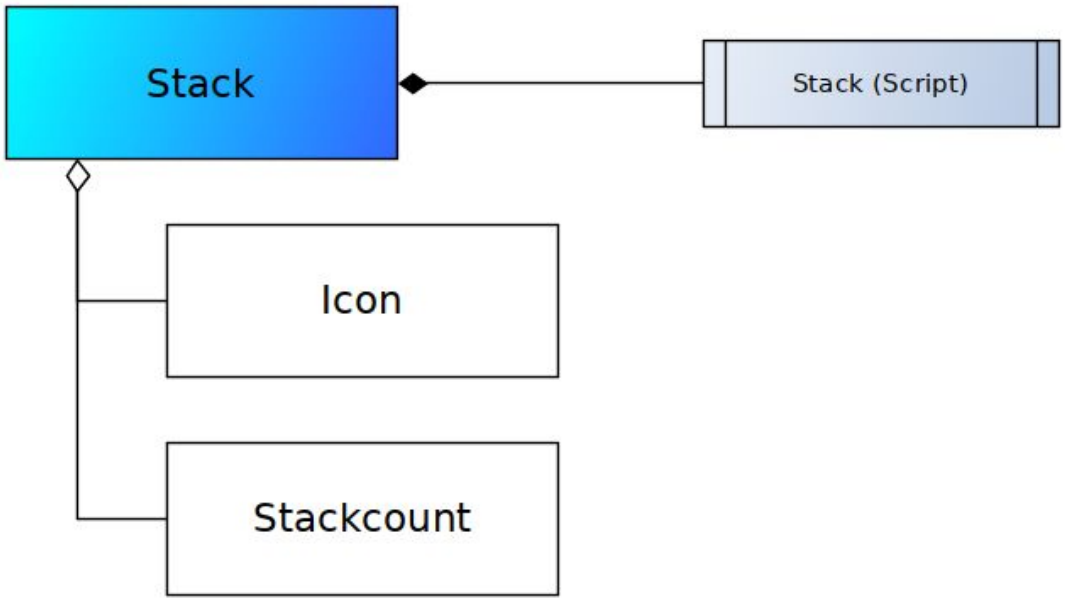
Every Inventory has a equally named Script and multiple Inventory Slots.



Every Slot has an equally named Script and can contain a Stack.



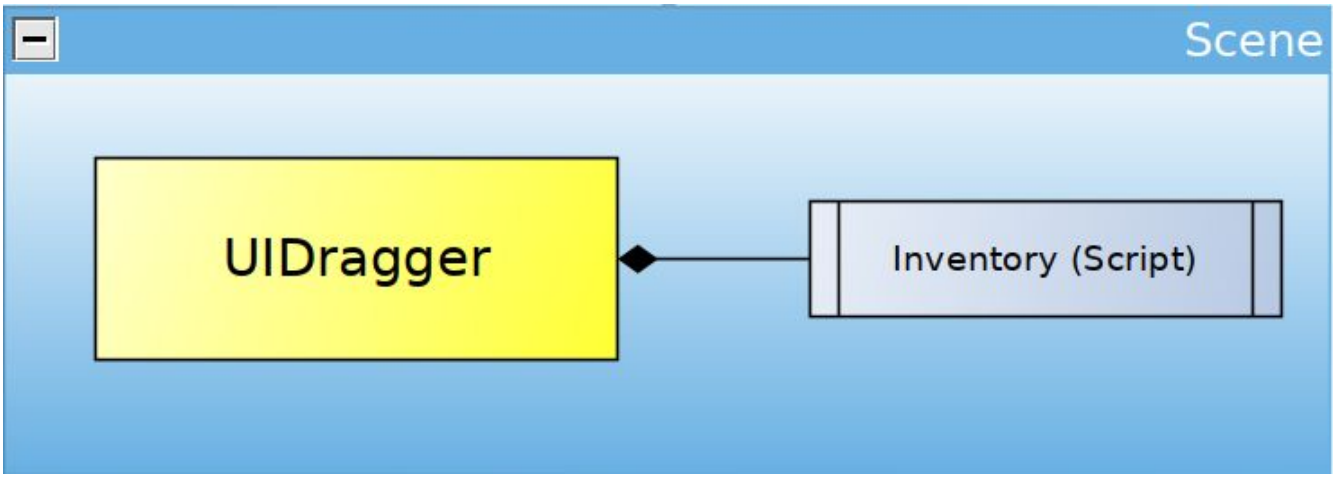
A Stack Contains one or multiple Inventory Items and has a counter to indicate how many items are stored in the Stack.



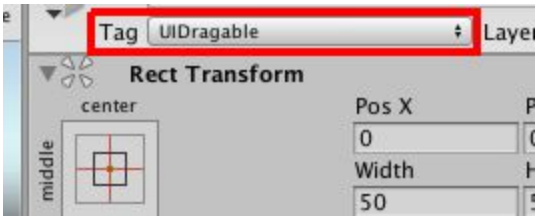
Drag and Drop

Note: There will be a Workflow Document for this Topic in the future

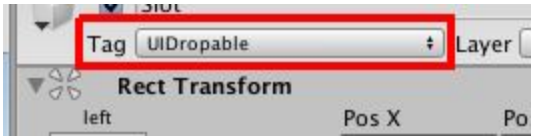
Drag and Drop works over the UIDragger Prefab/Script.



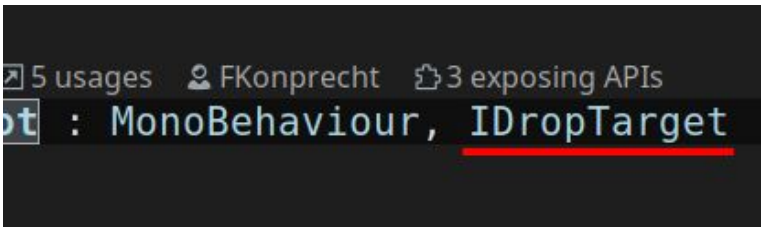
This Script holds a List of Canvas References that determine on which Canvases the Drag and Drop functionality works.
Draggable are only Canvas Items Taged with “UIDraggable”



Droppable are only Canvas Items Taged with “UIDroppable”



How Objects react if a Items gets dropped on them can be controlled with implementing the “IDropTarget” Interface in the local Script.



This Interface is optional though. The dragged Item will be attached to the UIDropable anyway.