





Technical Doc

Created by	 Judds Art
Created time	@September 19, 2025 12:21 AM
Last edited by	 Judds Art
Last updated time	@September 19, 2025 9:53 PM

Rooftop Garden

Author	Date	Version	Comments
Timi Moilanen	3.9.2025	0.0.1	Document created
Timi Moilanen	10.9.2025	0.0.2	Document updated
Ioana Raileanu	10.09.2025	0.1.0	Updated document
Ioana Raileanu	11.09.2025	1.0.0	Finished first draft

Table of contents

[Rooftop Garden](#)

[Table of contents](#)

[Game / Software overview](#)

[Core features](#)

[Technical goals](#)

[Platforms and system requirements](#)

[Tools and third-party libraries](#)

[Coding standards](#)

[Folder structure](#)

[Gitlab base folder structure](#)

[Unreal project folder structure](#)

[each folder can be further sub-divided into categories](#)

[Art assets folder structure](#)

[SW Architecture](#)

[Class diagrams](#)

[Inventory system component](#)

[Farming Plot](#)

[Application flowchart](#)

Game / Software overview

Rooftop Garden is a first-person single player game about farming in a rooftop garden. You are trapped in a compact room, with nothing but your garden to keep you company. Your only friend is the carnivorous plant that mysteriously lives on your rooftop.

You have to grow your own produce to keep yourself alive, as well as avoid the carnivorous plant's wrath.

Core features

The main mechanic of the game is farming, and it consists of the following sub-mechanics:

- Ploughing the soil
- Planting the seeds
- Watering the plants
- Removing bugs and weeds from the plants (if necessary)
- Collecting your produce

The farming mechanic is supported by the day-night cycle feature, as well as by the carnivorous plant, which plays an important role in the farming process: it will sometimes give you random seeds to expand your garden.

Another mechanic branching from the farming is the hunger meter: yours, and the carnivorous plant's. You have to grow enough food to maintain both of you.

Technical goals

- Reliability

- The game should not have any existing crashes/errors by delivery date
- Error handling is consistently done from the start
- Maintainability
 - It should be really easy to build on top of the current framework
 - Component/Interfaces are preferred over casting
- Scalability
 - It's easy to add new features on top of the pre-existing systems
- Data-driven design
 - The artists/designers can easily configure new game content through simple blueprint/data table changes, without having to mess with the code

Platforms and system requirements

Platforms: Windows 10/11 (64-bit)

System requirements:

- **Minimum:**
 - **OS:** Windows 10 64-bit
 - **Processor:** Dual-core CPU (Intel i3-8100 / AMD Ryzen 3 1200 or equivalent)
 - **Memory:** 8 GB RAM
 - **Graphics:** NVIDIA GTX 1050 Ti / AMD RX 560 (4 GB VRAM)
 - **Storage:** ~5-10 GB available
 - **DirectX:** Version 12
- **Recommended:**
 - **OS:** Windows 11 64-bit
 - **Processor:** Quad-core CPU (Intel i5-9600K / AMD Ryzen 5 3600 or equivalent)

- **Memory:** 16 GB RAM
- **Graphics:** NVIDIA GTX 1660 Super / AMD RX 6600 (6 GB+ VRAM)
- **Storage:** SSD with ~10 GB free space
- **DirectX:** Version 12

Tools and third-party libraries

- Operating System: *Windows 10/11*
- Game Engine: *Unreal Engine 5.4.4*
- Programming language: *C++ & Blueprints*
- Source Control: *GitLab (hosted by Kamk/Kamit)*
- Modeling, Rigging, Animation: *Blender 4.4.1, Maya*
 - Blender addon: *Super Batch Export*
- Development Environment: *Visual Studio 2022, JetBrains Rider*
- Textures: *Substance Painter, Photoshop, Blender 4.4.1*

Coding standards

For a more comprehensive list of the coding standards used in this project, please check [Unreal's Coding Standard](#). The more important coding standards are listed below.

- **Raw pointers should almost never be used!** Unreal's type `TObjectPtr` is preferred (e.g. `TObjectPtr<FStaticMesh>`)
- **All public variables need to be marked as UPROPERTY!** Otherwise the garbage collector will create issues.
- **Globals should NOT be used!** Use the game instance or `UworldSubsystem` for something similar.
- It's not always necessary to check if a variable is valid! If the variable is supposed to always be valid, printing an error message when it's not is expected instead of just performing when the variable is valid.

- Functions that return a boolean should ask a question (e.g. `IsCollectable`)
- Comments over non-explanatory public functions should always be included, further explaining the function's utility (and parameters)

3.1. Naming Convention

Generally, the base naming convention C++ for Unreal Engine is PascalCase. There shouldn't be any underlines in the names. Unreal's types have an additional upper-letter prefix (e.g. `FObject`):

- Template classes are prefixed by `T`
- Classes that inherit from `UObject` are prefixed by `U`
- Classes that inherit from `AActor` are prefixed by `A`
- Classes that inherit from `SWidget` are prefixed by `S`
- Classes that are abstract interfaces are prefixed by `I`
- Enums are prefixed by `E`
- Boolean variables must be prefixed by `b`
- Most other classes are prefixed by `F`

For asset naming conventions refer to the [Unreal Directive](#).

For asset naming convention inside the `ArtAsset` folder use the following main rules:


- Texture prefix is `T_`, followed by the mesh name and the map type as a suffix "`T_[AssetName]_[MapType]`". Use the following as suffixes:
 - `_BaseColor`
 - `_Normal`
 - `_Opacity`
 - `_Roughness`
 - `_AO`
 - `_Metallic`


- SM_MeshName for static meshes
- SKM_MeshName for skeletal (rigged) meshes
- AS_AnimationName for animations


Folder structure


Gitlab base folder structure


ArtAssets

 **Source** - contains all source art assets used in the game

 **Exports** - contains all the exported assets (e.g. .fbx) in the game. Updating an exported asset is done by rewriting the original file, so it can be easily reimported in Unreal's editor by *RMB->Reimport*

 **Dev** - contains one Unreal project: the main development folder used in the production phase

 **Main** - contains a stable version of the game - receives regular updates from the Dev folder

 **Prototype** - contains tests and experiments; will be used mostly during the prototype phase

Unreal project folder structure

each folder can be further sub-divided into categories

Animations

 **AnimBlueprints**

 **AnimMontages**

 **AnimNotifies**

 **AnimSequences**

 **BlendSpaces**

Audio

 **SoundCues**

 **SoundWaves**


Blueprints

-  Pawns

-  Actors

-  Components

-  DataTypes

-  Game -> for GameInstance, GameMode, HUD, etc.

-  Particles

-  Widgets

-  Misc

Fonts

Input

-  Actions


Maps

-  Debug

Materials

-  MaterialInstances

Meshes

-  PhysicsAssets

-  SkeletalMeshes

-  Skeleton

-  StaticMeshes

Textures

Art assets folder structure

The art folder should mainly resemble the Unreal project's structure, with the addition of being further subdivided into categories.

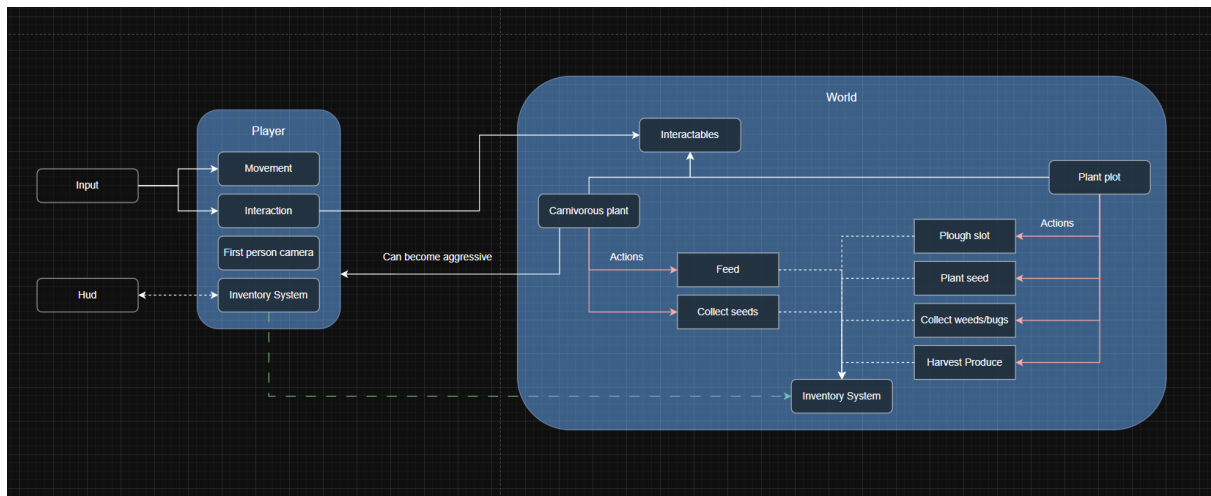
-  Concept art

-  Prototype

-  Meshes

- 📁 **Plants**
 - 📁 **PlantName** – includes all growth stages
- 📁 **Environment**
- 📁 **Animals**
- 📁 **Tools**
- 📁 **Textures**
 - 📁 **Plants**
 - 📁 **PlantName**
 - 📁 **Environment**
 - 📁 **Animals**
 - 📁 **Tools**
 - 📁 **UI**

SW Architecture



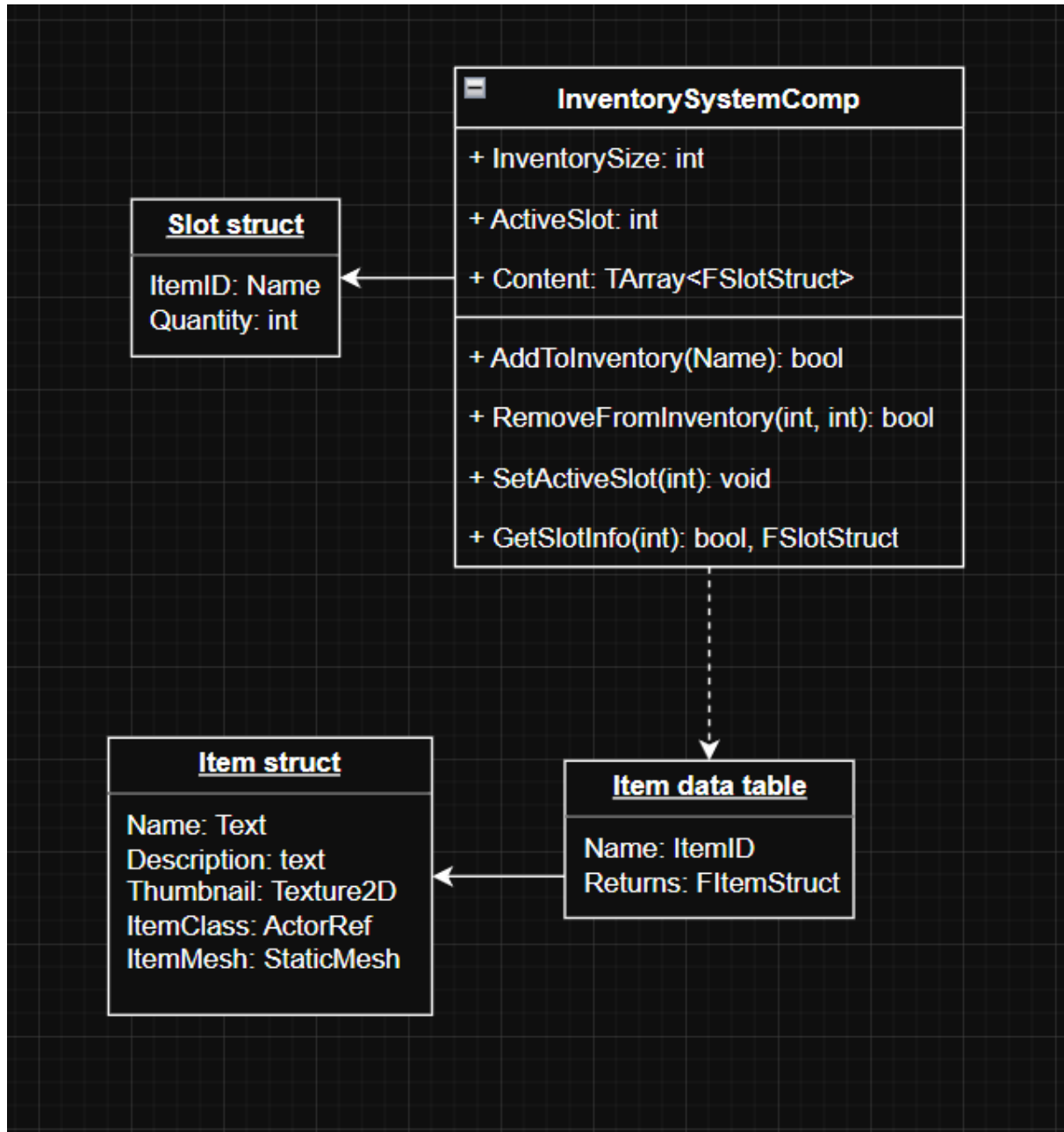
The one system that connects to all interactables is the inventory system, since it involves either using something in your inventory, or collecting something in your inventory.

Input is received in the Player Controller class, which sends an event towards the player that performs the required action (movement, interaction).

All interactable objects use the Interactable interface.

Class diagrams

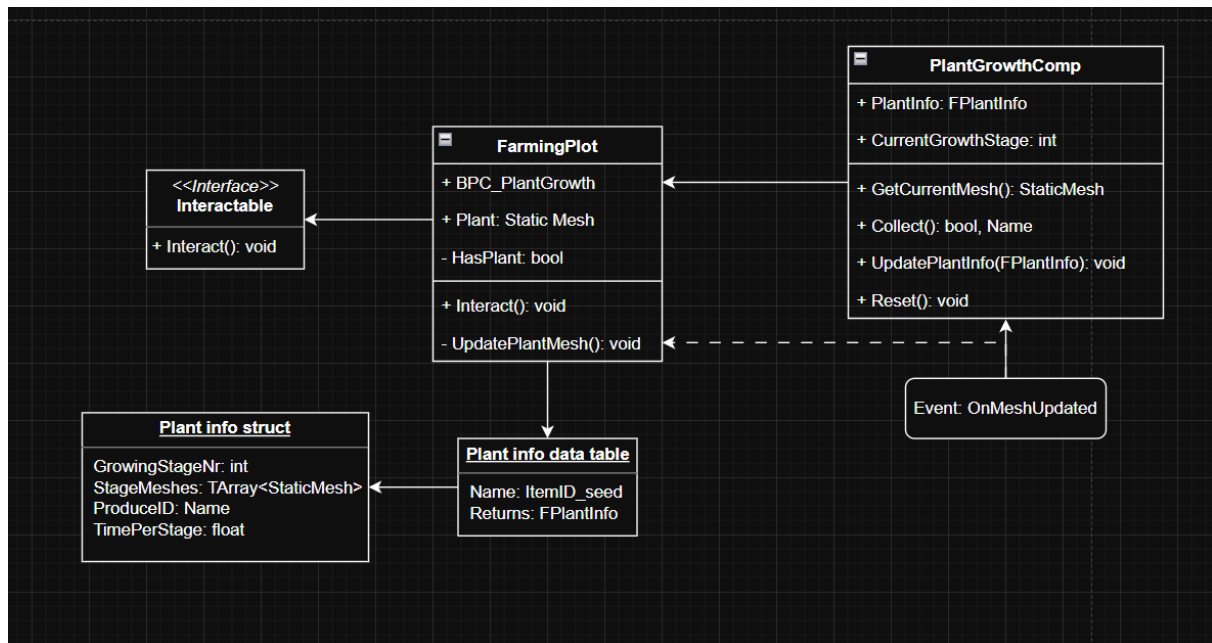
Inventory system component



The inventory system component is contained in the Player class, and is referenced by most, if not all, of the interactables.

The pick-up objects each has an item ID which they can be identified by, and found in the Item Data Table.

Farming Plot



The farming plot is an essential actor for the farming mechanic. It manages plant growth, planting seeds, collecting produce, and will soon also include ploughing the soil, weeding and collecting the bugs.

Application flowchart

