Astroneer Modding Documentation

Release 1.0

Astro Techies

Jul 07, 2023

CONTENTS

This documentation site includes the formal standards that mods for the game Astroneer should adhere to, but also practical guides for actually creating mods.

If you need help feel free to check out the Astroneer Modding Discord.

Formal standards can be found in the *Modding standards* section which includes the *Metadata standard* and the *Index File Standard*.

If you want to learn how to actually make mods checkout the *Making Mods* section.

Note: This site is not affiliated with System Era in any way and is exclusively community-run. Also Astroneer has no official mod support and everything is community-built.

Contents:

CHAPTER

ONE

MODDING STANDARDS

Contents

Note: This site is not affiliated with System Era in any way and is exclusively community-run. Also Astroneer has no official mod support and everything is community-built.

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.

1.1 General Information

All mods are Unreal Engine 4 .pak files. The .pak file format is used by the Unreal Engine for storing large amounts of data in a compact space. When placed into the `%localappdata%\Astro\Saved\Paks` directory, these .pak files are loaded by Astroneer as patches, or partial replacements, of the main game assets found in the `Astro-WindowsNoEditor.pak` file.

1.1.1 .pak Mod File Names

On distribution, .pak mod file names MUST follow the following format, as to facilitate the usage of mods even without a mod loader:

`{PRIORITY}-{MOD ID}-{VERSION}_P.pak`

- `{PRIORITY}` represents a 3-digit number, such as `001` or `005`. Larger numbers are generally loaded later by the engine, so they always have priority when multiple mods override the same file. Most mods will likely want to use a priority of `000` or `001`, but in some cases, higher priorities are called for. The priority levels `9xx` is reserved for external applications, and MUST NOT be used by regular mods.
- `{MOD ID}` is any alphanumeric string representing the ID (and, roughly, the name) of the mod, as a means of distinguishing it from other mods. Mod IDs MUST NOT contain any characters other than uppercase ASCII letters, lowercase ASCII letters, and the digits zero through nine; as such, they also MUST NOT include any special characters, including, but not limited to, hyphens, underscores, and spaces. Mod IDs SHOULD be formatted in upper camel case, also known as Pascal case. A single dot is allowed to extend the Mod ID with an Author ID to distinguish Mods made by different people.

- `{VERSION}` represents the current revision of the mod file. The version MUST be represented by at least two numbers separated by periods, but ultimately SHOULD be `MAJOR.MINOR.PATCH` .
- The `_P` at the end of the file's name is a requirement for the mod to load.

1.1.2 Index File Standard

This file contains data and direct download URLs for one or more mods. It is used to automatically find a mod's download link and to facilitate auto-updating. All mod listings are contained in the `mods` field. Each field contains data for one mod and the field key should be the `mod_id` of the corresponding mod.

As an example, here is a valid index file:

```
{
    "mods": {
        "ExampleMod": {
            "latest_version": "1.1.0",
            "versions": {
                "1.0.0": {
                    "download_url": "https://example.com/upload/123/file.pak",
                    "filename": "000-ExampleMod-1.0.0_P.pak"
                },
                "1.1.0": {
                    "download_url": "https://example.com/upload/124/file.pak",
                    "filename": "000-ExampleMod-1.1.0_P.pak"
                }
            }
        }
    }
}
```

1.1.3 Metadata standard

Note: If you are looking for metadata v1 standard, it has moved here. Please note that metadata v1 standard is outdated, and it's recommended to use metadata v2 standard.

The following describes Schema Version 2.

The metadata of mods is stored in the JSON format, as described in RFC 8259, within a file with the name `metadata.json`. This file MUST be placed at the root directory within all .pak mods, MUST be encoded in UTF-8, and MUST NOT include a byte order mark.

The following is a list of fields that can be specified within the root object of the metadata:

- `schema_version`: An integer that represents the current version of the `metadata.json` standard that is being used. The schema version is incremented every time there is a backwards-incompatible change to the format. This field SHOULD be specified, but if it is left unspecified, it defaults to 1, the initial version of the standard.
- `name`: A plain text display name for the mod. This field is represented as a string, and is REQUIRED.
- `mod_id`: The ID for the mod, which MUST be the exact same as the mod ID specified in the mod's original file name, and follows the same restrictions and recommendations described within the *.pak Mod File Names* section. This field is represented as a string and is REQUIRED.

- `author`: The author of the mod. This field is represented as a string, and is OPTIONAL, defaulting to an empty string.
- `description`: A plain text display description of the mod. This field is represented as a string, and is OPTIONAL, defaulting to an empty string.
- `version`: A version for the mod, which MUST be the exact same as the version found in the mod's original file name, and follows the same restrictions and recommendations described within the *.pak Mod File Names* section. This field is represented as a string, and is REQUIRED.
- `game_build`: The Astroneer build for which the mod was built. This field is represented as a string, and is OPTIONAL. It defaults to `null`, which is generally understood to mean that the mod works regardless of the current Astroneer build.
- `sync`: The sync mode between servers and clients. This field is represented as a string, and is OPTIONAL, defaulting to `"serverclient"`. Valid options are:
  - `none`: Represents a mod which will be ignored while syncing.
  - `server`: Represents a mod which will only be installed server-side.
  - `client`: Represents a mod which will only be installed client-side.
  - `serverclient`: Represents a mod which will be installed both server-side and client-side.
- `homepage`: A link to the homepage of the mod, a web page where users can go to to find more information about the mod or the author of the mod. This field is represented as a string, and is OPTIONAL, defaulting to an empty string.
- `download`: An object with fields defining how to auto-update the mod. This field is represented as an object, and is OPTIONAL, defaulting to `{}`, in which case auto-updating is disabled. These are the valid fields:
  - `type`: The type of download. This field is represented as a string, and is OPTIONAL. Valid options:
    - `"index_file"`: This mod can be downloaded through an "index file" hosted on the web, which contains the version info of any number of mods and their direct download web URLs. See *Index File Standard*.
  - `url`: If the type field is set to `"index_file"`, this is set to the web URL of the mod's index file. This field is represented as a string, and is OPTIONAL.
- `integrator`: A json object containing integration sections to load into the game
  - `persistent_actors`: A standard JSON array of asset paths to actors to bake into the level. This field is represented as an array, and is OPTIONAL, defaulting to `[]`.
  - `mission_trailheads`: A standard JSON array of asset paths to mission trailheads (such as those found in the `/Game/Missions` folder) to bake into the level. This field is represented as an array, and is OPTIONAL, defaulting to `[]`.
  - `linked_actor_components`: A standard JSON object, where the keys are game paths to Actors and the values are standard JSON arrays that provide a list of game paths that the mod integrator will automatically attach to the specified Actors. This field is represented as an object, and is OPTIONAL, defaulting to `{}`.
  - `item_list_entries`: A standard JSON object where the keys are game paths to any asset and the values are standard JSON objects where the keys are array names to modify in the asset and the values are standard JSON arrays which list entries to add to the specified array as object pointers. Alternatively, the array names to modify in the asset can be specified with the format `category name.array name` in order to hone in on one particular array. This field has a niche use, but is especially important in adding entries to commonly used item lists, such as the list of items that a certain printer can print or the global master list that contains items that need to be referenced on bootup for the research catalog or otherwise. This field is represented as an object, and is OPTIONAL, defaulting to `{}`.

  - `biome_placement_modifiers`: Placement modifiers used for adding custom procedurally generated actors.

- `dependencies`: A json object containing dependencies that must be fetched for this mod to work. Dependency version requirements follow the semver standard.

As an example, here is a valid `metadata.json` file incorporating all of the defined fields:

```
{
    "schema_version": 2,
    "name": "Coordinate GUI",
    "mod_id": "CoordinateGUI",
    "author": "ExampleModder123",
    "description": "Adds a coordinate display that toggles with the F3 key.",
    "version": "0.1.0",
    "game_build": "1.19.143.0",
    "sync": "client",
    "homepage": "https://example.com",
    "download": {
        "type": "index_file",
        "url": "https://cdn.example.com/index.json"
    },
    "integrator": {
        "persistent_actors": [
            "/Game/ExampleModder123/ExampleGUI/ExampleGUIActor"
        ],
        "mission_trailheads": [
            "/Game/ExampleModder123/ExampleMod/MissionTrailhead04-Example"
        ],
        "linked_actor_components": {
            "/Game/Character/DesignAstro": [
                "/Game/ExampleModder123/ExampleGUI/MyActorComponent"
            ]
        },
        "item_list_entries": {
            "/Game/InitialUnlocks_Generous": {
                "ItemTypes": [
                        "/Game/Items/ItemTypes/Components/LevelingBlock"
                ]
                },
                "/Game/Items/BackpackRail": {
                    "PrinterComponent.Blueprints": [
                        "/Game/Components_Terrain/LevelingBlock",
                        "/Game/ExampleModder123/ExampleGUI/ExampleItem_BP"
                ]
            }
        }
    },
    "dependencies": {
        "ModA": ">=1.2.0",
        "ModB": "*",
        "ModC": {
            "version": "^1.2.3",
            "download": {
```

(continues on next page)

(continued from previous page)

```
                "type": "index_file",
                "url": "https://example.com"
            }
        }
    }
}
```

As another example, here is a valid `metadata.json` file containing only the `"schema_version"` field and the REQUIRED fields:

```
{
    "schema_version": 2,
    "name": "My Tiny Mod",
    "mod_id": "TinyMod",
    "version": "0.1.0"
}
```

CHAPTER

TWO

MAKING MODS

If you want to start making mods, this is the right place.

First you will learn the basics of how Astroneer mods are structured and how to make basic mods by editing game files. Then you will learn how to make more advanced mods by using Unreal Engine to create your own items.

2.1 About Astroneer Modding

Contents

This page will give you an overview of how Astroneer mods are structured and distributed.

Note: This site is not affiliated with System Era in any way and is exclusively community-run. Also Astroneer has no official mod support and everything is community-built.

2.1.1 About Astroneer

Astroneer uses a slightly modified version of Unreal Engine 4.23.1. Modding Unreal Engine games can be hard, but is not impossible. The modding tools that will be shown later in this guide can simplify the process extensively, depending on the mod you are trying to make.

Doing small tweaks for stuff like power values is quite simple, but adding your own logic or items takes a lot of time.

2.1.2 Unreal Pak Files

All mods (or at least what we are focusing on) are Unreal Engine 4 .pak files. These files are comparable to .zip files. They store a bunch of compressed files inside them. Distributing a mod is as easy as giving somebody else a .pak file.

When put in specific directories (manged by the modloader) they will loaded by the game after the main `Astro-WindowsNoEditor.pak`. Then they can either replace or modify existing parts of the game or even add new files and items to the game.

Mod .pak files also include some extra data in the `metadata.json` file placed at the root of the directory structure inside. This includes information like the mod name, author, where to get updates and some special instruction for the mod integrator, which is responsible for avoiding conflicts between mods.

2.1.3 Unreal Asset Files

All the actual game data like values, textures, models, some code and much more is stored in Unreal asset files. They usually have the .uasset extension. There are two different types of asset files:

- Uncooked files. These files are used during developement and are what are stored in a Unreal project. They can be easily opened and modified in the Unreal Editor.
- Cooked files. These files are included in distribution game builds and have a lot of data stripped. Opening and modifying these files requires special tools because they are usually generated from uncooked ones and not meant to be modified directly. But there are all that we have for modding.

In most games cooked .uasset files also come with a .uexp for the same asset. This extra file stores the bulk of the data for optimization.

Up until the Xenobiology Update (aka. Snail Update) Astroneer did not use .uexp files and bundled all the data in the main .uasset file. This change broke a lot of mods.

2.1.4 File Structure and Paths

The files in .pak files in Unreal Engine games are always structured similarly. Here is how it is in Astroneer.

```
root
├─Astro
│   ├─Config
│   │   └─...
│   ├─Content
│   │   ├─Animations
│   │   │   └─...
│   │   ├─Items
│   │   │   └─...
│   │   └─...
│   └─...
└─Engine
    └─...
```

All files important for the actual game are in the `/Astro/Content` directory. However paths in assets don't actually directly referece that. Instead the they use the `/Game/...` path which gets remapped to `/Astro/Content/...` at runtime. Also Unreal uses forward slashes as path separators.

When making a mod you would place an asset file `/Astro/Content/Mods/ModderName/MyModId/MyAsset.uasset` in you .pak file but referece it as `/Game/Mods/ModderName/MyModId/MyAsset.uasset` inside assets.

2.1.5 Tools For Creating Mods

Typically, mods are created using either UassetGUI or Unreal Engine directly. Each of these tools provides benefits over the other. UassetGUI, which was originally written for Astroneer modding, is a great way to quickly generate a mod that involves small changes to the game, such as changing numbers. An example of this is modifying the max speed of a rover, which is very easy to do with UassetGUI. UassetGUI's main downside is that it's limited in capabilities for more advanced projects, such as adding new items to the game. Your first tutorial for making mods will utilize UassetGUI. Unreal Engine Editor 4.23.1 is also used to create mods, utilizing an environment similar to game development on the platform. While it is a bit harder to learn than UassetGUI, and does not have all of the game files available to it (yet), it allows for much more advanced mods to be created. Unreal Editor also has less chance of causing mod conflicts due to how .uasset files are registered within the modloader. **Using Unreal Engine Editor is recommended for advanced projects.**

2.1.6 Items in Astroneer

One of the most common things you might want to do is tweak or make are items. Items in Astroneer consist of two assets, ItemType and PhysicalItem.

- ItemType is used by Astroneer to get in item's recipe, menu icon, item properties, description, etc. Most of them are found in `/Game/Items/ItemTypes/`. They sometimes have the suffix `_IT`.
- PhysicalItem is the object that is going to be created in the world. Most of them are found in `/Game/Components_*/` and `/Game/Items/`. They sometimes have the suffix `_BP`.

Also keep in mind that the Astroneer game files are very messy and it can be hard to find stuff. Windows search is your biggest help.

2.1.7 Getting Help

If you ever need help or stuck with a mod you are trying to make ask in the Astroneer Modding Discord in the #making-mods-help channel.

To get started go to the *Setting up Modding Tools* section.

2.2 Setting up Modding Tools

Contents

2.2.1 Setting up Folders

Start by creating a folder `AstroneerModding` in your PC. A good location is your documents folder. This folder must be on a drive with at least 10GB of free space. If you also want to make mods with Unreal Engine later you will need another 30GB.

2.2.2 Setting up unreal_pak_cli

When working with .pak files you need some kind of program to extract and create them. We will be using the community-created `unreal_pak_cli` program. The source code can be found here. To download a pre-built binary and some useful .bat files click `HERE`. (We will be using this)

Once you have downloaded the .zip extract it to your `AstroneerModding` folder. Then go into the `unreal_pak_cli`. You should see the .exe and two .bat files. We will be adding the bat files to the Windows Send to window to speed up the modding workflow. This is not strictly neccessary and use the .exe like any other cli tool.

Right click both .bat files and select `Create Shortcut`. Then press `Win+R` and enter `Shell:sendto`. This will open a new window. Drag both shortcuts into that window. Then rename them `Repack folder with unreal_pak_cli` and `Unpack .pak with unreal_pak_cli`.

2.2.3 Extracting the Game Files

First create a new folder in your `AstroneerModding` folder called `GameFiles`. This will be where you will extract the game files to.

Next, you will need to find your Astroneer's .pak file by going to where you have Astroneer installed.

- The usual location for this is `C:\Program Files (x86)\Steam\steamapps\common\ASTRONEER\Astro\Content\Paks`.
- If you have the Game installed anywhere outside the default Steam location you can find it on Steam by right clicking on Astroneer then going to `Properties` and then `Local Files > Browse`. From there go to the sub folder `Astro\Content\Paks`.

Copy the `Astro-WindowsNoEditor.pak` files to your `GameFiles` folder. Finally in your `GameFiles` folder right click on the `Astro-WindowsNoEditor.pak` file and select `Send to > Unpack .pak with unreal_pak_cli`. This may take a minute.

2.2.4 Setting up UAssetGUI

UAssetGUI is tool for viewing and editing cooked .uasset files. Downlaod the latest version from GitHub. Extract the .zip file to your `AstroneerModding` folder.

2.2.5 Opening an Asset

Go to `AstroneerModding\GameFiles\WindowsNoEditor\Astro\Content\Items\ItemTypes` and double click on `FloodLight_IT.uasset`. Since you probably have not opened one of these files before Windows will ask you for a program. Select "more apps," then "Look for another app on this PC." Browse to and select the `UAssetGUI.exe` we extracted earlier. This will set UAssetGUI as the default program for .uasset files, which you will need for making your first mod.

To make your first mod continue with *Basic Modding*.

2.3 Basic Modding

Contents

In this section you will learn how to create your first mod. We will start by simply reducing the byte cost of the Floodlight.

2.3.1 How Making a Mod Works

Using UAssetGUI is a fairly simple way to create a mod. The process, which is covered in detail below, involves making a copy of Astroneer's file structure (while only containing the necessary files) inside your mod folder. Then, UAssetGUI is used to modify those game files to your liking. Once the metadata file is set up correctly, the mod folder is packaged using unreal_pak_cli, and your mod is ready to use.

> While the ways in which the game files are modified will vary, most other methods of creating mods involve a similar process as this one.

2.3.2 Setting up Folders

First start by creating a new folder called `TutorialMod` in your `AstroneerModding` folder. We will be using that folder in this and future tutorials.

Next create another folder inside the `TutorialMod` folder called `000-TutorialMod-0.1.0_P`. This name matches what the mod .pak file will later be called. You can read more on the specifications for mod .pak files in the *.pak Mod File Names* section of the documentation.

2.3.3 Creating metadata.json File

Each mod must have a metadata.json file at the root. This file contains information about the mod. In your `000-TutorialMod-0.1.0_P` folder create a file called `metadata.json`. Then paste the following JSON into the file. Make sure to replace `YOUR_NAME` with your name.

```
{
    "schema_version": 2,
    "name": "Tutorial Mod",
    "mod_id": "TutorialMod",
```

(continues on next page)

(continued from previous page)

```
    "author": "YOUR_NAME",
    "description": "A tutorial mod.",
    "version": "0.1.0",
    "sync": "serverclient"
}
```

2.3.4 Finding the Asset to modify

Because we are modifying an item that is already in the game we need to find the asset of the item. We want to change the byte cost of the floodlight so we need to find the ItemType asset of the floodlight. Sometimes finding the proper file can hard, but Windows search can help.

The asset file we are looking for is located at `GameFiles\Astro-WindowsNoEditor\Astro\Content\Items\ItemTypes\FloodLight_IT.uasset` and `GameFiles\Astro-WindowsNoEditor\Astro\Content\Items\ItemTypes\FloodLight_IT.uexp`. It is important to remember that each asset is split into two files and you always need to keep both together. Now we are going to copy the two files over to our folder so that we can modify them. Copy them to `TutorialMod\000-TutorialMod-0.1.0_P\Astro\Content\Items\ItemTypes\FloodLight_IT.uasset` and `TutorialMod\000-TutorialMod-0.1.0_P\Astro\Content\Items\ItemTypes\FloodLight_IT.uexp`. You will have to create the folders if they do not exist yet. It is very import that the files have the exact same folder structure as the game.

Never directly modify the files in your `GameFiles` folder. Should you wish to create a different mod utilizing those same files, you will have to recreate the entire GameFiles folder if you accidentially save over the original files.

2.3.5 Modifying the Asset

Now open the `FloodLight_IT.uasset` file in the `TutorialMod` folder using UAssetGUI.

Each asset is made from multiple sections. The most interesting ones are usually `Name Map`, `Import Data` and most importantly the `Export Data` section. Assets can have a few to hundreds of exports. Exports are roughly labeled with what they contain. Finding the correct export for what you are looking for requires some practice. For small assets like this one, you can go to View > Expand All to see all the subproperties of the asset.

Here you will quickly spot the `ItemCatalogData` export. This is the export that contains information like unlock cost and other stuff relevant for the catalog. If you are unable to find what you are looking for, you also can search for specific text in UAssetGUI by pressing `ctrl+f` and typing your text into the search bar.

Each export has multiple subproperties arranged in a tree structure. You can see the subproperties by expanding the export. Click on `ItemCatalogData(7)` to see the contents of the catalog data. This export has a very simple structure.

At the very top of the table view there is an `UnlockCost` property with a number to the right. Simply the change the number to what amount of bytes you would like the floodlight to cost. For example, `500`.

Then simply press `Ctrl+S` to save the file, or click on `File > Save`. You should see two `.bak` files appear. Ignore these files, they are simply backup files and are ignored by the software.

2.3.6 Packing the Mod

Close UAssetGUI and go back to your `TutorialMod` folder. Right click on the `000-TutorialMod-0.1.0_P` folder, then select `Send to > Repack folder with unreal_pak_cli`. This will create a `000-TutorialMod-0.1.0_P.pak` file in the same folder.

2.3.7 Installing the Mod

Double-check that the .pak file in the `TutorialMod` folder is named correctly. If not, rename it to `000-TutorialMod-0.1.0_P.pak`. Once you have verified that the name is correct, simply drag the newly created `000-TutorialMod-0.1.0_P.pak` file onto the Modloader window and you should see `Tutorial Mod` appear in the mod list. Make sure it is enabled in the modloader window.

2.3.8 Verifying the mod works

If the mod loads into the modloader without issues, start your game. If the modloader has issues loading the mod, double check that you followed the tutorial exactly. Then head to the catalog and check that the floodlight is now costs 500 bytes. If your mod has other issues that you need additional help with, joining the Astroneer Modding Discord is your best bet.

2.4 Setting up the Modding Kit

Contents

These are the requirements for making custom items with Unreal Engine and the Modding Kit.

2.4.1 Visual Studio

Download Visual Studio 2017 or higher.

When running the installer, on the "Workloads" page make sure you select "Desktop & Mobile > Desktop development with C++" and "Gaming > Game development with C++".

If Visual Studio is already installed you can run the installer and press modify to add the necessary workloads.

2.4.2 Unreal Engine 4

Open Epic Games launcher and go to "Unreal Engine" tab.

Next go to Library and add a new engine version.

Select 4.23.x where x can be any number, and press install.

2.4.3 Modding Kit

To develop your mods you will need a modkit which can be downloaded from this link.

If you are familiar with version control software you should clone it for easier updates.

2.4.4 Wwise (optional)

Astroneer uses Wwise as its sound engine. If you want to make mods which play sounds you need to install Wwise.

Go to Wwise website and head to `Get Wwise` -> `Download Wwise`.

If you don't already have an account, create one. After wwise installed has finished downloading, open it and select WWISE in the top bar.

Click on latest and change it to `All` > `2019.1` > `2019.1.8.7173` and press install. Once presented with options select these:

- Packages
  - Authoring
  - SDK (C++)
- Deployment Platforms
  - Apple
    - macOS
  - Microsoft
    - Windows
    - Visual Studio 2017
    - Visual Studio 2019

After it has finished installing go to `Unreal Engine` tab in the top bar. There press on the burger menu and `Browse For Project`.

Select `Astro.uproject` in the file picker.

Now press `Integrate WWise into project`. Here select `All` > `2019.1` > `2019.1.8.7173`.

In the wwise project path field press on the triangle on the right side and click `New`.

Now press the `Integrate` button.

2.4.5 Generating project files

To generate the project files we will need to run the following command:

```
"UE_INSTALL_PATH\Engine\Binaries\DotNET\UnrealBuildTool.exe" -projectfiles -project=
↪"PATH_TO_PROJECT\\Astro.uproject" -game -rocket -progress
```

Open cmd in your project directory and copy this inside cmd. Remember to replace **UE_INSTALL_PATH** with your unreal engine installation folder which is usually found at `C:\\Program Files\\Epic Games\UE_4.23\\`.

And remember to change **PATH_TO_PROJECT** with path to the modkit.

Example:

```
"C:\Program Files\Epic Games\UE_4.23\Engine\Binaries\DotNET\UnrealBuildTool.exe" -
↪projectfiles -project="C:\\Users\\username\\Documents\\Astro.uproject" -game -rocket -
↪progress
```

Run the command, and then open the `Astro.uproject` file and if it asks to build unbuilt modules press yes and wait.

2.4.6 Developing Mods

You can now start with *Adding Custom Items with the Unreal Editor*.

2.5 Adding Custom Items with the Unreal Editor

Contents

2.5.1 Creating the Mod Folder

After opening the project in unreal editor you will need to make a folder for your mod.

In the content browser head to `Mods` folder and create a folder `YOUR_USERNAME`, and open that folder.

You probably would want to change `YOUR_USERNAME` to your actual username.

Inside that folder create a folder called `TutorialMod` and open it.

2.5.2 Creating an Item

Items in Astroneer consist of two components, ItemType and PhysicalItem.

ItemType (you'll use an _IT suffix for these files) is used by Astroneer to get your item's recipe, menu icon, item properties, description, etc.

PhysicalItem (you'll use an _BP suffix for these files) is the object that is going to be created in the world.

These are different from the 3D mesh that you'll import, which has no suffix in its name and stores the 3D model of the object you'll see inside the game.

Creating the PhysicalItem

Your first step is to create the PhysicalItem blueprint class. Start by making sure Unreal Editor is in the `TutorialMod` folder before proceeding.

Click on "Create a blueprint class" and a pop-up window will appear. Click on "open All classes" if it isn't opened already.

In the search bar above the class type list, type in "PhysicalItem" and select it once it appears. This will determine what the blueprint will inherit from.

Then, create the class. Once the dialog window closes, you should be able to type in the name of the blueprint. Set it to `ExampleItem_BP` - it must

match this exactly. Then press enter to save your name, and double-click on it to open up the PhysicalItem's settings menu.

First of all we want to set the 3D model for that item, so that we have something to look at inside the game.

For that we will need to *import the mesh*

Importing the Mesh

1. Open your mod folder in the **Content Browser**
2. Download this example mesh `exampleMesh.fbx`
3. Drag&drop your mesh into the **Content Browser** and click `Import all` at the bottom right. Don't change any settings in this window.
4. Open the mesh and in the **Details** tab search for `Has navigation Data` and uncheck that checkmark.

> **Warning:** If you don't uncheck `Has navigation Data` the game will crash when loading your mod.

After importing the mesh, open the PhysicalItem again that you created earlier, and click on **StaticMeshComponent** in the left-hand side menu.

Note: If you don't see the **StaticMeshComponent** click on **Open Full Blueprint Editor**

There in the **Details** tab, set your mesh in `Static Mesh` field to the example mesh you imported.

Click on **Compile** and then **Save**.

Creating the ItemType

Now that we have created the PhysicalItem, it's time to make the ItemType for it.

Create a second blueprint class, and in the creation dialogue open All classes.

Select ItemType to inherit from, create the class, set its name to `ExampleItem_IT`, and open the class.

This is a simple item so we will leave most of the options untouched, you can experiment with them yourself and see what they do.

- Set **Pickup Actor** to be `ExampleItem_BP`.
- **Open Construction Recipe**
  - Press "+" on the **Ingredients** field.
  - Set Item type of the ingredient to **Astronium**
  - Set Count of the ingredient to 1.
- Set **Catalog Data** to `Item Catalog Data`
- `Is Base Item` in **Catalog Data** is used to determine whether this item will create it's own line inside the catalog or use an existing one from a base item.
- `Base Item Type` in **Catalog Data** is used to determine what row of the item catalog that the item will be listed in.
- `Varitation Sequence Number` in **Catalog Data** is used to determine the order in which the item will be listed in the item catalog.
- `Catalog Mesh` in **Catalog Data** is used to determine the mesh that will be displayed in the item catalog.

> **Warning:** If you enable `Is Base Item` and then set `Base Item Type` to equal another object, your item **WILL NOT** show up in the catalog.

- Set **Base Item Type** to `Consumable_JumpJet_IT` so it gets listed near jetpacks and hoverboards.
- Untick **Is Base Item** because you are using the same row as the jetpacks, which already has a base item.
- Set **Catalog Mesh** to the mesh we imported earlier.
- Set **Crate Overlay Texture** to `ui_icon_package_drill`. This is used to determine the icon that will be displayed on the packaged item.
- Set **Widget Icon** to `ui_icon_comp_drill`. This is used to determine the icon that will be displayed in the item catalog and on hovering on the item.

Open **Control Symbol** section and fill the fields out like this:

- **Name**: `ExampleItem`
- **All caps Name**: `EXAMPLEITEM`
- **Tooltip Subtitle**: `Example Item`
- **Description**: `This is an example item.`

These four entries do not have to match the names of your objects, they are used to determine the text used in the research catalog and tooltips in-game.

Linking ItemType and BP together

Now open the `ExampleItem_BP` yet again, and click on **ItemComponent**.

On the right open the **Item Component** dropdown, there, set the **Item Type** to `ExampleItem_IT`.

2.5.3 Cooking the Mod

Remember to save every asset you have changed before cooking.

Click on **File > Cook Content for Windows**

After the content has been cooked, create a folder in file explorer with the name `000-TutorialMod-0.1.0_P` and open this folder.

Note: This folder must be outside of unreal project.

Inside this folder create a file called `metadata.json`.

This file is responsible for telling the modloader where to find mod files for certain parts of the mod.

Fill this file out like this

```
{
    "schema_version": 2,
    "name": "Tutorial Mod",
    "mod_id": "TutorialMod",
    "author": "YOUR_NAME",
    "description": "A tutorial mod.",
    "version": "0.1.0",
    "sync": "serverclient",
    "integrator": {
        "item_list_entries": {
            "/Game/Items/ItemTypes/MasterItemList": {
                "ItemTypes": [
                    "/Game/Mods/YOUR_USERNAME/TutorialMod/ExampleItem_IT"
                ]
            },
            "/Game/Items/BackpackRail": {
                "PrinterComponent.Blueprints": [
                    "/Game/Mods/YOUR_USERNAME/TutorialMod/ExampleItem_BP"
                ]
            }
        }
    }
}
```

Replace `YOUR_USERNAME` with your name.

`/Game/Items/ItemTypes/MasterItemList$ItemTypes` contains ItemTypes for all items so we register our ItemType with this.

`/Game/Items/BackpackRail$PrinterComponent.Blueprints` contains ItemTypes that can be crafted so we need to register here too.

More info about the format can be found in *Modding standards*

In this folder, also create a folder structure like this `Astro/Content/Mods/YOUR_USERNAME`.

Now go to the unreal project folder and navigate to `Saved/Cooked/WindowsNoEditor/Astro/Content/Mods/YOUR_USERNAME` and copy `TutorialMod` folder to the folder we created previously.

So that the folder structure looks like this:

```
000-TutorialMod-0.1.0_P
    ├──metadata.json
    │
    └──Astro
        └──Content
            └──Mods
                └──YOUR_USERNAME
                    └──TutorialMod
```

Warning: Files in `ModdingKit/Saved/Cooked/WindowsNoEditor/Astro/Content/Mods/YOUR_USERNAME` and `UE_PROJECT/Content/Mods/YOUR_USERNAME` are different. Where UE_PROJECT is the path to unreal project. The first location contains the cooked files, while the second one contains the uncooked ones. You **MUST** copy from the first location because the game only accepts cooked ones.

Now that the mod structure is complete, time to pack the mod.

For packing the mod we will be using `unreal_pak_cli`.

To make life easier for us we have created `unreal_pak_cli` that will help us pack your mod folder, download and extract them.

Now that the scripts and the program are extracted we can pack our mod. Open two file explorer windows, one with the repack.bat file, and the other showing your mod folder.

Next, drag and drop your project's main folder (000-TutorialMod-0.1.0_P) onto the `repack.bat` file.

After unreal_pak_cli finishes you should be able to see `000-TutorialMod-0.1.0_P.pak` file.

To load this mod, drag&drop it onto the modloader window and check the checkbox to enable it.

After all this work you should be able to print your first item.

2.6 Picking Names

Contents

2.6.1 Naming is hard

Naming things is hard. That is something everyone has experienced at some point. But choosing good names for your mod(s) is even harder becasue they also need to be descriptive, short and readable. So here are some tips on how to do it right.

2.6.2 Allowed Characters

All IDs can generally only contain upper- and lowercase latin characters and the numbers 0-9. This restriction exists becasue they need to work in URLs and filepaths but also to avoid confusion.

2.6.3 What you need to choose

- Your Author ID (only once of course)
- A mod ID
- A name for the mod

2.6.4 Author ID

An Author ID is used to uniquely identify a person creating mods. It should be whatever nickname you have choosen condensed into the allowed characters. Note that this one is rarely shown to users and the one shown to users can contain basically any characters.

2.6.5 Mod Name and ID

Whatever you choose to call your mod, it has to clearly describe what the mod it/does while also being relatively short. Especially the mod ID has to be readable to allow users to identify mod files just by their file name.

It is recommend to use 2-3 full words. Do not use acronyms, unless they are already used by the base game (like `RTG`). Also avoid shortening words except for stuff like `Astroneer` => `Astro`. The mod name should include spaces (unlike the ID) and can also be a bit longer.

Here are some good examples:

- `RocketLauncher` for Rocket Launcher Mod
- `LavaLamp` for Lava Lamp Mod
- `MoreTradables` for Mo' Tradables

Here are some bad ones:

- `QTRTG` for Print QT-RTG. Here it is unclear what the mod does in the mod ID.
- `Astronium` for Super Astronium. Not even from the full name it is clear what the mod does.
- `Pumpkin` for PUMPKINS!!!. Again what does it do.
- `6A1S`. What does this even mean?

2.6.6 Mod ID extension

You can extend your mod ID with your author ID like this `modid.authorid`. In practice it looks like this `PickupRovers.Konsti`. This is to differentiate two mods with the same base mod ID that are by different authors. This new string will be the new mod ID used in both filename and metadata. Also this is the only time a single dot is allowed in IDs. Note that this is a relatively new addition and not all programs/website will support it.

2.7 Adding Missions

Contents

2.7.1 Adding the Mission Trailhead

Right click in the **Content Browser** and add a folder called `Missions`.

Inside this folder create a **Data Asset**

Note: **Data Asset** is located inside **Miscellaneous** in the right click menu.

Here select the `AstroMissionDataAsset` class and name your asset `MissionTrailhead-TutorialMod`.

Inside this asset you can define as many missions as you want, click on "+" to add a new mission.

Here we will fill out some data to tell Astroneer where to put our mission.

- **MissionId**: `TutorialMod-TestItemMission`
- **MissionCategory**: `TutorialMod`
- **Description**: `A mission that unlocks TestItem`
- **Notification Color**: `Astro Blue`
- **Byte Reward Value**: `1000`
- **Notification Icon**: `ui_icon_nug_astronium`
- **Notification Color** is the color of the notification that will be shown when you complete the mission.
- **Notification Icon** icon of the notification that will be shown when you complete the mission.
- **Prerequisite Missions** is a list of missions that must be completed before this mission can be completed.
- **Next missions** missions that will be unlocked after this mission is completed.

Now let's actually add objectives, for this tutorial we will be requiring the player to collect 2 pieces of clay.

Press "+" on **Objectives**

- **Description**: `Collect 2 pieces of clay`
- Add a new **Target type** and set it to **Clay**

- **Value**: `2.0`
- **Progress Notify Threshold**: `2.0`
- **Objective Type**: `Harvest Resource`
- **Value** determines the amount of resource we want to collect for this objective
- **Progress Notify Threshold** determines the amount of resource we need to collect to get the progress notification.

And now we can go ahead and add the reward, in this case we will give the player the TestItem.

Press "+" on **Rewards** and set the reward type to be `ExampleItem_BP` and the value to 1.

This should provide us with a basic mission for the player to complete.

Now we must add it to our mod.

2.7.2 Adding Mission Trailhead to the Mod

As usual cook the content and move it to the mod folder, `metadata.json` will be used from *Adding Custom Items with the Unreal Editor* with some changes.

We need to add this to our metadata for the modloader to add it into Astroneer mission system.

```
"mission_trailheads": [
    "/Game/Examples/TutorialMod/Missions/MissionTrailhead-TutorialMod"
]
```

So the file looks like this:

```
{
    "schema_version": 2,
    "name": "Tutorial Mod",
    "mod_id": "TutorialMod",
    "author": "YOUR_NAME",
    "description": "A tutorial mod.",
    "version": "0.1.0",
    "sync": "serverclient",
    "integrator": {
        "item_list_entries": {
            "/Game/Items/ItemTypes/MasterItemList": {
                "ItemTypes": [
                    "/Game/Examples/TutorialMod/ExampleItem_IT"
                ]
            },
            "/Game/Items/BackpackRail": {
                "PrinterComponent.Blueprints": [
                    "/Game/Examples/TutorialMod/ExampleItem_BP"
                ]
            }
        },
        "mission_trailheads": [
            "/Game/Examples/TutorialMod/Missions/MissionTrailhead-TutorialMod"
        ]
    }
}
```

Now cook the mod as in *Adding Custom Items with the Unreal Editor* and check it out!

2.8 Adding Mission Panels to items

Contents

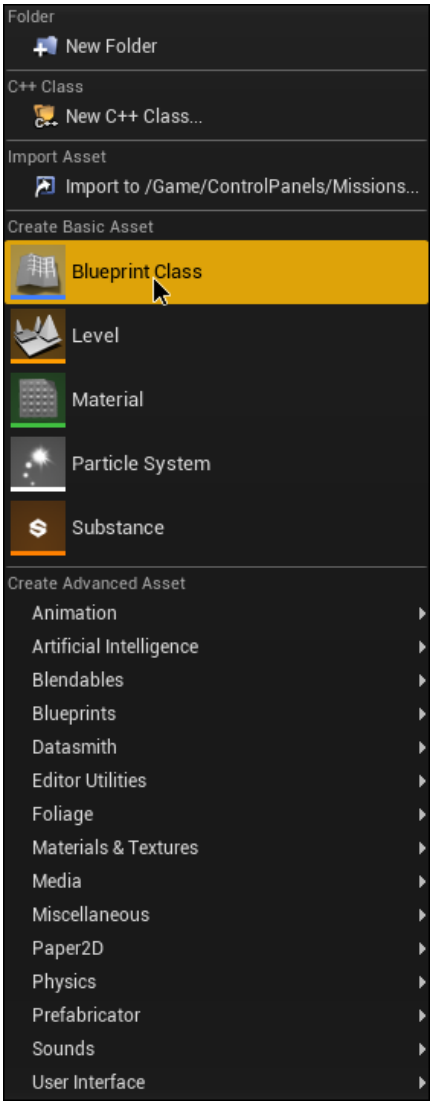
2.8.1 Creating the Mission Panel

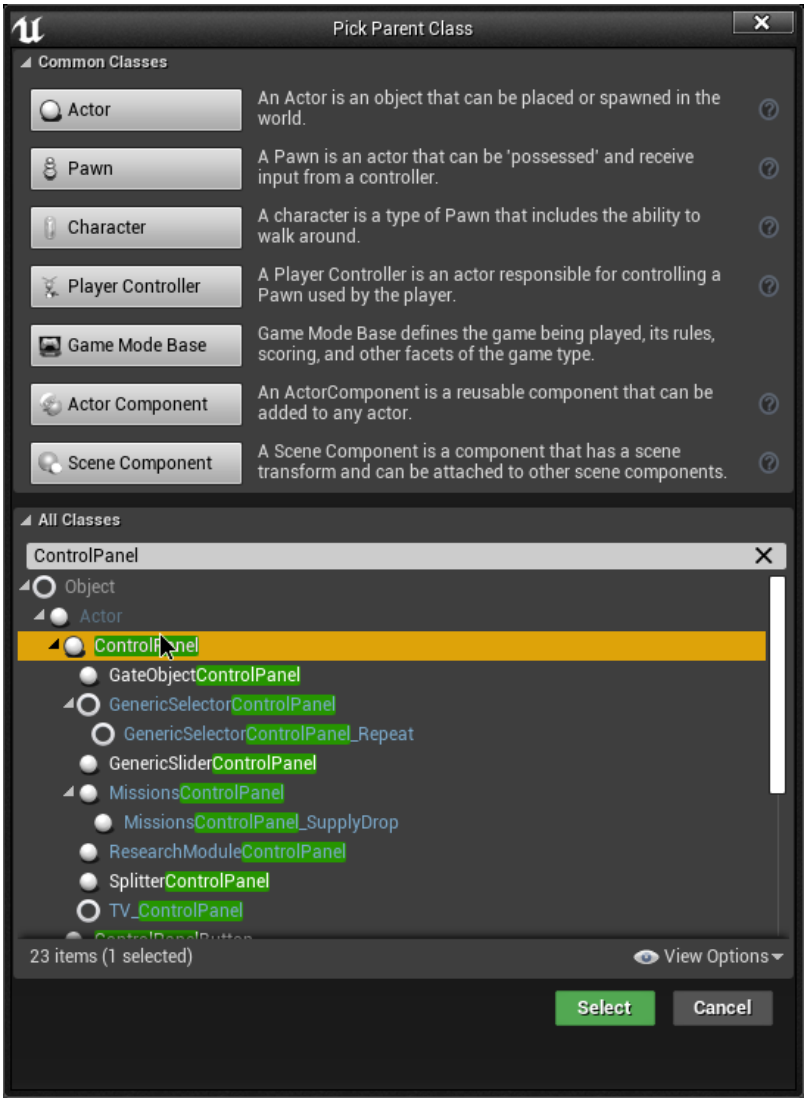
Note: This is an advanced topic, so it is preferred that you already have some experience modding Astroneer.

In Unreal Engine, Starting from the `Content` folder in the **ModdingKit** Project, right click in the **Content Browser** and create a folder named `ControlPanels` then in that folder create another called `Missions`, and after creating those directories the path should now look something like this: `/Content/ControlPanels/Missions/` in the **Content Browser**.

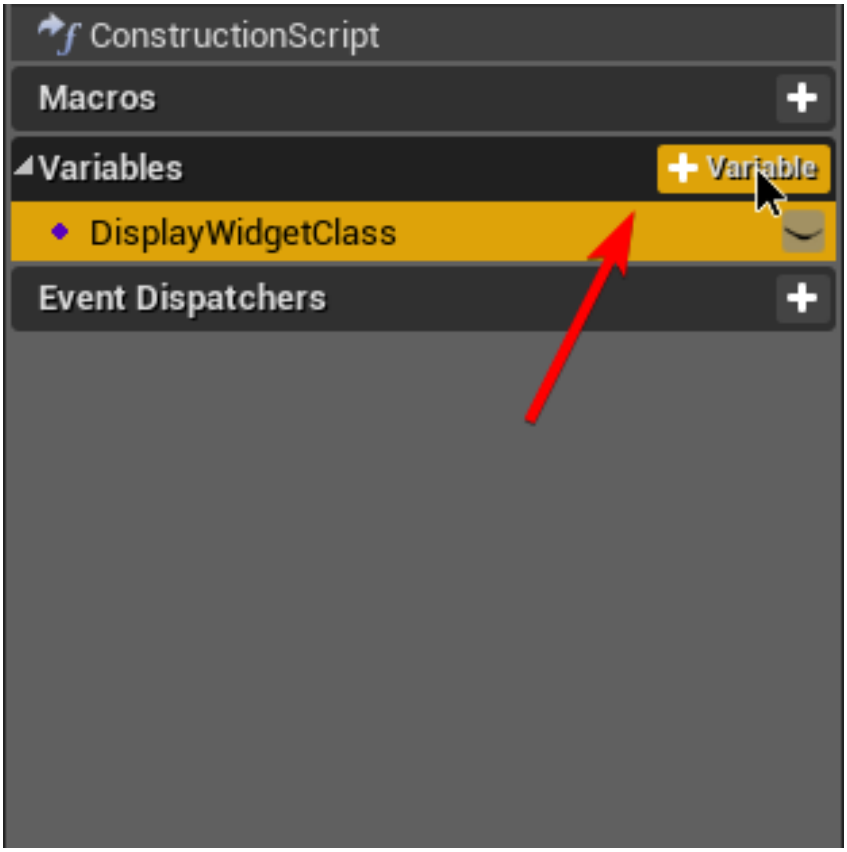
Note: Any asset created in the `/Content/ControlPanels/Missions/` directory, is not to be packed with the mod itself, we are creating these assets just for referencing purposes.

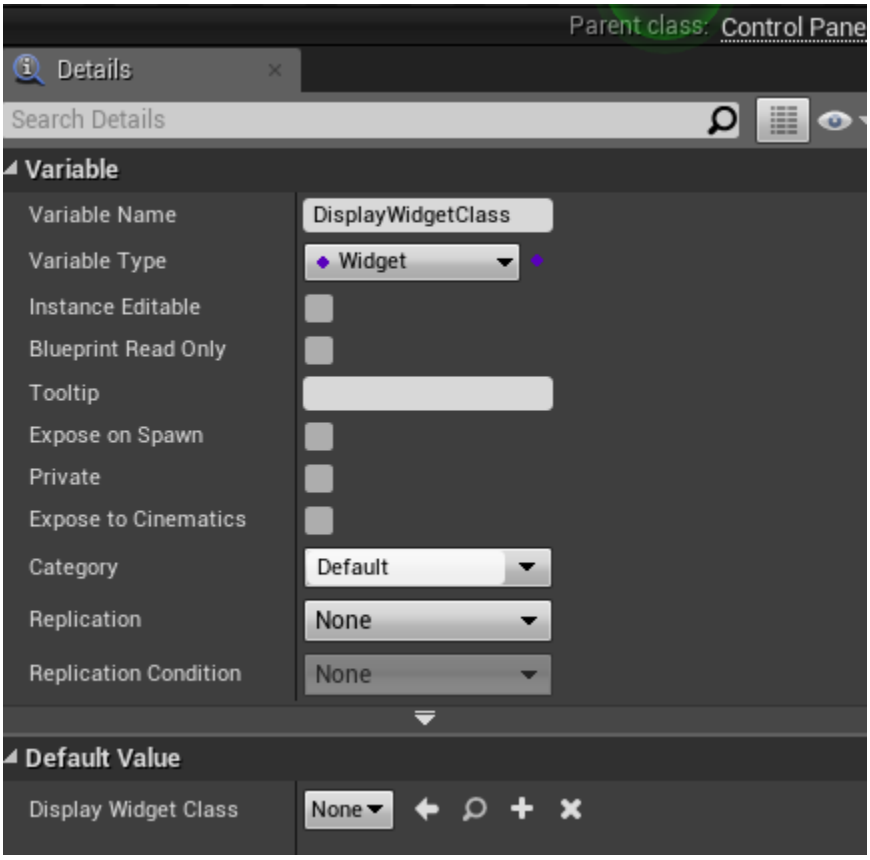
Go to the `/Content/ControlPanels/Missions/` directory that you just created in the **Content Browser**, and right click in the **Content Browser** to create a new BP Class that inherits from the `ControlPanel` Class, and name it `MissionsControlPanel`.


Folder
New Folder
C++ Class
New C++ Class...
Import Asset
Import to /Game/ControlPanels/Missions...
Create Basic Asset
Blueprint Class
Level
Material
Particle System
Substance
Create Advanced Asset
Animation
Artificial Intelligence
Blendables
Blueprints
Datasmith
Editor Utilities
Foliage
Materials & Textures
Media
Miscellaneous
Paper2D
Physics
Prefabricator
Sounds
User Interface

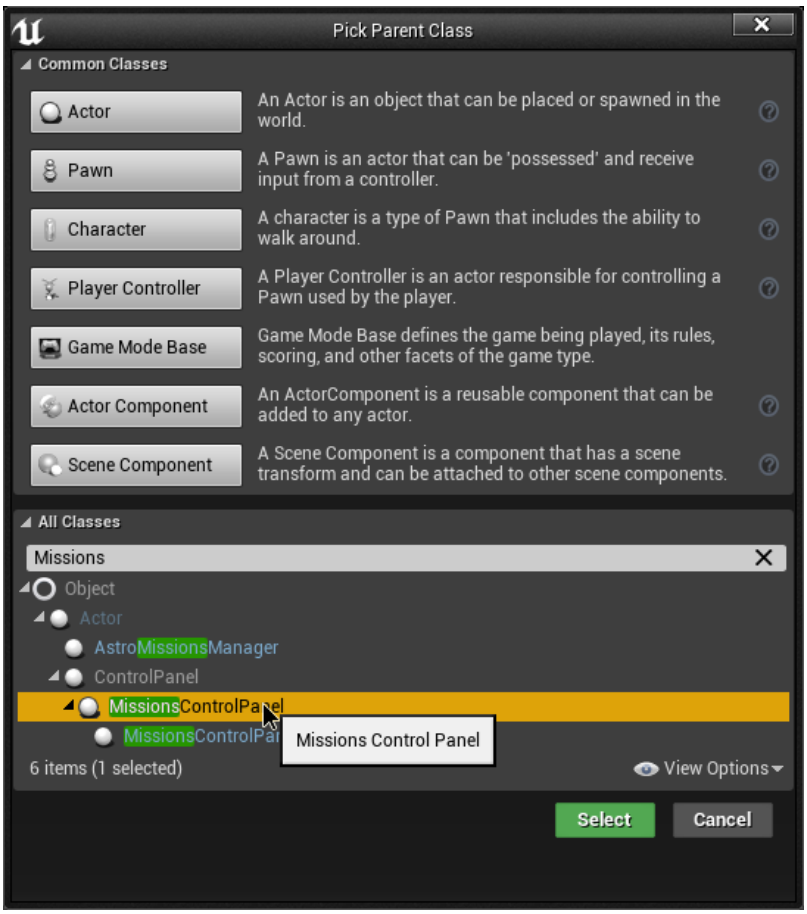


In the asset, create a variable and name it exactly this `DisplayWidgetClass`, then in the properties of the variable set it to be a `Widget` Class Reference, after doing that now compile and save this asset.

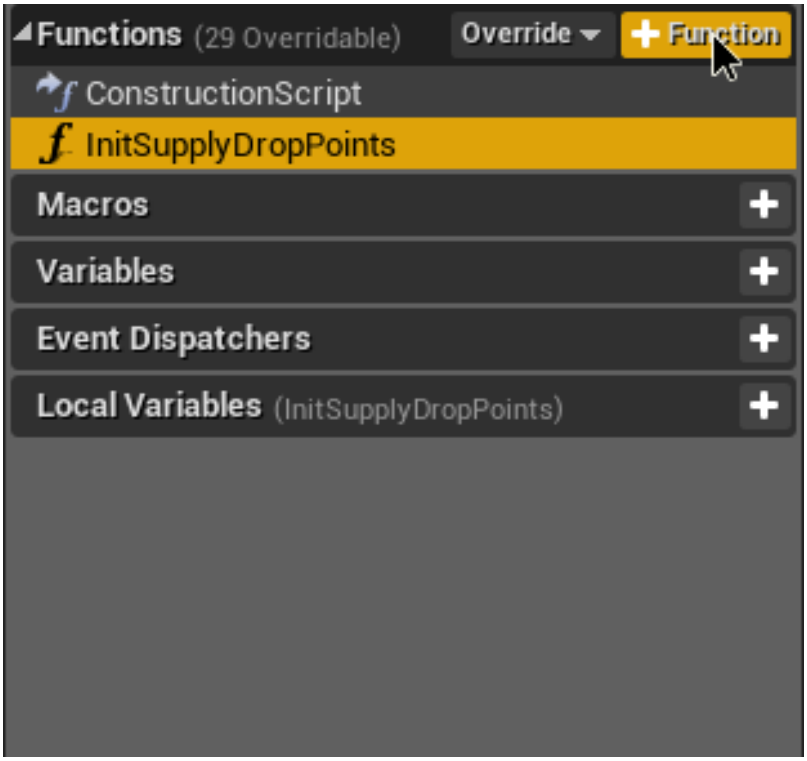

ConstructionScript
Macros
Variables
+ Variable
DisplayWidgetClass
Event Dispatchers

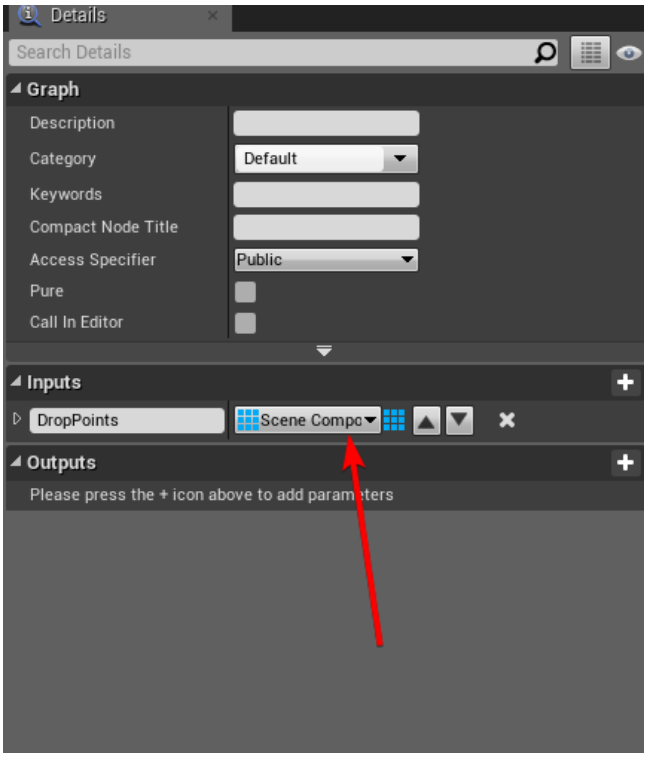

Parent class: Control Pane
Details
Search Details
Variable
Variable Name
DisplayWidgetClass
Variable Type
Widget
Instance Editable
Blueprint Read Only
Tooltip
Expose on Spawn
Private
Expose to Cinematics
Category
Default
Replication
None
Replication Condition
None
Default Value
Display Widget Class
None

Now back to the `Missions` folder in the **Content Browser**, right click in the **Content Browser** and create a new BP Class that instead now inherits from the `MissionsControlPanel` and name it `MissionsControlPanel_SupplyDrop`.

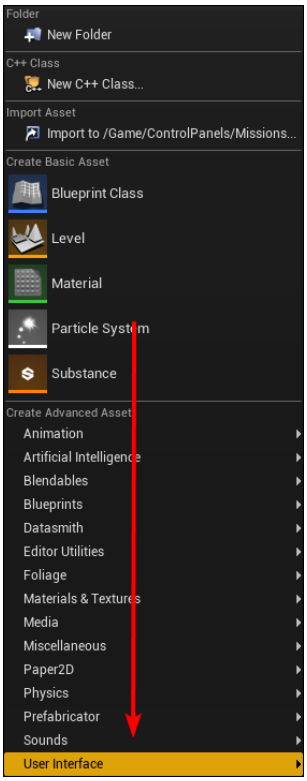


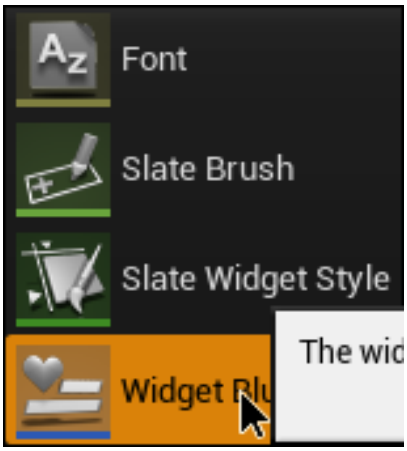
Now in this asset, create a new function and name it `InitSupplyDropPoints` and then add an input which should be an array of `Scene Component` Object References, now compile and save this asset.



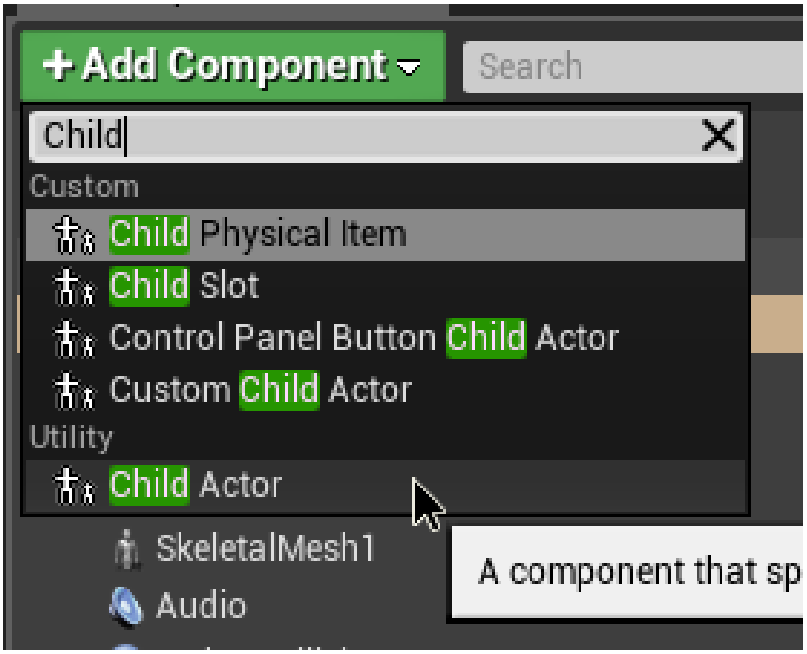


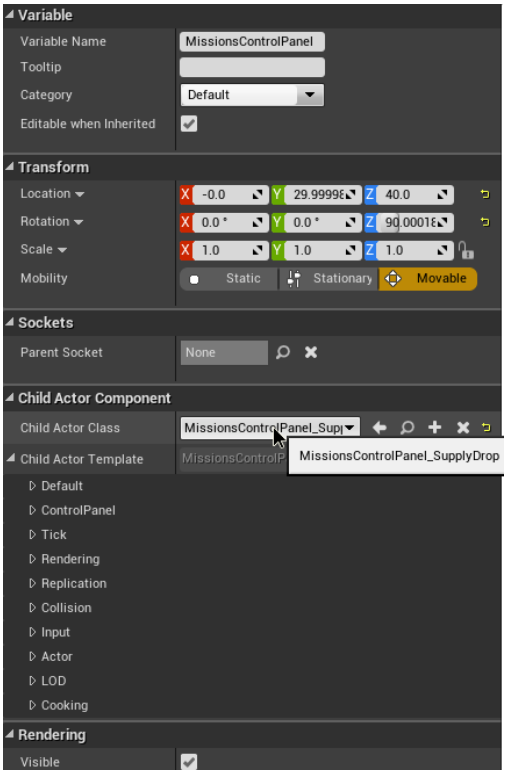
Again, back to the `Missions` folder in the **Content Browser**, now right click in the **Content Browser** and create a Widget Blueprint by going to `User Interface -> Widget Blueprint` in the right click menu, and name it `MissionPanelPrimary`



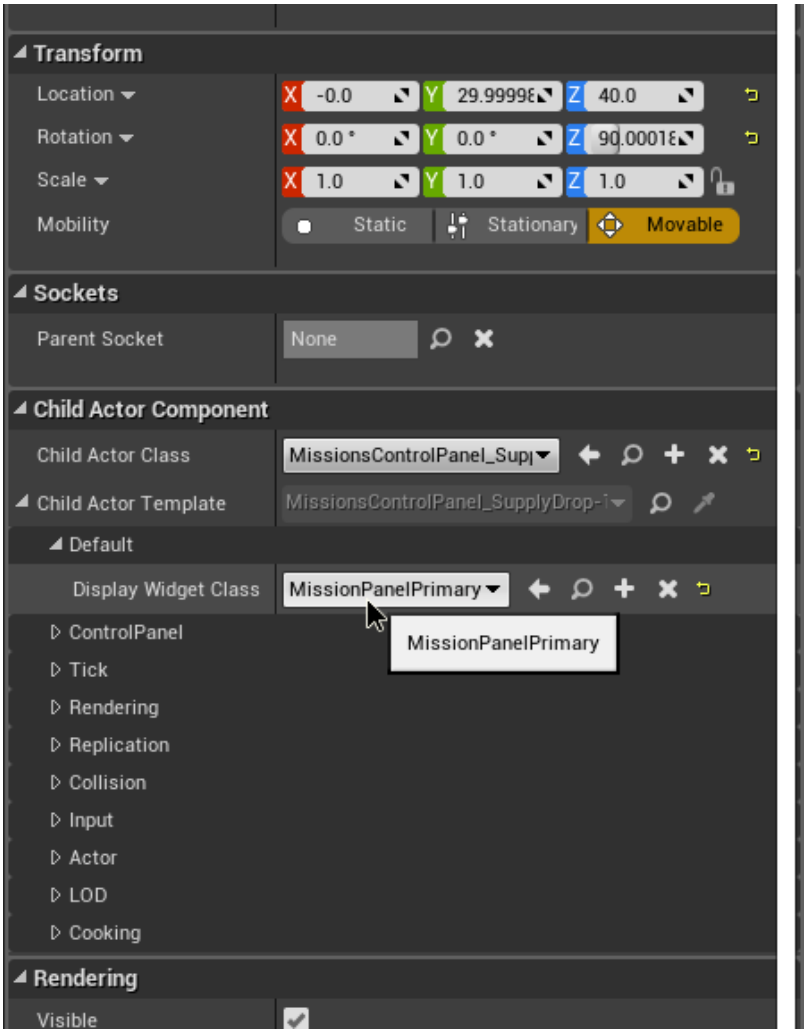


After creating the assets and the functions/variables for them, go to your item's BP, and create a `Child Actor Component` in the BP, position it how you like and set the `Child Actor Class` to the `MissionsControlPanel_SupplyDrop` class.



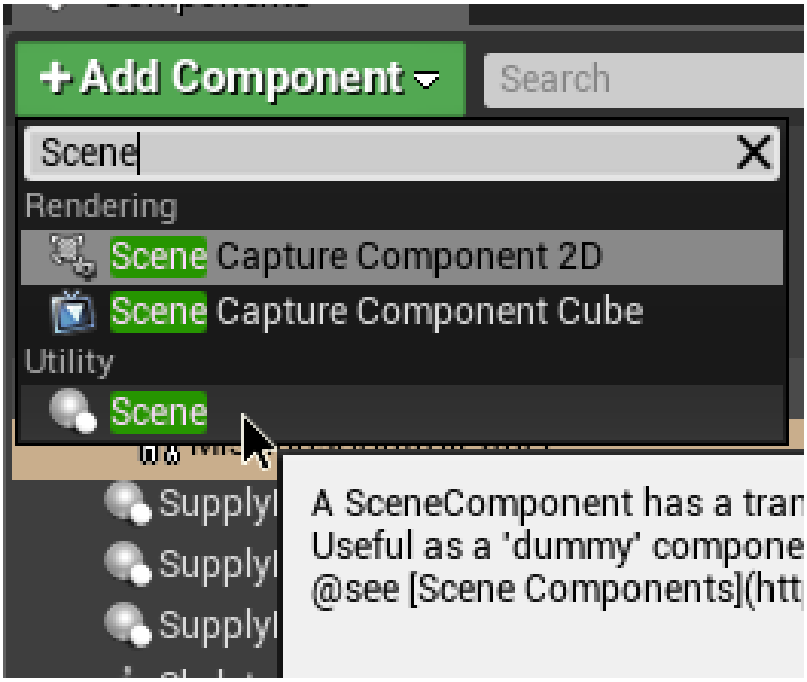


Now in the `Child Actor Template` dropdown, click on the Default dropdown and set the `DisplayWidgetClass`'s class to `MissionPanelPrimary`



2.8.2 Creating & Linking the Supply Drop Points

In your items BP create 3 new `Scene` Components and name them accordingly (ex. `SupplyDropPoint` 1, 2, 3)

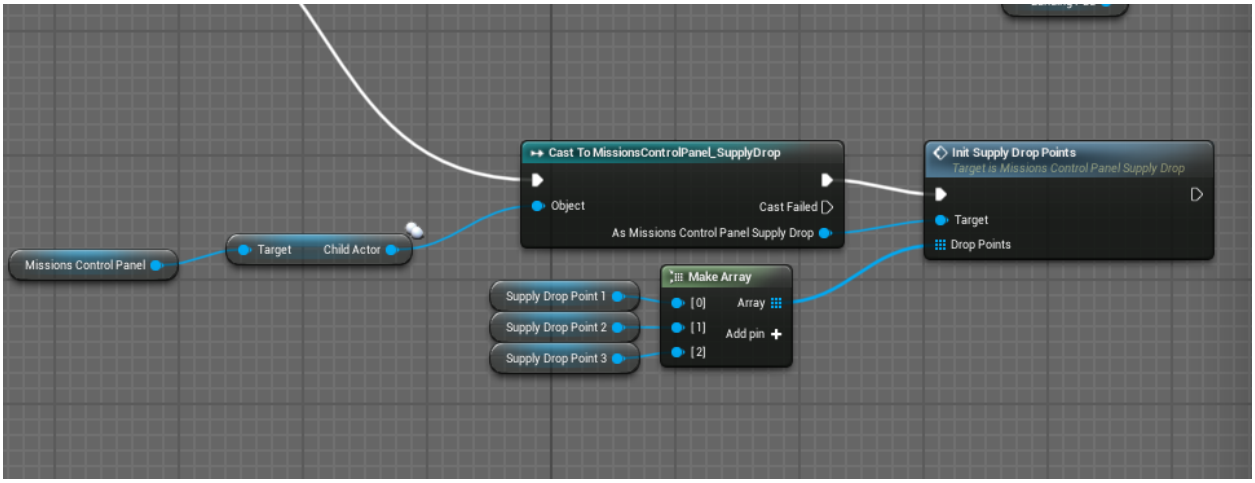


Then in the BP Graph, add a `Cast to MissionsControlPanel_SupplyDrop` node, and get your Child Actor Component's `Child Actor` by using the `Get Child Actor` and have the target be the `Child Actor Component`, and connect the `Actor Object Reference` from the `Get Child Actor` node to the `Object` input on the `Cast to` node.

Now connect the `Cast to` node's Exec input to somewhere on the `Begin Play` Event, and then call the `InitSupplyDropPoints` using the `As MissionsControlPanel_SupplyDrop` output.

Get the SupplyDropPoints made earlier and use a `Make Array` node to turn all 3 into an array and connect the array output to the `Drop Points` input on the function

(use image below for reference if confused)



2.8.3 Result

There you have it, now you can compile and save, package the mod and you should have a working mission panel on your item!

2.9 Diegetic UI

Contents

2.9.1 Making Diegetic UI

To add diegetic UI to your object we will need another actor.

This actor will be the actor that displays our ui.

Create an actor with parent class of **ControlPanel**.

First things first we need to have a component that describes orientation when the actor is displayed to the player.

For that we will add a **Scene Component**, position and rotate it as you wish.

Set the name of this component to `CrackedOrientation`.

For this component to actually do something we must specify its name inside **CrackableActorComponent**.

Open **CrackableActorComponent** and on the right side you should see a `Cracked Orientation Component` dropdown, open that dropdown.

In the dropdown set **Component Property** to be `CrackedOrientation`.

Also set these parameters:

Method: Set this to `Hover Face Camera`. This will make the object always follow camera when examining. **Click to close**: Uncheck this. This will prevent the ui from closing when we click anywhere. **Cracks only on examine**: Check this. **Screen scale ratio**: Set this to `1.8`. This determines the size of our object on screen when examining it. **Camera space offset**: Set this to `450.0 175.0 -20.0`. This determines the position of our object on screen. **Pivot angle**: Set this to 0.

But this will not do anything useful unless we setup **ClickableComponent** to actually open our ui.

Open **ClickableComponent** and set the following parameters:

Slow virtual cursor on hover: Uncheck this. **Has use interaction by default**: Check this. **Use action requires hold**: Uncheck this. **Default use context**: Set this to `UC Examine`. This deterimines which action will be displayed to the player when they hover over the object.

Now we need something to actually render on our screen, for this we will be using this example panel `panel.fbx`.

When importing this mesh to unreal make sure to check **Skeletal Mesh** under **Mesh** section and set the skeleton to None.

Now open the **Mesh** dropdown in **SkeletalMesh** component. Set **Skeletal Mesh** to be the mesh we just imported.

Note: We are using a skeletal mesh, because if we use a static mesh for this, it will not be rendered in game.

Warning: If the **Camera** or **Visibility** channel collision will be enabled on any of the child components in this blueprint, it will cause your camera to glitch.

2.9.2 Adding Control Panel to the item

Now that we created the control panel it's time to add it to the item.

Open your item blueprint.

Note: If you were following the previous tutorial it is probably `ExampleItem_BP`

In the item blueprint add a **Child Actor** component. On this component set the **Child Actor Class** field to the control panel we created previously.

Remember to position child actor component in the viewport where you want it to be.

We also need to set up **ClickableComponent** in this blueprint too.

Open **ClickableComponent** and set the following parameters:

Slow virtual cursor on hover: Uncheck this. **Has use interaction by default**: Check this. **Use action requires hold**: Uncheck this. **Default use context**: Set this to `UC Examine`. This deterimines which action will be displayed to the player when they hover over the object.

At this point, the diegetic UI is done, so cook the mod and test it!

2.10 Procedural Generation

Contents

2.10.1 Procedural Generation

Note: This is an advanced topic, so it is preferred that you already have some experience modding Astroneer.

Procedural generation in astroneer is done using their custom generation graph that for shipping games gets compiled to low-level code.

Thus we cannot easily change a lot of its behaviors, but what we can do, is use already existing behaviors and modify them a little bit.

To generate our object we must have a **Procedural Placement** and a **Procedural Modifier** class in our mod files.

Procedural Modifier is like a model of spawning, the only settings we can change are **Procedural Placement** which is perfect because that is what we will be doing. **Procedural Placement** is placements that are actually responsible for spawning your items.

Before creating them though, we must know that there is a restriction on filenames that dictate procedural generation in your mod.

This restriction comes from the Astroneer has a lot of its generation code compiled to low-level asm.

In your modkit you can find the file that describes procedural generation names restrictions in `Content/Mods/localcc/TPuzzle/proceduralGenerationModels.json`.

This is a json file that contains an object. Keys in this object are valid names for **Procedural Modifier** classes.

Keys can also be treated as "Models" that we can choose from to get different spawning behaviors.

Values for each key are valid names for **Procedural Placement** classes that the **Procedural Modifier** can contain.

Note: Models work cross-biome but not cross-planet (e.g. you can use `Forests_Terran_Props` for the Plains biome on Terran planet, but you cannot use it for Arid planet)

Planets in this tutorial are called not by their name, but by their type, because that is how the game stores them.

To create those files, right click in the content browser go to `Miscellaneous` and select **Procedural Modifier**.

We will name this one `Plains_Terran_Puzzles`.

Then right click in the content browser again, and this time select **Procedural Placement**.

We will name this one `ObjPl_Puzzle_Surface_Terran`.

Open this asset and set the following parameters:

- **Seed**: Set this to whatever.

- **Radius**: `600`. Radius determines how far objects will be spawned from eachother, treat is if it was objects rarity.
- **Max Projection Distance**: `1500` This is probably max distance the objects can be from eachother.

> **Warning:** **Radius** should always be <= **Max Projection Distance**, otherwise no objects will be spawned.

Now the fun part, spawning the objects. Open variants and click **+**.

In this variant set the following parameters:

- **Selection Weight**: `1.0`. This probably determines the chance for this variant to spawn in case there is more than one variant.
- **Placement Actor**: The actor you want to spawn.
- **Item Type**: This is for resources, if you want custom resource nodes you should set this to the resource type you want to spawn, and **Placement Actor** to **Decorator_MineralFlecks**.
- **Orientation**: `Align to planet up`. This determines how your object will be aligned when spawned.
- **Random Yaw**: Max object random yaw when spawning.
- **Random Pitch**: Max object random pitch when spawning.
- **Uniform scale**: Check this.
- **Scale XYZ**: Set min to `1.0` and max to `1.0`. This will choose a random scale between **min** and **max** when spawning your object.
- **Density scaling slop**: Set min to `1.0` and max to `1.0`.
- **Cull distance**: Set min to `23000` and max to ``25000`. This determines how far away the spawned object can be from the camera before it is culled.
- **Enable Density Scaling**: Uncheck this.
- **Cast Shadow**: Check this. This determines whether the spawned object will cast shadow.
- **Cast Shadow as Two Sided**: Uncheck this. This determines whether the spawned object will cast shadow as if it had a two-sided shader.
- **Receives decals**: Uncheck this.
- **Use as occluder**: Uncheck this. This probably should be checked for very large objects.
- **Should override destruction effects**: Uncheck this.

After setting up the **Procedural Placement** we need to add it to the **Procedural Modifier**.

Open your **Procedural Modifier** asset and add a compiled placement. Set the compiled placement to the one we just created.

Cook the mod files and copy them over to the mod directory.

2.10.2 Writing the metadata

Procedural generation requires a metadata entry to work.

This is an example of what your file should look like

```
{
    "schema_version": 2,
    "name": "ExampleMod",
    "mod_id": "ExampleMod",
    "author": "YOUR_NAME",
    "description": "An Example Mod",
    "version": "0.1.0",
    "sync": "serverclient",
    "integrator": {
        "item_list_entries": {
            "/Game/Items/ItemTypes/MasterItemList": {
                "ItemTypes": [
                    "/Game/Mods/YOUR_NAME/TutorialMod/ExampleItem_IT"
                ]
            }
        },
        "biome_placement_modifiers": [
            {
                "planet_type": "Terran",
                "biome_type": "Surface",
                "biome_name": "Plains",
                "layer_name": "None",
                "placements": [
                    "/Game/Mods/YOUR_NAME/TutorialMod/Plains_Terran_Puzzles"
                ]
            }
        ]
    }
}
```

This will add the procedural modifier to the plains biome on a terran planet.

To get which biomes/layers you can use on which planets there is a file at `Content/Mods/localcc/TPuzzle/biomeData.json`.

For example we want to add something to the valleys biome on exotic planet. In the file we will see something like this:

```
{
    "Exotic": {
        "SurfaceBiomes": {
            "Hills_Exotic": {
                "Layers": [
                "None"
                ]
            },
            "Valleys_Exotic": {
                "Layers": [
                "None"
                ]
```

(continues on next page)

(continued from previous page)

```
            },
            "Rolling_Exotic": {
                "Layers": [
                "None"
                ]
            },
            "Mountains_Exotic": {
                "Layers": [
                "None"
                ]
            }
            },
            "CrustBiome": {
            "Layers": [
                "CrustExotic1",
                "CrustExotic2",
                "CrustExotic3",
                "CrustExotic4"
            ]
        }
    }
}
```

To add a procedural modifier to a biome we must know `planet_type`, `biome_type`, `biome_name` and `layer_name`.

In this file we find an object with key `Exotic`, the key corresponds to `planet_type`.

In this object we see two biome types, `SurfaceBiomes` and `CrustBiome`. We know that valleys are a surface biome, so we look inside the `SurfaceBiomes` object.

Also note that this means that `biome_type` is `Surface`.

Here we can see all of the avaiilable surface biomes, we want valleys so look at `Valleys_Exotic`, this becomes our `biome_name`.

This biome contains only one layer, but we still must specify it, in this case it's `None`.

But now that we chose the biome, we must know that file names that we chose previously do not match the one permitted for exotic planet, this must be fixed.

Rename `Plains_Terran_Puzzles` to `Valleys_Exotic_Puzzles`.

Also rename `ObjPl_Puzzle_Surface_Terran` to `ObjPl_Puzzle_Surface_Exotic`.

Our metadata.json file should look something like this:

```
{
    "schema_version": 2,
    "name": "ExampleMod",
    "mod_id": "ExampleMod",
    "author": "YOUR_NAME",
    "description": "An Example Mod",
    "version": "0.1.0",
    "sync": "serverclient",
    "integrator": {
        "item_list_entries": {
            "/Game/Items/ItemTypes/MasterItemList": {
```

(continues on next page)

(continued from previous page)

```
                "ItemTypes": [
                    "/Game/Mods/YOUR_NAME/TutorialMod/ExampleItem_IT"
                ]
            }
        },
        "biome_placement_modifiers": [
            {
                "planet_type": "Exotic",
                "biome_type": "Surface",
                "biome_name": "Valleys_Exotic",
                "layer_name": "None",
                "placements": [
                    "/Game/Mods/YOUR_NAME/TutorialMod/Valleys_Exotic_Puzzles"
                ]
            }
        ]
    }
}
```

Now cook the mod and verify you see objects spawning.