UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Toomas Treikelder

Developing User-Generated Content Distribution Systems for Blastronaut Game

Game Development

Supervisor: Jaanus Jaggo, MSc

Tartu 2022

Developing User-Generated Content Distribution Systems for Blastronaut Game

Abstract: Maintaining the user base for a video game is a challenging task. Prior research has shown that enabling players to contribute their own content can help with the longevity of a game. The main goal of this thesis is developing a solution to support User-Generated Content (UGC) distribution in the video game Blastronaut. The thesis specifically focuses on distribution of game modifications. Tools for game content creation and distribution in other games are examined for reference. Subsequently, a new solution is built on top of pre-existing game data management logic in the game. The added content distribution solution is designed to allow players to download, create and publish content packages for Blastronaut as seamlessly as possible. To achieve this, the game's data management logic is extended to support the concept of distinct content packages that can be installed, uninstalled, and shared. The Steam Workshop is used as a platform for the distribution of this content. An initial working solution is finished, and usability testing is conducted to gather feedback about the developed solution. It is concluded that the developed system is suitably intuitive to use but would need future work to make it production-ready. Potential issues with the solution are analysed, and concrete suggestions made for future development.

Keywords: user-generated content, game modifications, mods, game development, usability testing, Godot engine, Steam Workshop

CERCS: P170 Computer science, numerical analysis, systems, control

Mängijate poolt loodava sisu levitamise süsteemide arendamine mängule Blastronaut

Lühikokkuvõte: Arvutimängu olemasolevate mängijate hulga säilitamine on keeruline ülesanne. Varasemad uurimused on näidanud, et kui lubada mängijatel ise mängu jaoks uut sisu luua, võib mängu kestvus tõusta. Käesoleva töö eesmärgiks on lisada tugi mängijate poolt loodava sisu jagamiseks mängus Blastronaut. Töö keskendub spetsiifiliselt mängu modifikatsioonide jagamise võimalustele. Näidisteks uuritakse teistes mängudes esinevaid lahendusi. Mängus olemasoleva andmehaldusloogika juurde ehitatakse uus tarkvaralahendus loodud sisu jagamiseks. Lisatud sisuloome ning jagamise lahendus võimaldab Blastronaut-i mängijatel laadida alla ning luua uusi sisupakke ning jagada loodud sisu teiste mängijatega. Loodava sisu jagamiseks kasutatakse platvormi Steam Workshop. Töö käigus luuakse esialgne töötav lahendus ning viiakse kasutajatega läbi lahenduse testimine tagasiside kogumiseks. Saadud tagasiside põhjal tehakse järeldus, et arendatud lahendus on piisavalt intuitiivne, aga vajab tulevasi arendusi, et muuta see sobilikuks kasutajatele avalikuks tegemiseks. Lahenduse puudujääke analüüsitakse ning pakutakse välja konkreetsed soovitused edasisteks arendusteks.

Võtmesõnad: kasutajate poolt loodav sisu, mängude modifikatsioonid, mänguarendus, Godot, Steam Workshop

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1 Introduction

For a small game development studio, maintaining a game is not a simple task. Distributing video games is becoming ever simpler with the emergence of digital distribution platforms like Steam[1]. Therefore, video games can launch having been developed by very small teams. At times games can have a substantial scope with just a single person developing and maintaining the project. Under these circumstances, creating new game content like assets and gameplay elements to keep the player base engaged can be very difficult due to the large amount of work required. Enabling players to create and distribute their own content within the game for others to play can be a viable way to boost the longevity of a game [1].

Blastronaut[2] is a 2D, side-scrolling exploration game set in a procedurally generated game world (Figure 1). The player can explore an alien planet, digging and mining resources and selling them to purchase new equipment to dig deeper into the planet's surface. The game world is randomly generated and composed of blocks that can be destroyed, allowing the player to dig tunnels and explore the underground world of the planet. Various environments (biomes) exist, which can contain different block structures, both generated and pre-made. Each environment contains different enemies, along with other dangers. Blastronaut is developed in the Godot Engine[3] version 3.3.2. While the game has had various developers working on smaller parts as part of other thesis projects, a single person primarily develops the game. This includes design and implementation of gameplay logic as well as game content creation. At the time of writing Blastronaut has not yet been released to the public.

[1] https://store.steampowered.com/
[2] https://store.steampowered.com/app/1392650/BLASTRONAUT/
[3] https://godotengine.org/

Figure 1. Screenshot from the gameplay of Blastronaut.

In this thesis a User-Generated Content (UGC) distribution solution is developed on top of the pre-existing dynamic game data framework and its custom editing interface (the editor). The developed solution focuses on creating and distributing in-game content made by players, referred to as game modifications or mods. For video games, mods are a large part of UGC. Chapter 2 describes UGC in the context of Blastronaut and gives an overview of the Steam Workshop. Additionally, similar solutions in other games are detailed. The pre-existing functionalities and the basics of the game's data structures are explained in detail in Chapter 3. Chapter 4 describes the features implemented as part of this thesis and the main technical details of that implementation. Chapter 5 describes the process and results of conducted usability testing. Chapter 6 details the results of the development process and possible future developments. In Chapter 7, conclusions are formed. The Appendix includes the following items:

- **Appendix I** – Materials and feedback of the conducted usability testing;
- **Appendix II –** Link to the private Gitlab repository containing game code;
- **Appendix III –** The license.

1.1 Motivation

Players creating and distributing content for games is not a new concept. As time has passed, developer support for officially creating in-game content has become much more common.

Part of the reason is likely to be the growth of the gaming industry as a whole - the scale of games has increased. Since gamers are generally difficult to please, it has become more costly and difficult to create a successful game [2]. Because long-time players of a game have been found to be more likely to keep buying games from the same developer, keeping the existing player base of a game has increased in importance. Due to the inherent cost of maintaining and updating a game to increase longevity of a game there is a difficult trade-off for the developer: to maintain an existing game or create a new one [2]. Creating game modifications has proven to be a way where the community of the game can help increase longevity by adding new game content and changing existing content.

Providing official support for player content creation tools requires additional development time and effort to create the tools and documentation needed. However, in the case of Blastronaut an in-game tool for creating and managing game content locally had already been developed, referred to as the editor. Considering the advantages of including built-in content creation support in a game it was decided to build functionality on top of the editor to allow distributing created content.

Blastronaut is planned to be distributed on the digital content distribution platform Steam. Developed by Valve Corporation, Steam is one of the most popular distribution platforms for video games in the modern time. It has popularized the concept of digital game distribution since its release in 2003 [1]. Integrating content creation solutions with game distribution platforms can potentially provide content creators with stronger incentives to use the tools [3]. Due to Blastronaut already using Steam as a distribution platform, it was decided that the content made by players could be distributed using the Steam Workshop[4]. More detail about the Steam Workshop is provided in Chapter 2.2.

[4] https://steamcommunity.com/workshop/

1.2 Goals

Given the existing support for dynamically loaded game content and a local editor interface, the decision was made to build additional functionality on top of this pre-existing logic. To achieve this, the thesis has three main goals:

1. Examine and describe solutions for User-Generated Content creation and distribution in other games for reference.
2. Implement a working solution for User-Generated Content creation and distribution in Blastronaut.
3. Conduct usability testing for the developed solution and analyse gathered results.

2 Background

This chapter briefly introduces two important topics of this thesis – User-Generated Content (UGC) and the Steam Workshop. Additionally, UGC creation and distribution solutions in other games are described.

2.1 User-Generated Content

The term User-Generated Content is a wide concept, covering diverse types of digital content created by the end users of a product. In video games, UGC often refers to content players can create within the game. This includes images, videos, and other media of the game. However, more importantly UGC in games also includes game modifications and in-game content [4]. In this thesis, specifically the creation and distribution of game modifications will be focused on.

The term for creating game modifications is referred to as "modding", from the word "modify". However, in Blastronaut instead of the term mod, another common term is used – content package. In the context of this thesis, these terms can be considered interchangeable. The main reason for this alternative term is to better match terminology used during the implementation of the distribution solution in Blastronaut.

2.2 The Steam Workshop

The Steam Workshop (referred to as workshop in this thesis) is a content distribution platform publicly available as part of the Steam platform since 2011 [1]. It provides an API for uploading and downloading additional game content and server space for hosting the content. Workshop support can be integrated into any game that is being distributed through the Steam platform by leveraging the provided API. At the time of writing, over 1700 games are listed as having workshop support on the workshop home page[5]. The home page shows an index of all games with workshop support (Figure 2). However, each game that has enabled workshop support also has its own workshop page.

[5] https://steamcommunity.com/workshop

Figure 2. Home page of the Steam Workshop.

When workshop support is added to a game, users can be allowed to upload any sort of game content to be hosted directly in Steam. An uploaded piece of content is called a workshop item. The exact implementation of what content can be uploaded is up to the developers of the game. Users can browse workshop items added by other players and subscribe to items in the workshop. An example of the page of a workshop item for the game Don't Starve[6] can be seen in Figure 3. The Steam client manages all data uploading and downloading operations similarly to how games are downloaded from Steam. This means that the developers are free to focus on higher level functionalities of the integration.

[6] https://www.klei.com/games/dont-starve

Figure 3. Example of a workshop item in the workshop of the game Don't Starve. When pressing the Subscribe button, Steam will download this item.

Other centralized platforms exist for the distribution of game modifications. For example, Nexus Mods[7] is a third-party service for distributing game modifications. However, the Steam Workshop stands out due to being integrated into Steam – the same platform that is being used to distribute the game. This could increase incentive for players to use the provided content creation tools [3].

2.2.1 Workshop types

Game developers have the option to pick from two types of workshop support - *ready-to-use workshop* and *curated workshop* [5]. The *ready-to-use* workshop model allows players to upload content without any additional checks by the developer. While there is a common license agreement that every uploader must accept, no verification of the uploaded content is performed. It is suggested to spend more development time on tools that would guarantee validity of uploaded content on the game's side [5]. Uploaded content is readily available for

[7] https://www.nexusmods.com/

other players of the game. This type of workshop is often used by developers who do not have the resources to approve each upload. All games detailed in chapter 2.3 use the *ready-to-use* workshop model.

The curated workshop model enables developers to have tighter control over the content uploaded to the game's workshop. Each item added requires approval by the developer. Submitted workshop items are first put through a voting process. Subsequently, popular items can be added to the game via a game update. This model requires more work from developers and is more often used by larger studios. An example of a game with curated workshop support is Team Fortress 2[8], developed by Valve Corporation.

2.2.2 Workshop integration

To add workshop integration for a game, the ISteamUGC interface[9] can be used, provided by Valve Corporation. The API provides methods for a variety of operations. These include downloading workshop items the player has subscribed to, uploading items, changing item metadata, uploading preview images and more. Whenever a player subscribes to an item in the workshop, the Steam client automatically downloads it. Workshop item content is downloaded into directories specified by the Steam client, which games can then access.

Each uploaded item in the workshop has a unique ID. This ID is called the *Subscribed File ID* in the workshop documentation[10]. In this thesis, the term file ID or workshop item ID will be used to refer to this value. The workshop page of any uploaded content can be accessed from a link containing the corresponding file ID. Specific implementation details of how the ISteamUGC interface is used in Blastronaut will be covered in Chapter 4.

2.2.3 The GodotSteam library

To integrate a game developed in the Godot engine with the workshop, a third-party library exists, named GodotSteam[11]. The GodotSteam library provides abstraction to access most Steam API functionalities using Godot's GDScript language.

At the core of all GodotSteam functions is a class named Steam. When an instance of this class is created, any supported Steam API calls can be made by calling methods on this instance.

[8] https://www.teamfortress.com/
[9] https://partner.steamgames.com/doc/api/ISteamUGC
[10] https://partner.steamgames.com/doc/api/ISteamRemoteStorage#PublishedFileId_t
[11] https://gramps.github.io/GodotSteam/

The methods cover a large part of the entire range of API functionalities provided by Steam, including the ISteamUGC interface methods used for workshop integration.

In Blastronaut, GodotSteam was already in use before this thesis. Beforehand, it was mainly used to enable cooperative multiplayer through the Steam API. Thus, workshop integration was also implemented using the GodotSteam library and the pre-existing Steam class instance.

2.3 Existing solutions

Many existing games have implemented official functionality to allow players to create or modify game content. Some official solutions can limit distributable content to just visuals, like the solution in Minecraft[12] for creating character skins[13]. In other cases, complicated solutions are created, allowing the implementation of complex gameplay-altering modifications. Examples of this include large-scale games like The Elder Scrolls V: Skyrim[14] or ARMA 3[15]. Many independent games fall between these two extremes, with varying levels of complexity. However, it is also possible to find independent games with comparatively small development teams implementing full support for complex mods. This includes everything from creating custom levels up to allowing for scripting custom logic in the game.

Due to Blastronaut being an independent 2D game developed mainly by one person, the existing solutions that were examined in detail were picked to be for smaller games of comparable size and planned scope. The focus was on games which had also implemented Steam Workshop support.

2.3.1 Teardown - Tuxedo Labs

Teardown[16] is a first-person action strategy game developed by Tuxedo Labs, released as an early access[17] game in October 2020. The game is built on top of a highly detailed custom physics engine. It focuses on the player's ability to alter the game's environment (levels) as preparation and tasks the players with completing various missions in these altered levels - from collecting secured items as quickly as possible to avoiding security robots. A screenshot from the game can be seen in Figure 4.

[12] https://www.minecraft.net/en-us
[13] https://www.minecraftskins.com/
[14] https://elderscrolls.bethesda.net/en/skyrim
[15] https://arma3.com/
[16] https://teardowngame.com/
[17] https://partner.steamgames.com/doc/store/earlyaccess

Figure 4. Screenshot from the gameplay of Teardown.

Enabling mod support was a highly requested feature of the game, as many players wished to use the detailed physics engine for activities other than what the developers had intended in the core gameplay. The core game is partly a puzzle game, being about finding more efficient ways to solve each mission. However, many players just wanted to use the custom physics engine for other purposes like demolishing buildings or finding and creating new game modes to enjoy.

At the time of writing Teardown has extensive modding support, with the ability to create both levels and in-game equipment and vehicles to be used by the players. A dedicated menu titled “Mods” was added to allow players to enable and disable their downloaded mods as well as configure any parameters of the mods (Figure 5). New levels can be created with the provided built-in level editor (Figure 6) which can be opened directly from a menu titled “Mods”.

Figure 5. Menu for managing mods in Teardown.

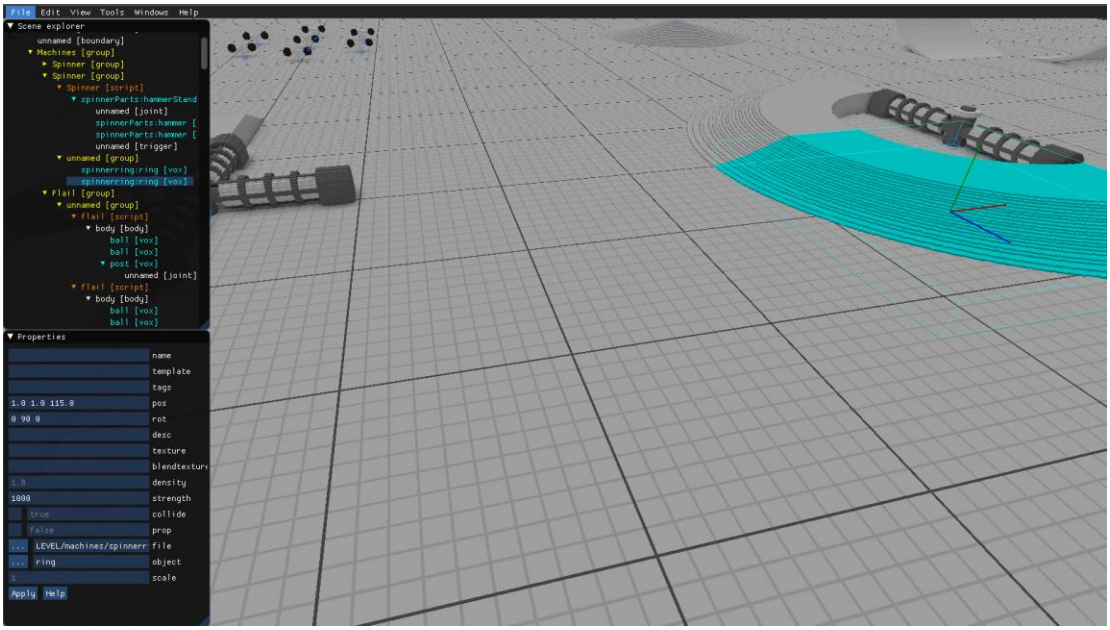

Figure 6. The built-in level editor in Teardown.

Modifications changing the functionality of the game can be created by the means of Lua[18] scripting. Lua is a scripting language often used in game modding. When integrated with the game by developers, such scripting language integration can allow players to leverage the

[18] https://www.lua.org/

game's scripting API to create new modifications. This way more varied modifications to the in-game logic can be created compared to cases where mods are limited to changing a fixed number of parameters of game objects that the developers have determined. At the time of writing any custom logic for mods in Teardown are intended to be created via external tools. Any text editor or IDE supporting the Lua language can be used for scripting. A free application called MagicaVoxel[19] is suggested for creating in-game object models. Extensive documentation[20] is provided by the developers to give information on how the scripting API can be used.

The game has full workshop integration, with players able to subscribe to workshop mods via a built-in "Mods" menu. When a button labelled "Manage subscribed" is pressed, an overlay is opened with Teardown's workshop page (Figure 7) open in it. If a player subscribes to an item in this window, then the item will immediately be downloaded and appear in the "Mods" menu. The mod can then be enabled and will appear in the game. Teardown has an active mod creation community. Having been released in early access in 2020 and as a full release only on 21. April 2022 it already has over 2500 entries published to the Steam Workshop by its players despite having only been released out of early access very recently.

[19] https://ephtracy.github.io/
[20] https://www.teardowngame.com/modding/

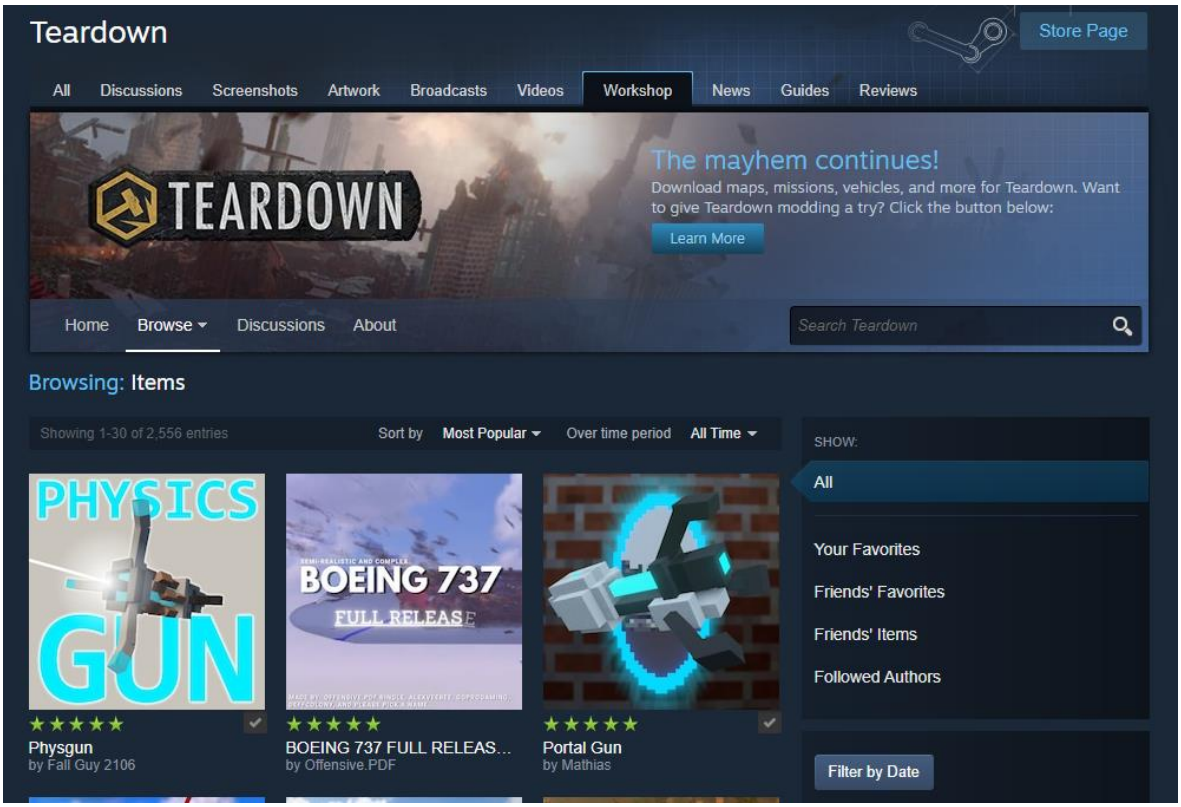

Figure 7. The Steam Workshop page of Teardown.

2.3.2 Don’t Starve - Klei Entertainment

Don’t Starve is a 2D survival adventure game developed by Klei Entertainment and released in April 2013. Players are placed into a hostile world containing various dangers and are tasked to survive. Players gather resources to craft items and structures to increase their chances of survival. See Figure 8 for a screenshot from the game.

Figure 8. Screenshot from the gameplay of Don't Starve.

The game offers full mod support, with players being able to create everything from small tweaks to the game's visuals or sound to complete overhauls of how the game looks and plays. Popular mods even include ones fixing various bugs in the main game. Mod tools for Don't Starve are available as a separate download from the "tools" section in the Steam library and are available to everyone who owns the game.

The Don't Starve Mod Tools installation includes the following external tools:

- A custom uploader utility for publishing a mod.
- A utility for editing animations (Spriter[21]).
- A utility for editing audio (FMOD Designer[22]).
- A utility for editing pieces that make up the game world (Tiled[23]).
- A text editor for editing Lua scripts (Sublime Text[24]).

An in-game menu titled "Mods" allows players to enable and disable any content that they have downloaded (Figure 9) A button labelled "More Mods" opens a separate browser window

[21] https://brashmonkey.com/
[22] https://www.fmod.com/
[23] https://www.mapeditor.org/
[24] https://www.sublimetext.com/

showing the game's workshop page outside of the game. Mod creation and publishing is not intended to be done from within the game, but rather via the external utilities mentioned above.

Figure 9. The menu for managing mods in Don't Starve.

Integration with the workshop allows Don't Starve players to subscribe to mods and have them automatically downloaded by the Steam client. While the mods are not downloaded immediately and require re-launching the game this still provides for a seamless experience for players. The workshop support for the game was released in 2013 [6]. At the time of writing over 3000 items exist for the game in the workshop[25].

[25] https://steamcommunity.com/app/219740/workshop/

Figure 10. The Steam Workshop page for Don't Starve.

2.3.3 Terraria - Re-Logic

Terraria[26] is a 2D action-adventure game developed by Re-Logic and released in May 2011. Players explore a procedurally generated world and fight enemies, trying to survive and gather better equipment. The world is composed of blocks, allowing the player to dig underground in search of resources and build structures in-game. See a screenshot of Terraria in Figure 11.

[26] https://terraria.org/

Figure 11. Screenshot from the gameplay of Terraria.

Workshop support was added to Terraria in July 2021. Even though this was over 10 years after the release of the game, the workshop has been extensively used with over 54 000 items uploaded at the time of writing. There is an in-game menu titled “Mods” that allows uploading and downloading user generated content in the workshop (Figure 12). However, not all publishable content can be created in-game, as resource packs for Terraria must be created with external tools. What is more, the official workshop integration is limited to sharing in-game level saves and resource packs modifying game visuals and sound. An example of the Terraria workshop can be seen in Figure 13.

For more sophisticated mods that add items or change the logic of how the game functions an external tool is used, called tModLoader[27]. This tool was originally created by the community of Terraria players in 2015 as an unofficial open source modding tool and later added as a separate download in the Steam store in 2020. It was officially recognized[28] in 2021 by Re-Logic.

[27] https://github.com/tModLoader/tModLoader
[28] https://forums.terraria.org/index.php?threads/unleash-your-creativity-terraria-steam-workshop-support-launches-today.104084/

Figure 12. The menu for managing mods in *Terraria*.

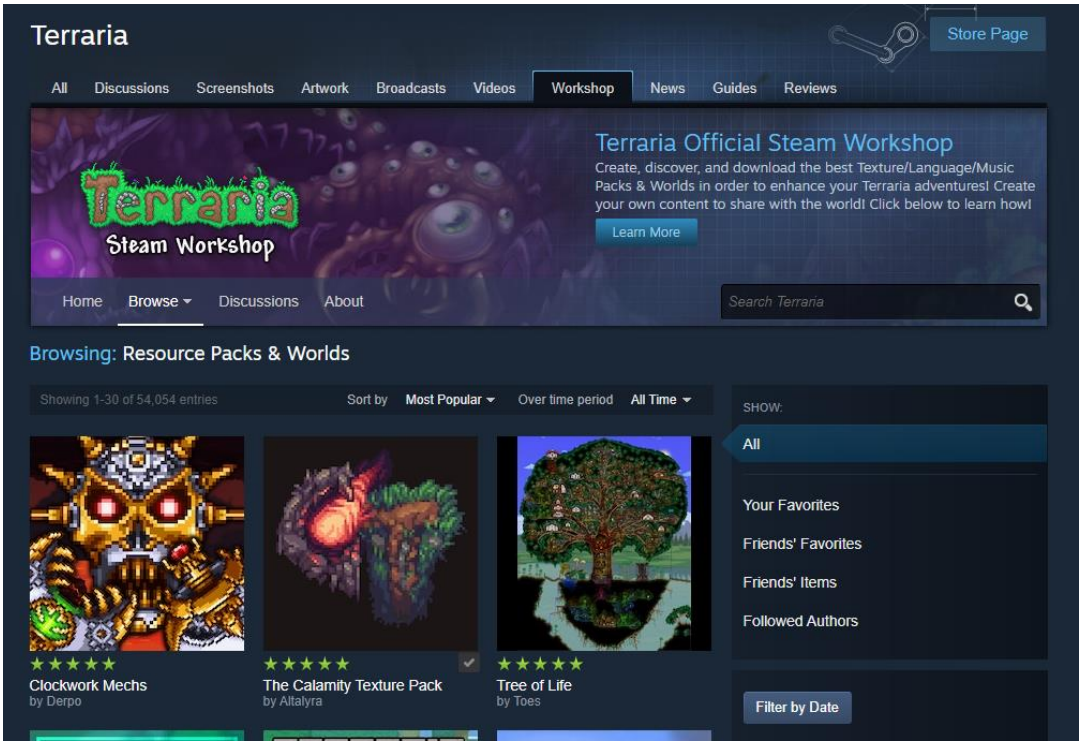

Figure 13. The Steam Workshop page for Terraria.

3 Pre-existing game data solutions in Blastronaut

The User-Generated Content distribution functionality developed in this thesis was implemented as an addition to already existing game content loading logic. This Chapter details how the game's content is structured and how data loading worked before these developments. Additionally, the pre-existing editor interface is described.

3.1 Godot engine

Blastronaut is developed in the Godot Engine version 3.3.2. Godot is a free and open-source game engine allowing for development of 2D and 3D video games. One advantage of using the Godot engine over various other engines like Unity[29] is its highly permissive MIT license. While using open-source software can have both benefits and drawbacks [7], having the engine code available for the developers can simplify development.

3.2 Game content

At the start of developing this thesis there was already a solution implemented in Blastronaut allowing in-game objects to be defined at runtime. Instead of defining content in the Godot project, most of the game content is loaded from external JSON[30] files. Loaded data includes the game's visuals, audio, and many in-game parameters that affect gameplay. This differs from other aspects of the game, which are fully defined in game code. Such elements include for example the core logic for in-game object behaviour, physics, and player input handling.

Each distinct type of game content in Blastronaut has a defined data structure. Internally, each of these content types is a GDScript[31] class. These data classes are called Systems in the context of Blastronaut. The term System will also be used to refer to the data classes in this thesis. Each System has a collection of properties (Strings, Arrays, Booleans etc.) and additional logic to return and manage these properties at runtime. See Figure 14 on page 25 for an example of an object's JSON file.

Some examples of Systems in Blastronaut are:

- **Spritesheet** - Describes an imported image file. Includes a file path to the actual image file in the filesystem. It has methods to retrieve texture information from the image file.

[29] https://unity.com/
[30] https://www.json.org/json-en.html
[31] https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html

- **Image** - Describes an area on a Spritesheet. Includes a reference to a Spritesheet object and has properties for defining a rectangular area cut out from that Spritesheet as well as a point to use as an origin point of the Image.
- **Sound** - Describes one or more imported audio files. Includes an array of file paths to the audio files in the filesystem. A randomly selected audio clip is played when the Sound object is triggered in-game. Properties include ones for defining pitch and volume variation and timing.
- **Block** - Describes a terrain block used in-game. Includes a reference to an Image file to use as the visual for the block. Additional properties include internal data values, items dropped when the block is broken etc.
- **Template** - Describes a collection of arranged Block objects in a grid. It is designed for pre-creating block structures to later be placed in the world as parts of larger structures. Includes properties that contain the collection of Block objects.
- **Biome** – Describes a randomly generated area in-game. Different areas can have different Block and Template objects generating. Additionally, different enemies can appear in each area. Properties specify game terrain generation parameters, as well as lists of structures that are generated in each area.

In total there are 21 distinct Systems in Blastronaut with plans to add more when needed by the main developer. These Systems describe everything from the graphics, sound and building blocks of the world to more complex structures like the Biome System. A full list of all the Systems in Blastronaut can be found in Appendix I – Usability Testing Materials in the document "Usability_testing_modding_reference.odt". All JSON files for each System are read upon launching the game and converted into corresponding System instances, referred to as objects in this thesis. The objects are then saved into Dictionary[32] type variables, referred to as collections. Each System has its own collection containing its objects. The collections are globally accessible from other parts of the game code.

In addition to these Systems smaller pre-defined structures exist, called Subsystems. While these structures are also GDScript classes, they differ from Systems in that they are not loaded from their own JSON files and do not have a separate collection. Instead, they are only used inside objects as values of some properties. See an example of a Subsystem in Figure 14 B.

[32] https://docs.godotengine.org/en/stable/classes/class_dictionary.html

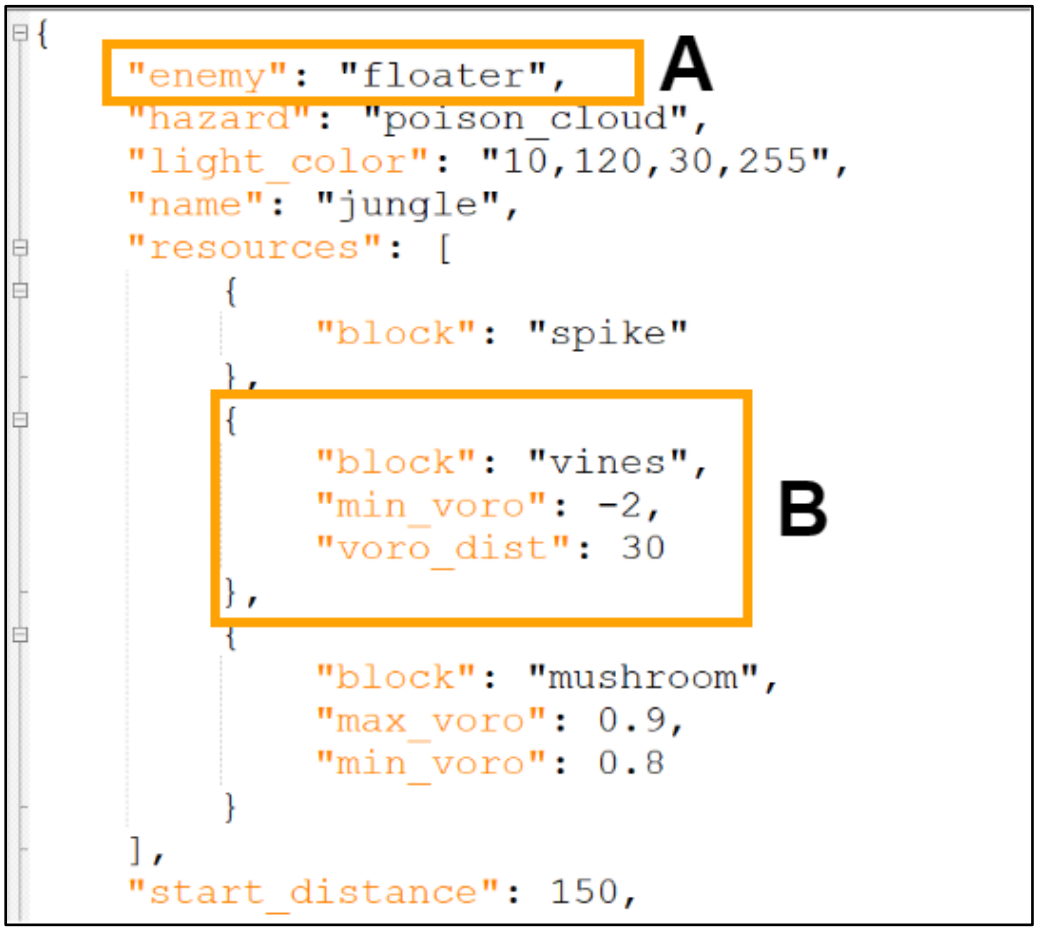

Figure 14. A Biome object's JSON file. Marked with red - a String type property (A) and a property containing a Subsystem in an array (B).

3.3 The editor

For easier editing of object JSON files, a custom GUI had already been added beforehand - the editor. A screenshot of the editor displaying an Image object can be seen in Figure 15. The editor contains four primary areas of interest:

- **System selection tabs** for selecting a System for which the user wants to create, modify, or delete objects (Figure 15 A).
- **File panel** for selecting specific objects of the selected System (Figure 15 B). Each element in the list corresponds to one object. Three buttons above the file panel allow the user to create new objects and duplicate or delete existing objects.
- **Object inspector panel** for changing any properties of the object selected in the file panel (Figure 15 C).

- **Preview window**, which shows a visual preview of the selected object if relevant (Figure 15 D).

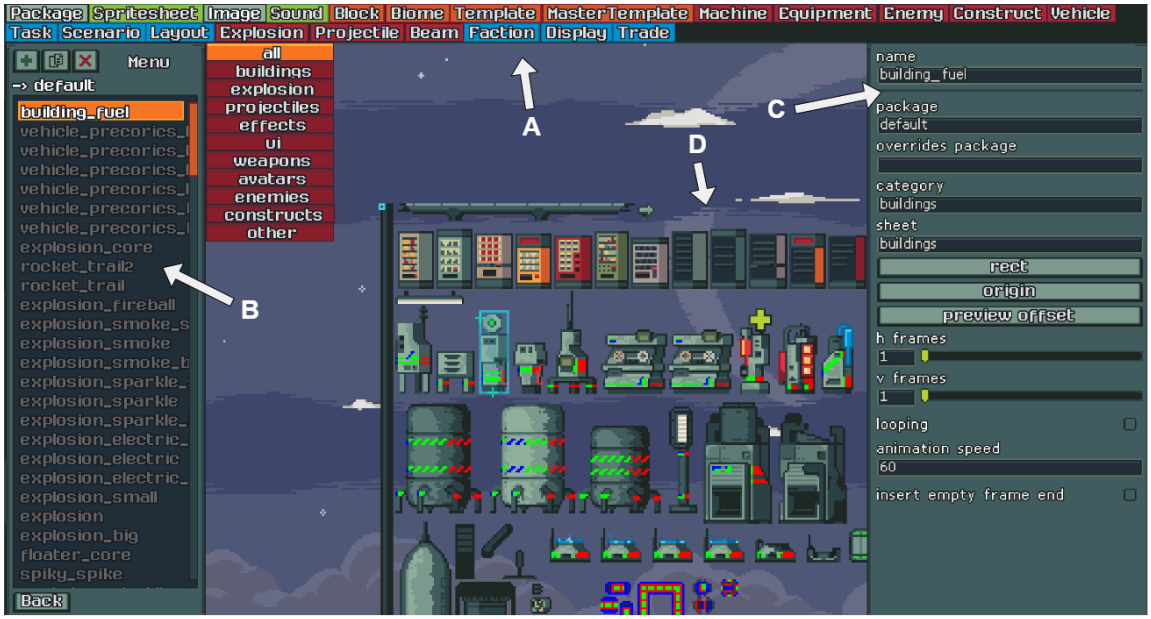

Figure 15. Blastronaut Editor window displaying an Image object. The four main areas of the Editor window are marked with letters A-D.

3.4 Relations between objects

During prior work by the main developer of the game the decision was made to keep all objects saved in a human-readable JSON format. The purpose of this decision was to make editing the files easier with external tools if needed. Each System has a human-readable *name* property to make identifying objects easier. This property is a String which acts as the unique identifier of the object. After loading objects from files, the *name* properties were used as keys in the collections containing the objects.

The object *name* properties can also be used to create relationships between objects. This allows developers to define more complex Systems which include properties referencing other objects. As an example, an Equipment object defines a piece of equipment the player can use in the game - for example, a weapon or other utility. It has the following references to objects from other Systems (System names are marked in bold):

- A reference to an **Image** object that specifies how the piece of equipment looks.
- References to multiple **Sound** objects that specify how using the equipment sounds. Multiple properties are defined, with separate references for *fire_sound* and *loop_sound* to be used in different gameplay cases.

- A reference to a **Projectile** or **Beam** object that will be fired out of the equipment on use - for example explosive gel for mining or smoke particles for a jet pack.

Each of the mentioned references is a String type property containing the name of another game object. See an example of an Image object JSON file in Figure 16.

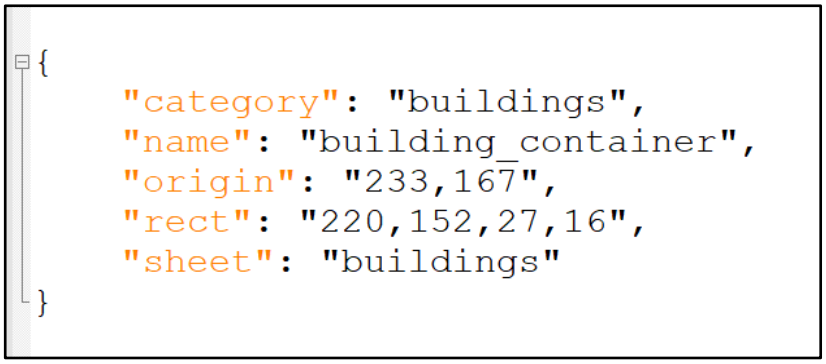

Figure 16. Example of an Image JSON file. The property *sheet* is a reference to a Spritesheet object with the name "buildings".

After loading a JSON file, referenced object names were used to retrieve the actual objects from their Systems' collections. Godot export hints[33] were used to specify which System's object a property is referencing. For example, if a property is meant to reference an Spritesheet object, then it would be marked in code with an export hint "Spritesheet" (see Figure 17).

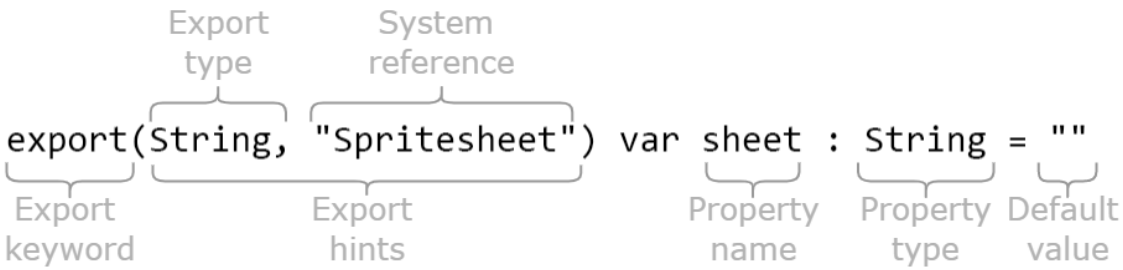

Figure 17. Code example of an exported String type property. Using export hints, the property is specified to be referencing a Spritesheet object.

Using the export hint, the correct collection can be picked to retrieve an object. The export hints also allow the editor UI to display a list of objects that a property is allowed to reference. An example of referencing another System's object in the editor can be seen in Figure 18.

[33] https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_exports.html

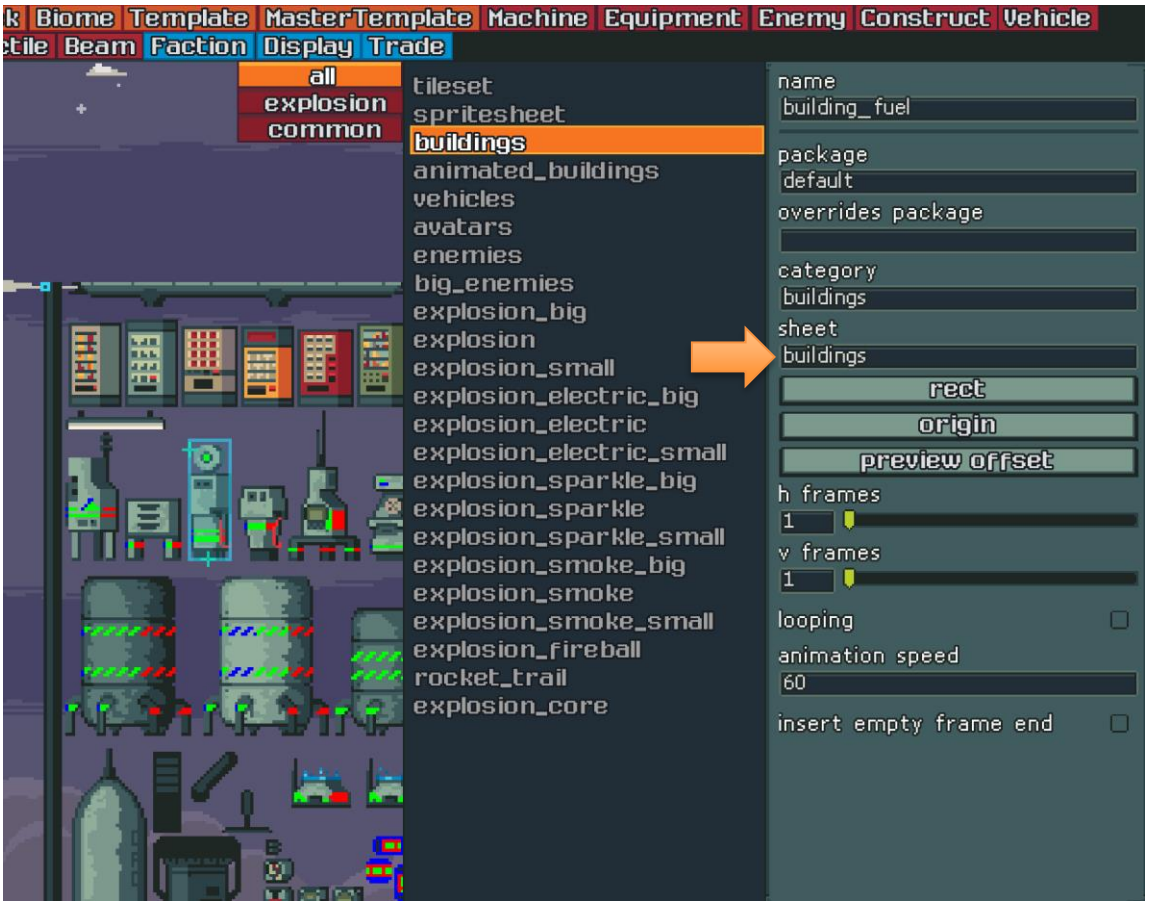

Figure 18. Editor UI panel for picking the *sheet* property of an Image object. The editor shows a list of Spritesheet objects in a fold-out panel.

Property export hints can also allow a developer to automatically pick the correct UI elements to display for each property. For example, specifying a String property with multiple String type values as a property hint would display a dropdown menu with the hint values as options (Figure 19). In this way the entire editor interface can be data-driven, allowing developers to define new Systems without worrying about adding support for them in the editor.

```
                Export  Empty          Dropdown
                type    hint           options
              ┌──┴──┐ ┌──┴──┐ ┌──────────┴──────────┐
export(String, "", "construct", "research", "store") var mode : String = ""
└──┬──┘                └──────────┬──────────┘          └─┬─┘   └──┬──┘ └──┬──┘
Export                          Export                  Property  Property Default
keyword                         hints                   name      type     value
```

Figure 19. Code example of an exported String type property. An additional export hint specifies options for a dropdown menu.

3.5 Game data paths

Before the development of this thesis Blastronaut loaded data from two file system locations – the game's resource path and the user path (Figure 20). The resource and user paths are automatically resolved[34] in the Godot engine to point to file system directories depending on where the game is installed. By default, the resource path refers to the game installation directory and the user path to an *AppData* folder subdirectory (when using Windows).

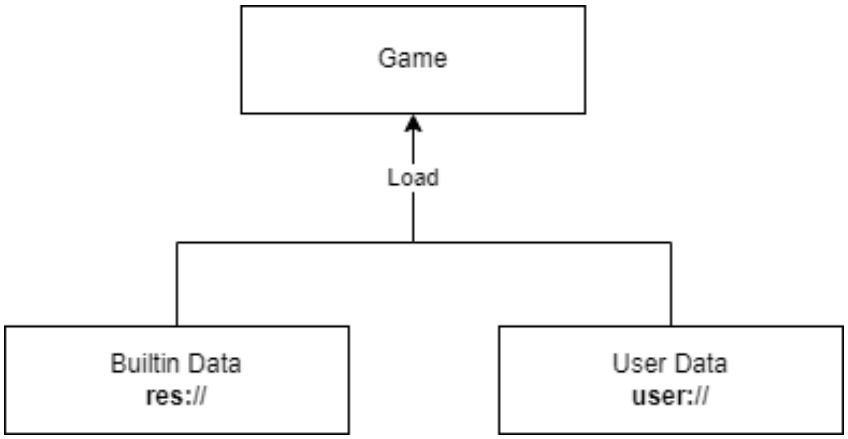

Figure 20. Blastronaut file system directories before the developments of this thesis - resource and user paths.

A JSON file in the resource directory contains metadata for Systems, including *path_user* and *path_resource* properties for each System. The properties contain exact resource and user path subdirectories for each System in the game. This file is loaded before all other data. For all Systems, the game checks the corresponding resource and user paths and loads any object JSON files found.

In addition to the *name* property mentioned above, each System also has a Boolean property named *external*. This property is not loaded from the object's JSON file but is rather set by the game for each object at load time. The value of the property depends on where on the file system the object has been loaded from. Any objects loaded from the user path would have their *external* property set to *true*, while resource path objects would get a value of *false*. In this way the game data management logic can distinguish between the two types of content. As an example, this property is used to prevent players from deleting built-in content in the editor. Modification of resource path objects is also disabled, with the UI elements for editing the object's properties deactivated.

[34] https://docs.godotengine.org/en/stable/tutorials/io/data_paths.html

Upon exporting the game as an executable, all content in the resource directory gets packaged into the game binaries. All built-in content shipping with the game is contained within the resource path subdirectories and those files are not directly editable to any end user downloading the game.

4 Implementation in Blastronaut

For the implementation, three sub-goals were set. The implemented solution would need to enable players to perform three activities:

1) **download** content packages from the workshop;
2) **create** content packages;
3) **publish** content packages to the workshop.

An overarching core requirement was that each of these three activities would have to be available from within the game, without the user needing to close the game and use external tools. When discussing functionality to support these activities it was agreed that necessary modifications would need to be made in the whole logic of how game data is loaded into the game, as well as any UI and usability changes required by the workflows necessary.

Implementation of user-generated content distribution functionality in Blastronaut required extensive changes to some parts of the game code base. The most impacted was pre-existing data loading and handling logic. Additional work was done on the editor UI to allow users to interact with the developed solution. All implementation was done as an extension of the pre-existing in-game editor and game data framework detailed in Chapter 3.

The work required to implement the user-generated content distribution solution did not follow the three implementation goals in sequential order. Based on the logical order of development activities the performed development steps were instead:

1. Adding supporting features for the content package functionality.
2. Adding editor content package functionality.
3. Adding functionality for the management of content packages.
4. Integrating with the Steam Workshop.

During development, these steps needed to be approached in the order listed, as the implementation of each was dependent on the previous steps being implemented.

4.1 Supporting features for the content package functionality

In the initial discussions, a few areas of the pre-existing system were detected which were deemed necessary to rework. These changes were necessary for the later developments, which in some cases would otherwise have been exceedingly difficult or impossible. While smaller

fixes were done as the need was encountered, two larger developments can be brought out as important supporting functionality:

1. External file loading for asset files (images and sounds).
2. Propagation of object name changes.

Out of these two, the propagation of object name changes proved to be an unexpectedly time-consuming problem to solve. A large portion of development time of this entire thesis focused on improving and debugging this system specifically.

4.1.1 External file loading for assets

In the initial game data management logic external assets such as image and sound files were defined as absolute paths to files in the game's resource and user directories (*res://* and *user://*). To allow adding new files to any new content being created by the player, this solution needed to be reworked. Firstly, using absolute paths would not work for distributable content packages. When a package is downloaded by another player, their package contents would likely not be in the same location on their file system as on the package creator's file system. Secondly, the existing system would have made it cumbersome for players to add their own custom image or sound files. It was deemed unacceptable to force players to manually copy their files from the file system to the mod's directory.

To make adding asset file properties to Systems easier, support was added for a property export hint[35] "FILE". This hint can be used with String type properties (Figure 21). When the hint is added, the String property is known to contain a path to a file in the file system. An additional property hint is defined containing a comma-separated list of allowed extensions.

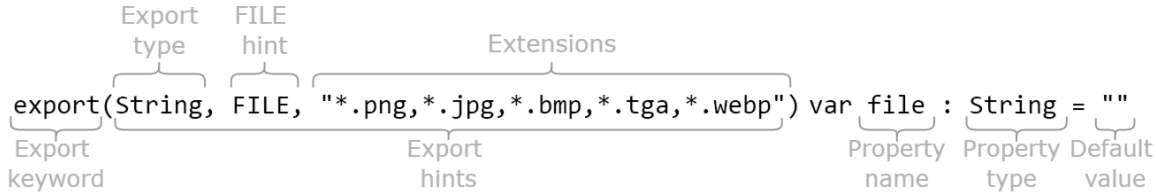

Figure 21. Code example of an exported String type property specifying a file path. An additional export hint specifies allowed extensions.

Using the export hint, it is then possible to determine in the editor UI when a String field should be treated as a file path. Changes were made in the components handling the UI display for

[35] https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_exports.html

game object properties. As a result, the editor displays a corresponding UI element to edit a file path property (Figure 22-A). Upon pressing a button next to the text box, a file browser dialog[36] is opened (Figure 22-B). The additional export hint containing extensions is then used to pass allowed extension information to the file browser dialog.

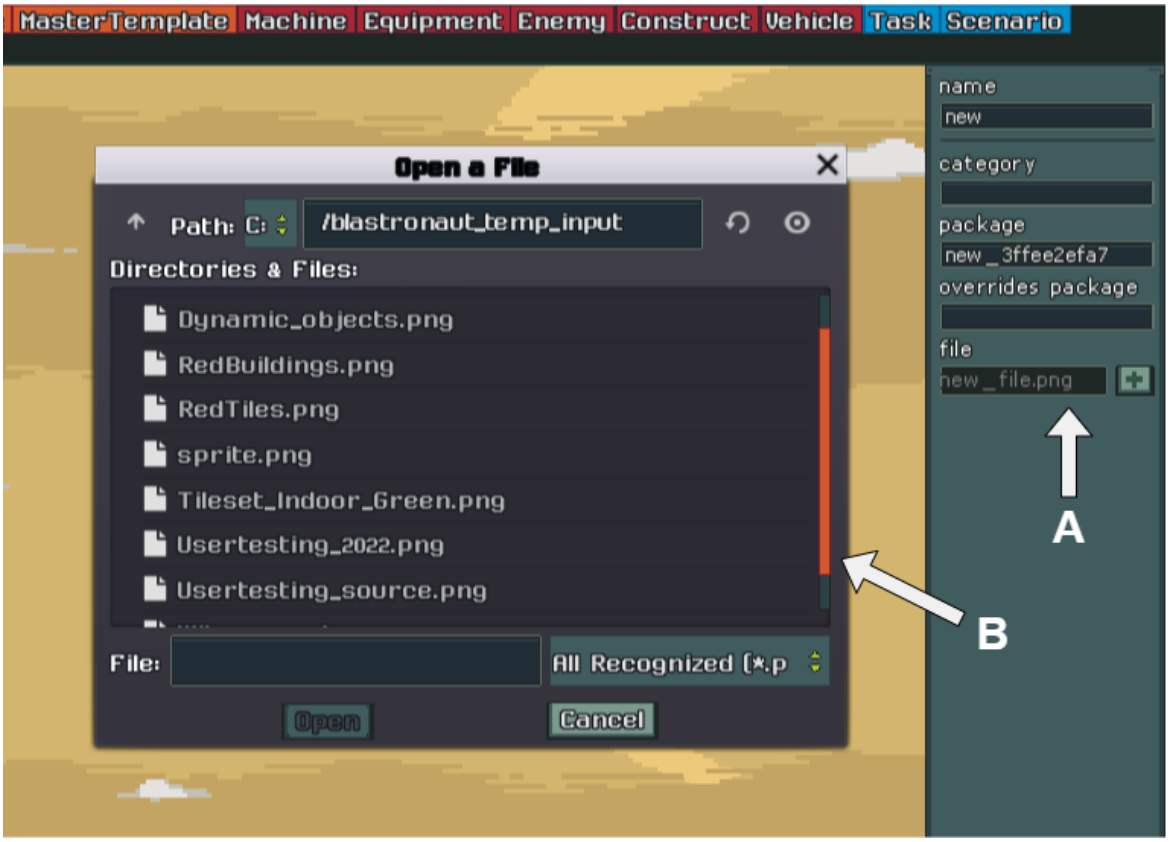

Figure 22. Editor external file property (A) and file browser dialog (B).

The practice of using export hints was picked due to it being in line with Godot design practices regarding property exports. Similarly, it followed the conventions already in place before these additions, mentioned in Chapter 3.4.

Background management of asset files was implemented to keep all game-related data confined within game directories. This included:

- Logic to make a copy of the asset file selected in the custom inspector panel into the same user path subdirectory with the object JSON file. The object's file path property is then set to reference the location of the copy relative to the object.

[36] https://docs.godotengine.org/en/stable/classes/class_filedialog.html

- Logic to rename the asset file to match the name of the object to avoid filename conflicts with other asset files in the same directory. The asset file name is composed of both the object name and the property name in case an object has multiple different file properties (Figure 23). This is done on every change of the object's name to keep the asset file name matched with the object it belongs to.

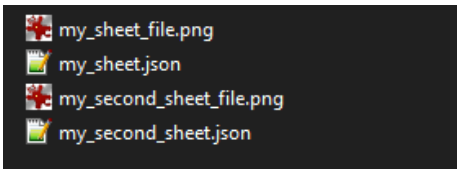

Figure 23. Spritesheet object JSON and asset files

- Logic to delete the asset file upon deletion of the game object.
- Logic to duplicate the file upon duplication of the game object.

In case of audio files attached to Sound objects, additional work was needed. These objects have an Array type property containing the paths of multiple audio files instead of a single file. In this case an Array of Strings can be defined in the class definition with the same FILE hint and additional extensions hint as above (see Figure 24). Additional file management logic was required for when elements are removed from the array, which causes an asset file to be deleted. The position of each element in the Array property was also included in the asset file name to guarantee a unique file name. See Figure 25 for an example.

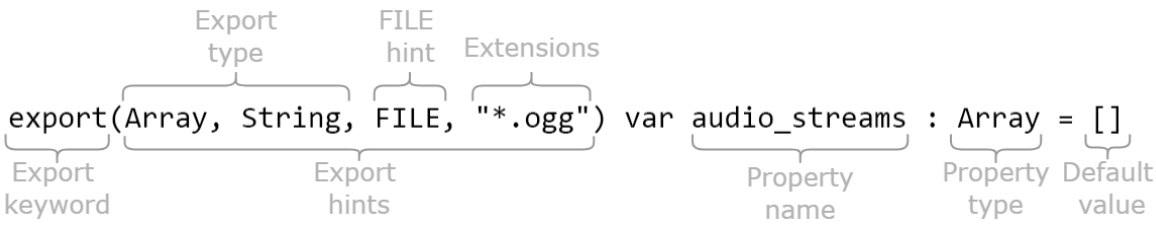

Figure 24. Code example of an exported String Array type property containing file paths. An additional export hint specifies allowed extensions.

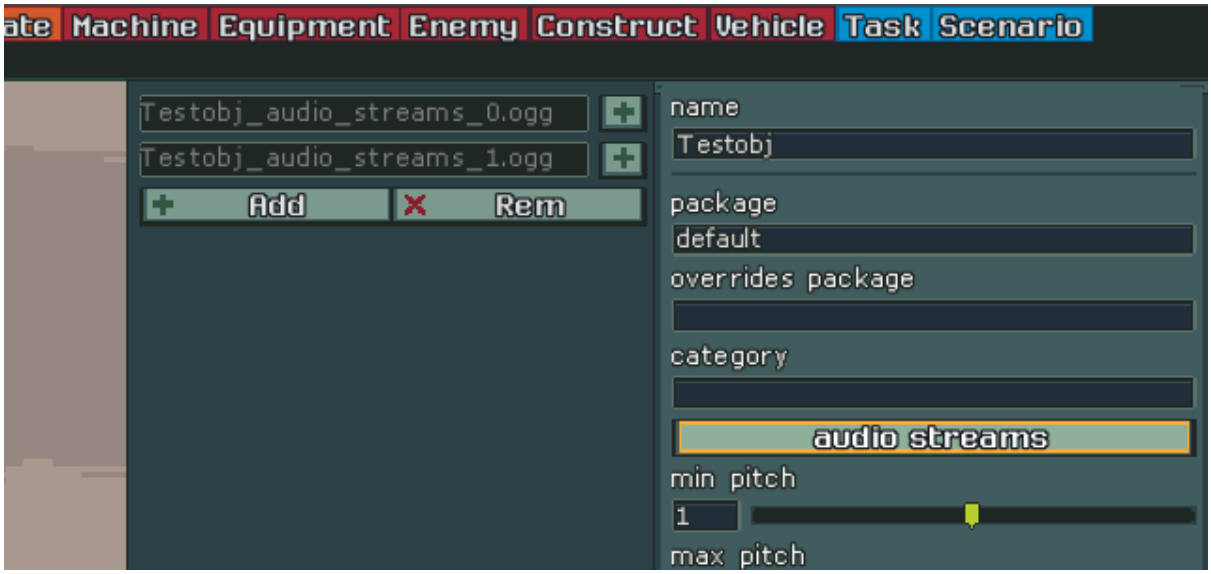

Figure 25. "Audio streams" property of a Sound object. Files are added one by one in the left panel.

All functionality related to game object's external asset files was implemented in a manner where it is as simple as possible for developers to add these properties to new Systems. No logic for managing these asset files was written into the data classes themselves. Adding a new asset file property only requires defining a new exported String or String Array property with the necessary export hints.

4.1.2 Propagation of data name changes

The game data loading logic of the game uses object names to associate game objects with each other. In all cases of object-to-object relations one game object's file will contain the name of the other object as a value of a String type property. After loading all game objects from files into memory, these names are then looked up in the internal collections of objects and the actual object instances are retrieved. The reason behind this is to maintain human-readability in the JSON files where the game saves its game object data.

To keep with the requirements agreed upon with the main developer of the game it was decided that this human-readable format would be preserved in all future developments. However, the pre-existing system did not have any logic for handling situations where an object's name is changed when another object has a reference to it. In such cases the original name of an object would still be stored in the dependent object and the association would break. Upon next load, the actual object could not be found using the old name. Consequently, content would have missing parts, broken logic and in some cases crashes. Since the mod creation and distribution

tools are intended to be used by any user willing to do so, such a limitation would be very unintuitive and labour-intensive.

To solve the problem of preserving game data relations, logic was added where any changes to a game object's name would trigger a propagation of this change through the dependency tree existing between the objects. The general work principle of the name propagation system has two main parts - mapping dependencies between systems and propagating name changes through the resulting network of dependencies.

On launching the game and after all Systems have been imported, the following steps are performed to find dependencies:

1. Check one instance from every System for properties which reference other Systems.
2. For each System, create an entry in a Dictionary type variable containing entries for every other System that can have references to it, along with the property's name where the reference is located.
3. For each System, additionally mark down each property containing a Subsystem and recursively run steps 1. and 2. on these Subsystems.

The result of these steps is a Dictionary of inter-System dependencies, allowing for the quick retrieval of information about which Systems are dependent on which. If the name of a game object is changed then this information can be used to determine which Systems' objects and which property values need to be checked. All properties present in the dependency dictionary are checked for the old name of the changed game object. If a match is found, then that value is changed to match the object's new name. The same is done if a dependent game object's Subsystem contains a reference to the object.

Figure 26 depicts the logic of how an Image object name change is propagated to an Enemy object. Steps depicted by the numbered arrows in the figure are:

1. An Image object's name "critter_enemy_leg1" is changed to another value. The Image System is looked up in the dependency dictionary.
2. Each dependent System is checked, the Enemy system among them.
3. Each property and Subsystem property for the dependent system is checked on every Enemy object. In the Subsystem *body* property *idle_image*, this Enemy object has a value that doesn't match - nothing is done.

4. In the Subsystem *legs* property *image* this Enemy object has a matching value (“critter_enemy_leg1”). This property is changed to the new name of the Image object.

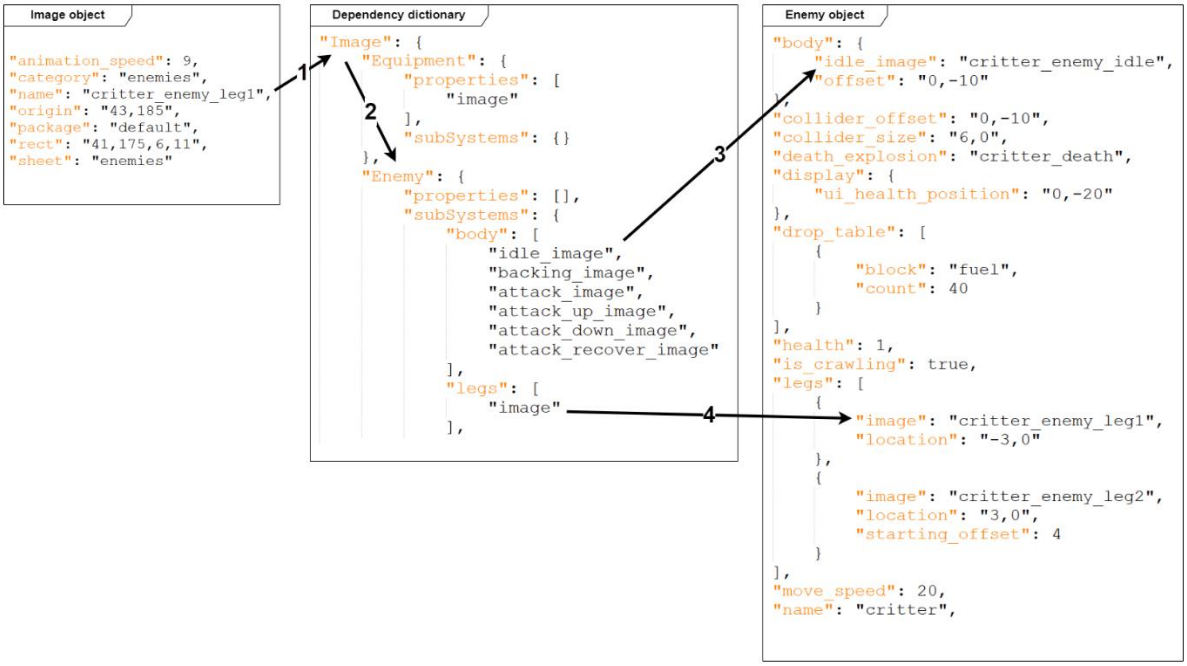

Figure 26. Changing the name of an Image object “critter_enemy_leg1”.

To detect the dependencies between the Systems, three separate methods were used. Relatedly, three conventions were agreed upon which should be followed when defining new Systems and properties in the future:

1. If a property in a System is a reference to one (String) or more (Array of Strings) other Systems, the name of that System must always be included as a Godot export hint.
2. If a property contains a Subsystem, a second property with the name format “<original property name>_type” must always be included. This property specifies the Subsystem class referenced by the original property and can be used to determine which Subsystem to check. These properties were already being included beforehand for editor UI display, but the practice was solidified as a hard convention.
3. If a property in a System is a Dictionary type containing references to another System or a Subsystem, a separate method with the name format “<property name>_editor” must always be included. This method returns a metadata Dictionary containing either *type* or *keyType* values. These values specify the System referenced. This option was necessary due to Godot’s GDScript language not allowing type hints on Dictionary type

properties at the time of development. Like in convention 2 above, such methods were already being included in the project for editor UI display.

Due to the name propagation logic being relevant only while in the editor, the speed of the solution was not of the highest priority. Rather the focus was on the dynamic nature of the solution in both retaining the existing architecture and human-readability of the data files. Even so, there were no notable performance issues encountered in the editor after this development. The propagation did not have a delay noticeable enough to cause concern even when propagating through large amounts of objects. While this functionality is not directly part of the content package system it was determined to be essential for usability. Furthermore, this logic would later also be used when allowing duplication of content packages and logic related to content package name changes.

4.2 Content package solution

One of the core parts of player-made content distribution in Blastronaut is the content package management system itself. The initial content package implementation was designed to be independent of the final goal of integrating with the Steam Workshop. As such, this section covers only the local parts of the content package logic which was intended to work without workshop integration.

With the goal being to allow players to create their own content packages, the minimal requirements for the content package solution were:

- Allow the player to create content packages.
- Allow the player to delete content packages.
- Allow the player to duplicate existing content packages.
- Allow the player to add any game object data to belong in a specific package.
- All objects in a content package will be contained in a single directory for portability.

After iterations on the initial design, the following additional requirements were specified:

- Game objects with the same name do not conflict with each other if contained in separate packages.
- Package names do not conflict with each other.

- Inter-package dependencies should be allowed for objects. For example, if an Image object in the content package “Trees” references a Spritesheet object from the package “Foliage”.
- Objects in content packages should be able to overwrite objects in other content packages. This would allow modifying content shipping with the game. This is due to many common modifications being small changes to the game’s existing content rather than additions to the game.

4.2.1 The Package System

To organize game objects into content packages, a data structure was needed to hold metadata for defining a content package. The overall logical architecture for content packages was designed to follow a similar architecture to other Systems in the game. A new System for storing content package metadata was defined, called Package. From the perspective of re-using existing functionality, large parts of the management of content packages could follow already existing object management logic. This includes creating, duplicating, deleting, and renaming packages, as well as displaying the UI elements required to perform these actions. Additionally, existing logic for defining inter-object relations could be leveraged to make any object aware of what content package it is in. A property with the name *package* was added to all Systems. This property contains a reference to a Package object, specifying the content package an object belongs in.

The Package System was initially integrated into the game similarly to all other Systems. Its *path_user* and *path_resource* values were defined in the System metadata file and displayed as a tab among the editor System selection tabs. However, a few key differences existed when comparing with other Systems already in the game.

Firstly, creating a Package object needs to simultaneously create a new directory in the file system to contain the data that belongs in the content package. The purpose of this was to make implementing the distribution of the content packages as straightforward as possible. File system directory management logic for the following cases was added:

- Whenever a new Package object is created, a new directory with the name of the Package object is created.
- When a Package object is deleted, its content directory along with all its contained objects are deleted as well.

- When an existing Package object is duplicated, all objects referencing it are duplicated and changed to reference the duplicate Package object.

The location of content package directories is specified by the *path_user* value for the Package System. In the final implementation, this was set to be a directory named "packages" in the game's user path (Figure 27).

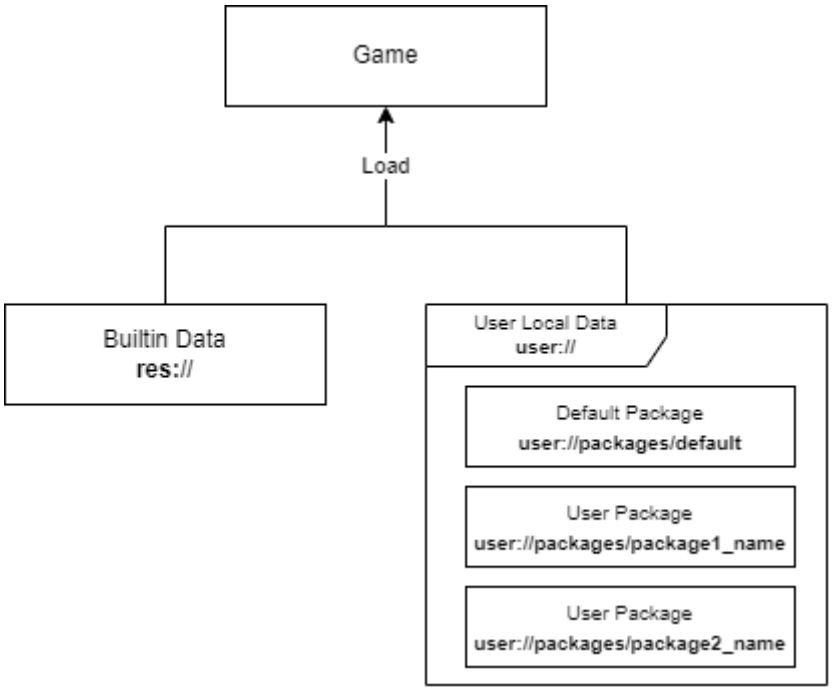

Figure 27. Blastronaut file system directories after adding content packages.

Two Package objects were defined in the resource path of the game:

- **builtin –** a content package containing all objects shipping with the game;
- **default –** a content package containing any object created in the editor that is not yet placed in a package.

These two content packages are included with the game and cannot be disabled or deleted by the player. This way built-in content loading can follow a similar logic to user-generated content. Having fewer differences between different types of content simplifies the loading solution significantly.

4.2.2 Package content loading

Each Package object specifies a directory with the same name as the object itself. This means the JSON file location of other objects is dependent on the Package object they are referencing. In the pre-existing solution all Systems were simply loaded from their respective resource

(*path_resource*) and user (*path_user*) paths defined in the System metadata JSON file. However, after adding content packages the resource and user paths for all Systems were changed from absolute to relative paths. The full JSON file path of each object now contains the name of the content package they are in, making absolute paths impossible to use. To keep data loading structured, the Package objects are loaded before any other Systems. For each Package object the objects referencing it can then be loaded.

To maintain existing data loading and saving logic, a utility method was added that would return the full user path for any object. The path is assembled from the object's *name* and *package* property values and its System's *path_user* value. Every case where the System's *path_user* value would be retrieved in the past was replaced with a call to this method.

The initial implementation for loading and saving Package objects had the JSON files for Package objects outside the directories which contain their other content. To make later implementation of content package distribution simpler, this logic was changed. The final implementation was designed so Package object paths would always resolve to the root of their own content directory.

4.2.3 Package-aware names

The pre-existing internal data structure of the game contained an object collection for each System. These collections contained key-value pairs tying an object's name to the object itself. When factoring in the concept of content packages the previous solution would run into an issue if two packages contain a game object with the same name. While making modifications it is very likely that eventually the same common names would need to be used for simple game objects - like "rock" or "tree" etc. Due to this, the *name* property of an object could not be used to uniquely identify a game object anymore.

To uniquely identify an object, a combination of an object's name and the name of the package it belongs to was used. This is referred to as a package-aware name throughout the thesis. The format picked for package-aware name Strings was "<object name> [<package name>]" - the object's name with the package's name in square brackets. This format was deemed easily readable and thus suitable for maintaining human-readability of the game object's JSON files. Any case in the code where an object's name would be used was changed to use a package-aware name instead. For example, package-aware names would be used in any cases of inter-object references (Figure 28).

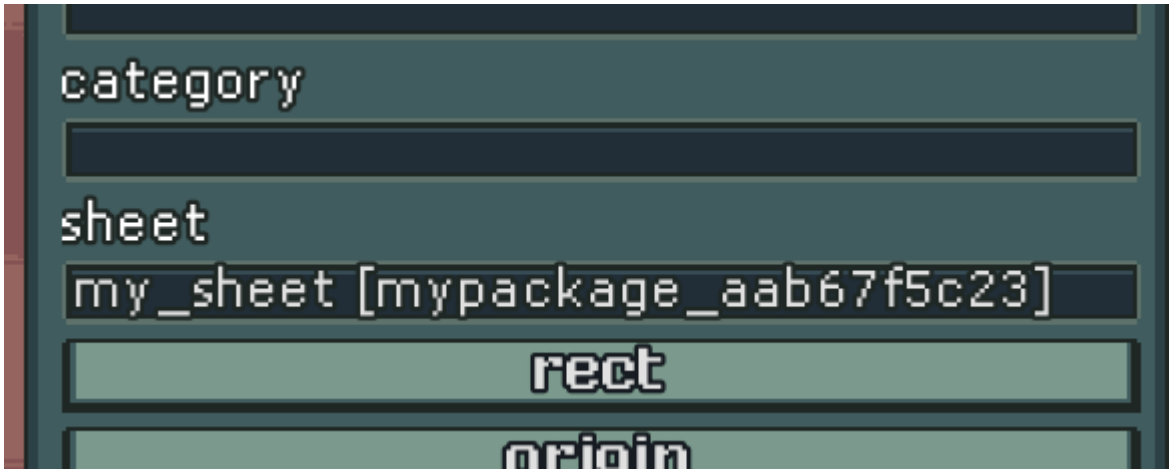

Figure 28. Image object referencing Spritesheet “my_sheet” from the package “mypackage_aab67f5c23”.

Using package-aware names in inter-object references required additions to the name propagation functionality described in Chapter 4.1.2. Since Package names are now also part of the object reference, a propagation needs to be triggered not only if the object’s name changes, but also when its package-aware name does. This includes two new cases:

A. An object is moved into a new package - the *package* property changes.

B. The name of an object’s referenced Package object is changed. First, the package’s name change is propagated to change the object’s *package* property. The change is then re-propagated as the object’s package-aware name changes.

4.2.4 Package identifiers

One problem to overcome with the creation logic for packages was the issue of content package name collisions. The first iteration of the package creation functioned simply like all other systems: the user enters a *name* property value when creating a Package in the editor. While this design worked in theory, the concept had a few flaws. The issue is that package names are used for directory names in the user’s file system as well as in object-to-object references. Any overlap in names would cause the content packages’ directories to interfere with each other in the file system. Furthermore, references to objects in other packages could not be guaranteed to be resolvable to a single package. At the same time, name collisions between packages created by multiple different people were considered to be a real possibility.

To solve the issue the initial proposed solution was to use a similar logic used by the pre-existing data framework to avoid object name collisions. In that case when a user downloads a

package called “Trees” and another package with the same name, the second one would locally be changed to be called “Trees1”. Changing the names upon installing the packages would ensure that each person’s local data structures contain no duplicate names. However, this solution was deemed to be flawed. However, this logic would not work in all cases and introduces unnecessary complexity in constantly renaming local versions of packages.

Instead, logic was designed which would aim to guarantee the *name* properties of Package objects being generated to be globally unique across all content creators. To achieve this, editing the *name* property of a Package object was disallowed in the editor. Instead, a *package_identifier* property was for Package objects. When the *package_identifier* property of a Package object is changed, a value is generated into the *name* property of the object. This new name consists of the package identifier itself, as well as the beginning of an md5 hash of the identifier and the player’s Steam User ID as a string (Figure 29). This way a package with the same identifier created by the same player will always have the same generated name, leading to a predictable naming scheme. If a player accidentally deletes a package that they are already referencing in another object, they can for example re-create it to make the broken references work again. If the game is not connected to Steam, a random number is used instead of the Steam User ID, resulting in more random hashes.

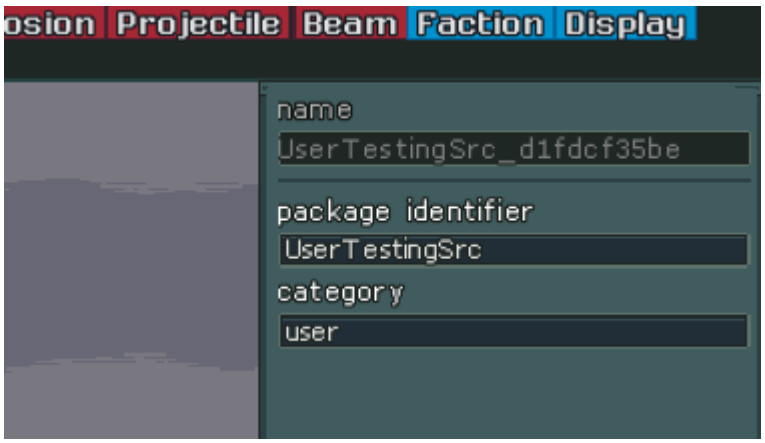

Figure 29. The “name” and “package identifier” properties of a package.

The MD5 algorithm was picked due to being faster[37] than SHA1 and SHA256 - the other two algorithms available natively in Godot. Furthermore, only the beginning of the hash was used

[37] https://www.geeksforgeeks.org/difference-between-md5-and-sha1/

to keep the generated names easier to read. This was not considered an issue due to the estimated amount of Blastronaut mods.

When generating the *name* property value, the local game object collections are checked for packages with the same generated name. If any exist, internally a number is added to the end of the identifier and the hash re-created until the name is locally unique. Due to the amount of possible hashed strings and *package_identifier* values, it was deemed unlikely that these names would collide.

4.2.5 Overriding content

Many modifications to games tend to be smaller changes to existing content of the game, rather than additions. In the initial implementation of the package system, overwriting built-in content or content from other packages was not possible, as each game object would have a distinct package-aware name even if the object's name itself is the same. Before the implementation of packages, the game would technically allow overwriting content by renaming objects to match the name of built-in content. Since the names were used as references and keys in every data structure in the game, data from the user directory would always overwrite built-in data from the resource directory if they had the same name. This was partly due to user data being loaded after builtin data. The same principle was decided upon to be used for overwriting content in the context of content packages as well.

The most straightforward and robust way to implement this logic as a proof of concept was to add an *overrides_package* property to each object, referencing a Package object. This property would be considered when creating a package-aware name for the object. If an object has a *overrides_package* value, the package-aware name of the object uses the name of the package specified in the override field, rather than its own package. This way, if an object is renamed to match some builtin content and its *overrides_package* property is set to "builtin", its package-aware name will be the same as the one for the built-in content.

An issue remained of multiple overrides in different packages overriding the same content in a single package. In those cases, it is currently not fully predictable which package would be loaded first and get overwritten by the other. However, this was deemed to be a common enough issue of game modifications changing the same in-game objects and was decided to be acceptable for the initial user-generated content distribution solution.

4.3 Functionality for the management of installed content packages

The implemented content package system was designed to function both as a way for players to create their own content packages as well as allow for downloading content packages created by other people from external sources. To allow flexibility in exploring other people's content as well as work on multiple local packages intermittently, a way for enabling or disabling locally existing content packages needed to be added. Disabled packages would be omitted from appearing in the game and in the editor.

4.3.1 Enabling and disabling packages

To track the enabled status of a content package throughout the game, a *package_enabled* Boolean property was added to Package objects. This property could not be saved to the Package's directory, as this would constitute modifying the content package contents. In cases of other people's packages being downloaded from the workshop or elsewhere this status would get overwritten with every update. Instead, the enabled status of content packages was tracked in a separate JSON file residing in the game's user directory. This file defines pairs of package names and Booleans, with each entry signifying whether a Package with a specified name is enabled or disabled. On each load of the game's data this file is read, and all loaded Package objects are checked against the entries. Names of new Package objects not appearing in the dictionary would be added there as disabled packages. Subsequently, any Package object names in the dictionary that do not exist in the game's Package collection are removed from the dictionary.

When loading game data, all Package objects are loaded before other Systems' objects. The game then uses the *package_enabled* property to determine whether objects contained in a content package should be loaded as well. If a Package object is disabled, objects referencing it will not be loaded from the file system to object collections. Consequently, these objects will not appear in the game. Since all existing Package objects are always loaded, it is still possible to display disabled packages to the user. This way, they can be enabled at runtime.

4.3.2 Mods menu

A separate menu (Figure 30) was added to the main menu, titled "Mods". This menu is referred to as the mods menu in this thesis. The mods menu provides players with a user interface to:

1. List any local content packages existing under the user data directory.

2. Enable or disable each package by toggling the Package object's *package_enabled* property.
3. Shows additional information about each package (display name, description).

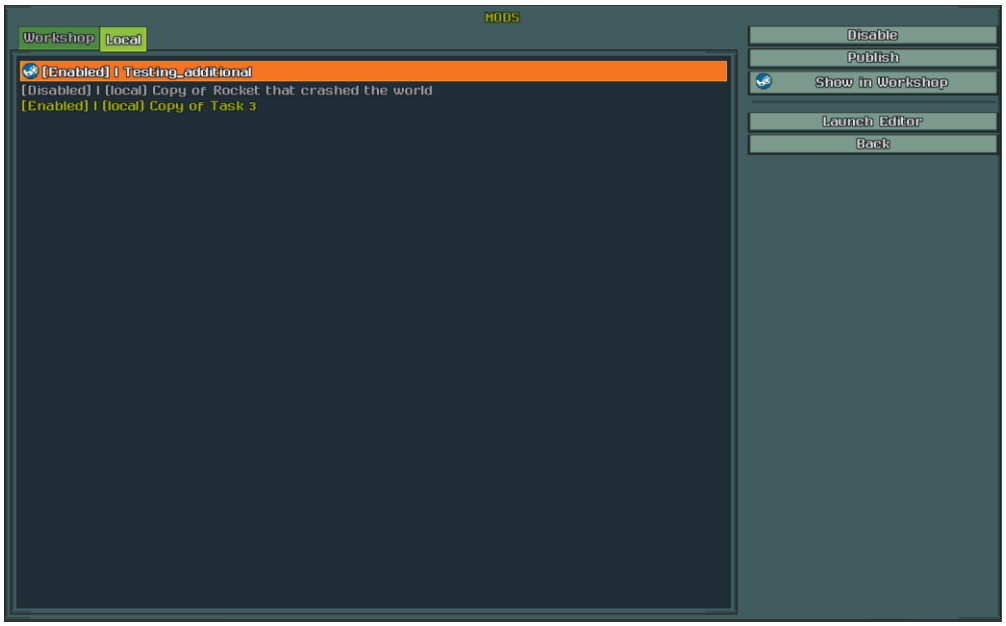

Figure 30. The mods menu showing local modifications in the user path.

Functionality was also added to the mods menu throughout the implementation process of workshop integration. Detailed functionality of the mod menu is described in those chapters.

4.4 Workshop integration

The fourth major development activity was to integrate the created content package framework with the workshop. Required functionality for the workshop integration was the following:

1. The user can subscribe to content packages uploaded to the workshop by other users.
2. The user can upload a local content package to the workshop.
3. The user can update an already uploaded content package.
4. The user can view, enable, and disable content packages that have been downloaded from the workshop separately from their own (local) packages.

The *ready-to-use* Steam Workshop model (described in Chapter 2.2.1) was picked due to it being much simpler to manage for a small development team. Many games use this model because of its simplicity.

4.4.1 Workshop configuration

Before workshop integration can be added, configuration needs to be done on the Steamworks[38] page for the game. The Steamworks site contains pages and utilities for editing any settings necessary for the distribution of a game on Steam. This includes everything from designing the game's Steam store page to uploading builds of the game. Many integrations with Steam require editing settings on the game's corresponding Steamworks page (Figure 31).

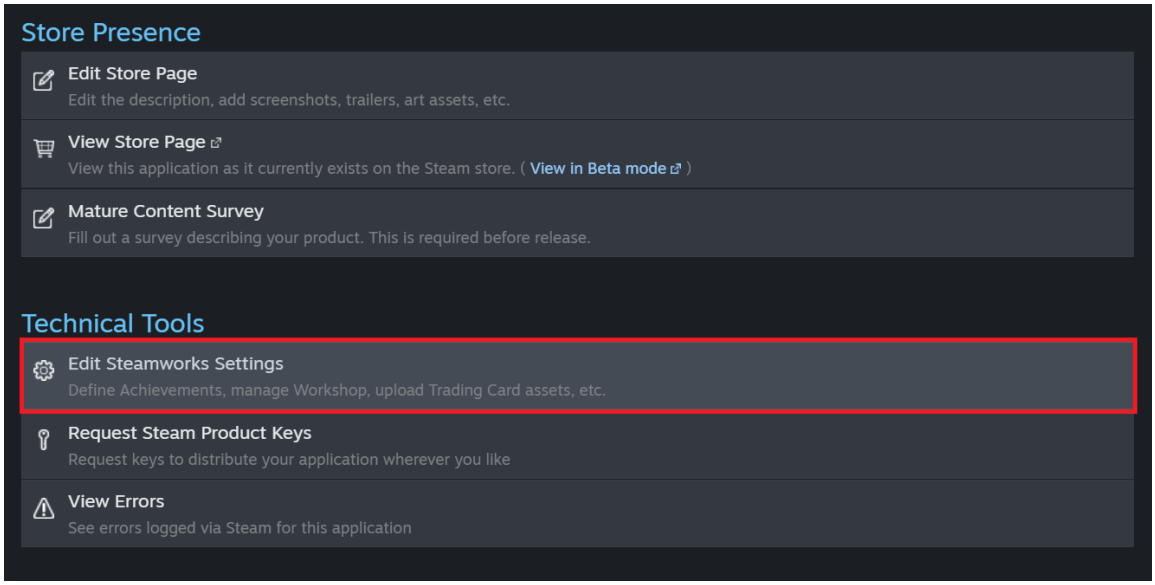

Figure 31. Part of the Steamworks dashboard for Blastronaut. Workshop configuration can be done in the "Steamworks Settings" section (highlighted).

To enable workshop integration for a game through the ISteamUGC interface, this support needs to be enabled on the game's Steamworks page [8]. Two actions are required. First, the "Byte quota per user" and "Number of files allowed per user" values need to be set on the Steam Cloud Settings page[39] (Figure 32). These settings relate to storage of workshop item preview images [8].

[38] https://partner.steamgames.com/
[39] https://partner.steamgames.com/apps/cloud/

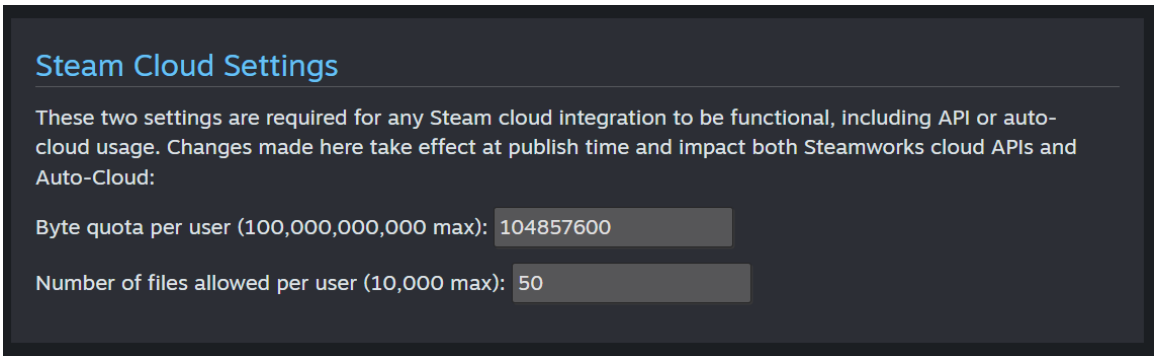

Figure 32. Configuring the Steam Cloud Settings to enable workshop integration.

Secondly, the "ISteamUGC file transfer" option needs to be enabled on the Steam Workshop Configuration[40] page (Figure 33). This allows the game to use the ISteamUGC interface described in Chapter 2.2.2.

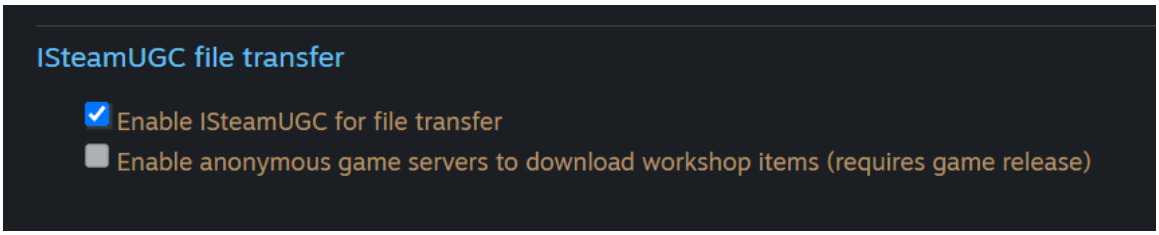

Figure 33. Enabling ISteamUGC for file transfer in workshop settings.

4.4.2 Downloading packages from the workshop

Downloading packages was considered potentially one of the most common use cases of the user-generated content distribution solution for Blastronaut. More users download other people's mods rather than create their own [9]. Consequently, the implementation of the logic for downloading content would be used the most.

During the design process it was decided that Blastronaut would implement a solution similar to games like Teardown, described in Chapter 2.3.1. This means it would be possible to find new mods, download them and enable them, all from within the game client. While not all games offer built-in and convenient modding tools, it is seen as a positive if downloading content is as seamless as possible. This serves to keep the player base satisfied and increase usage of the developed solution [3]. For this purpose, an additional tab labelled "Workshop" was added to the mods menu described in Chapter 4.3.2. This tab displays any content packages that have been downloaded from the Blastronaut workshop (Figure 34).

[40] https://partner.steamgames.com/apps/workshop/

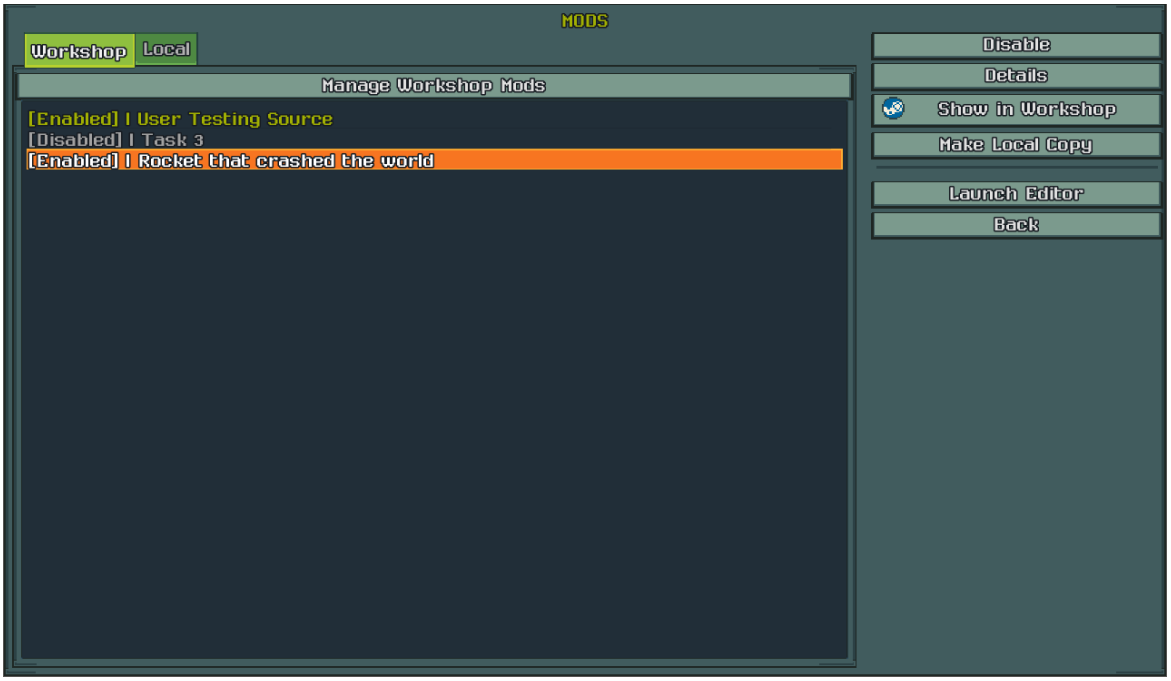

Figure 34. The mods menu “Workshop” tab. Installed content packages downloaded from the workshop are displayed.

The Steam client automatically downloads any item that a user subscribes to in a game’s workshop to a subdirectory in the Steam client’s installation path as described in Chapter 2.2.2. While it would be possible to create a solution that automatically copies those files to the user path of the game, it was decided that this would not be necessary. Instead, a third location for loading game content could be enabled for the game data loading logic (Figure 35). This way no additional logic would be needed to make sure content in the game’s user directories is up to date. The Steam client can then fully handle downloading and updating workshop content.

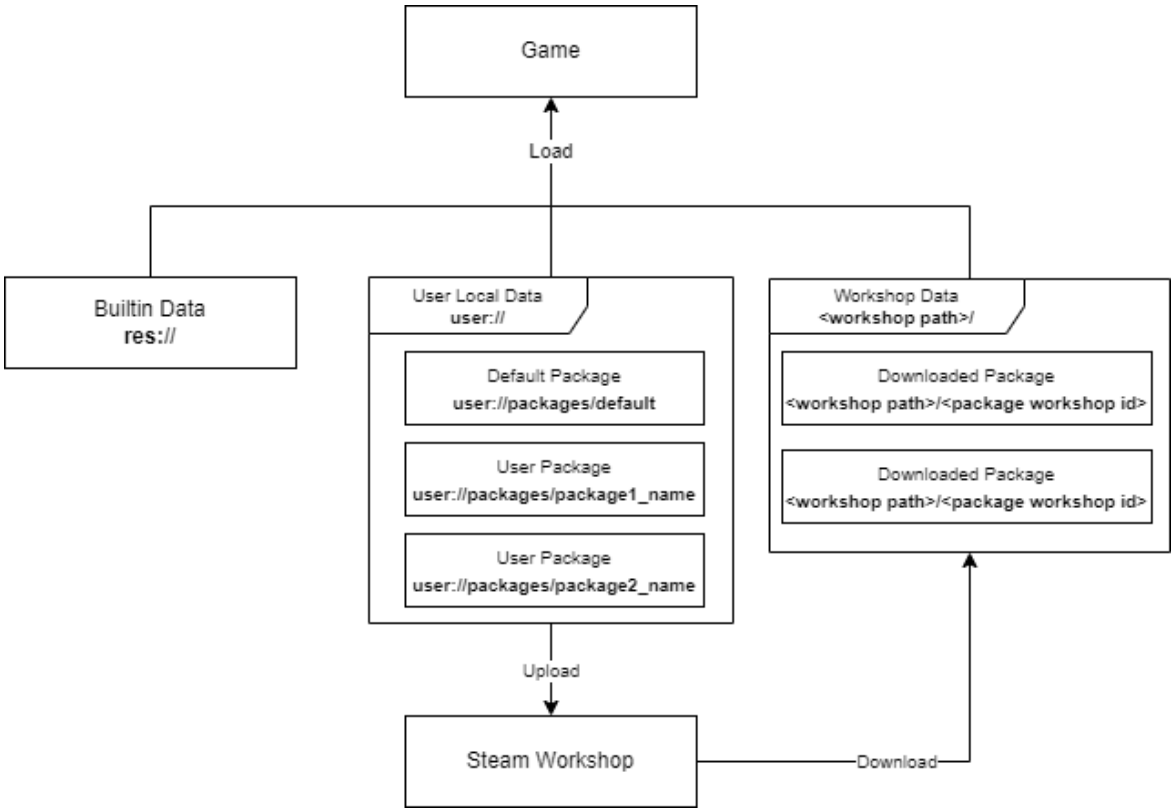

Figure 35. The final structure for Blastronaut file system directories, including workshop support. The workshop path depends on where Steam and the game are installed.

The ISteamUGC interface can be leveraged through the GodotSteam library's UGC module[41]. The *getSubscribedItems* method is used to query a list of workshop item ID-s for all items the currently active Steam user is subscribed to. Calling the *getItemInstallInfo* method for each returned workshop item ID, information is received about the state of the subscribed items. If the workshop item is downloaded to the user's computer, this method will return the following values:

- **ret** - a Boolean indicating whether the item is installed locally;
- **folder** - the local path of the workshop item's content directory in the user's file system;
- **size** - the size of the workshop item in bytes;
- **folder_size** - the size of the downloaded local directory in bytes;
- **timestamp** - a timestamp with the last update time of the item.

When loading the game's data, subscribed item states are queried. For each installed item, the *folder* value is checked. If the path specified contains a JSON file for a Package object, then

[41] https://gramps.github.io/GodotSteam/functions-ugc.html

the content of this directory is loaded similarly to how data was loaded from the resource and user paths.

An additional Boolean property named *workshop* was added to all Systems, similar to the pre-existing *external* property. This property is set to *true* if an object is loaded from a Steam client download path instead of the game's resource or user paths. Using the *workshop* property, it is possible to disallow modifying and deleting downloaded packages in the editor. Directly editing content downloaded by the Steam client would result in the modifications getting overwritten whenever the content is updated and re-downloaded. Instead, the intended workflow is to create a local copy of the package and modify this copy instead.

A button labelled "Make Local Copy" was added to the mods menu for this purpose (Figure 36 D). Pressing the button creates a duplicate of the selected content package as a local package in the user path. The player can then edit this copy of the downloaded content package like any other local content package. Additionally, a button labelled "Details" was added to view the content package's metadata, like its description (Figure 36 B).

To remove a content package downloaded from the workshop from the game, the intended way is to have the player unsubscribe from the package in the workshop. Otherwise, if the user were to delete the local files of a workshop item the Steam client would automatically re-download the files. To allow easier subscribing and unsubscribing for Blastronaut content, a button labelled "Manage Workshop Mods" was added in the mods menu (Figure 36 A). Pressing this button opens Blastronaut's workshop home page in an overlay window on top of the game (Figure 37). To view the page of a specific content package, a button labelled "Show in Workshop" was added (Figure 36 C). This button opens the workshop item's page directly.

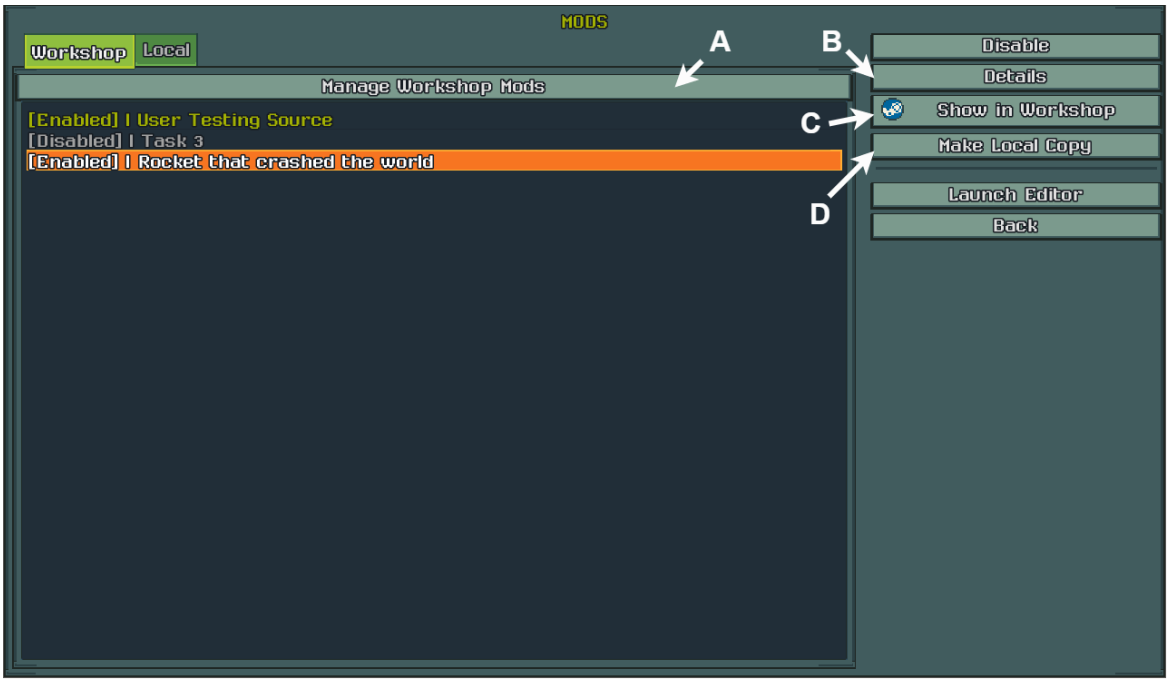

Figure 36. The mods menu workshop tab. Buttons are marked with letters A-D.

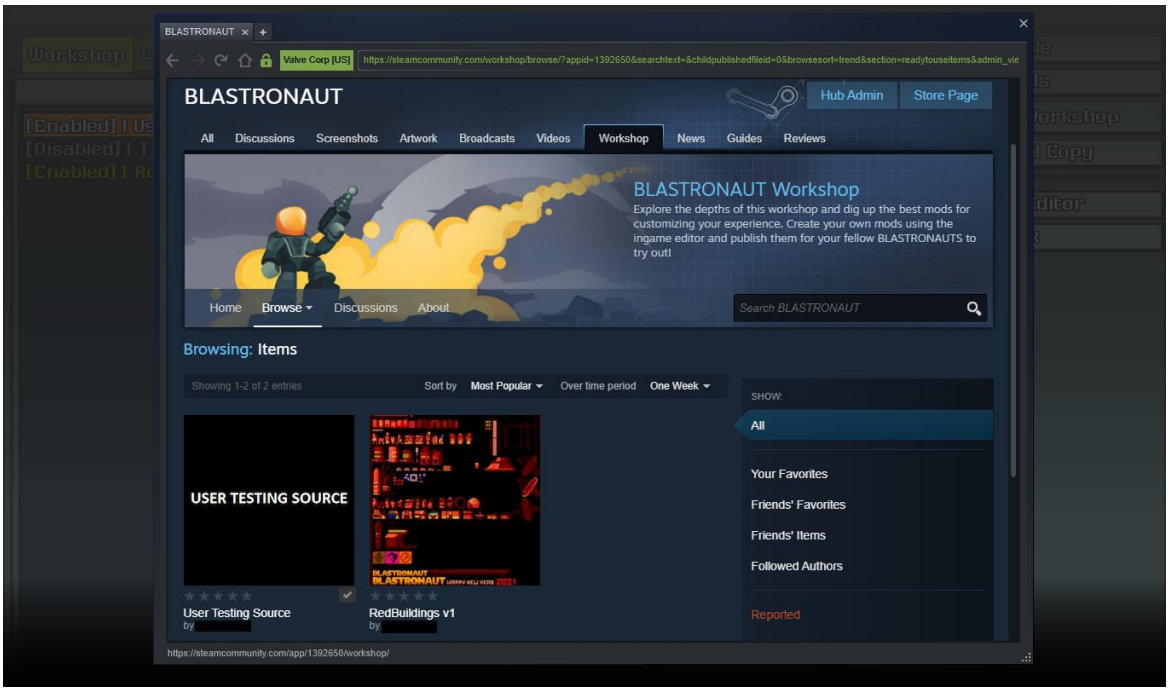

Figure 37. In-game overlay showing the home page of the Blastronaut workshop.

This solution uses the in-game overlay[42] feature of the Steam client. The in-game overlay allows opening any web page as a window layered on top of the game. Since game workshop pages can be viewed as web pages, Blastronaut's workshop can also be displayed in the

[42] https://help.steampowered.com/en/faqs/view/3978-072C-18DF-FBF9

overlay. Various other games have used this method in the past, including Teardown, mentioned in Chapter 1.4.1. Through the overlay the player can subscribe to and unsubscribe from mods the same way as if they navigated to the game's workshop page outside of the game.

In cases, even games with extensive content creation tools and workshop integration like Don't Starve have had the download process of subscribed content purely rely on the Steam client to trigger the physical download of the data. However, this means that the game needs to be restarted before content appears in the game. By default, the Steam client does not start downloading content while the game is running.

To improve the user experience in Blastronaut, whenever the in-game overlay window is closed the game again queries a list of content that the player is subscribed to. This is done via the ISteamUGC interface like on initial loads. Subsequently, the following two checks are performed:

1. Trigger a download for any workshop item that is not locally installed but that the user is subscribed to by calling the *downloadItem* method from the GodotSteam UGC module. The method is given the mod's workshop item ID as an argument.
2. For any workshop item that exists locally, but that the user is no longer subscribed to, remove it from the game's internal game data collections.

The GodotSteam UGC module provides GDScript signals[43] that are emitted when a workshop item is installed, updated or the Steam in-game overlay toggled. The data loading logic of the game was connected to these signals. This allows the game's data loading logic to load or unload any content packages whenever they are installed or removed. Data loading in turn emits its own signals, which the mods menu connects to. When a content package is loaded or unloaded, the mods menu content package lists are updated immediately to reflect this change. This pattern is commonly called the observer pattern [10]. The pre-existing game data loading logic could only load all the game's content at once. Therefore, the logic was reworked to allow both single Package objects and the objects in the content packages to be loaded separately at any time during the game.

[43] https://docs.godotengine.org/en/stable/getting_started/step_by_step/signals.html

4.4.3 Uploading packages to the workshop

Not all games integrate a tool to publish content to the Steam Workshop from within the game. For example, in Don't Starve an external tool is provided for this purpose. However, when designing the solution for Blastronaut it was decided that this functionality would be implemented in the game itself. One of the reasons is to adhere to the principles followed when the integrated editor was created. That is, having all tools be part of the game was deemed to be the most consistent in terms of user experience. Consequently, all activities for publishing created content packages are performed via the mods menu.

A button was added to the mods menu, labelled "Publish" (Figure 38-A). Pressing this button opens a separate menu interface seen on Figure 39. This menu will be referred to as the publish menu. An additional button in the mods menu labelled "Show in Workshop" is enabled whenever a local package has been uploaded to the workshop (Figure 38-B). When enabled, this button works identically to the one in the "Workshop" tab of the menu.

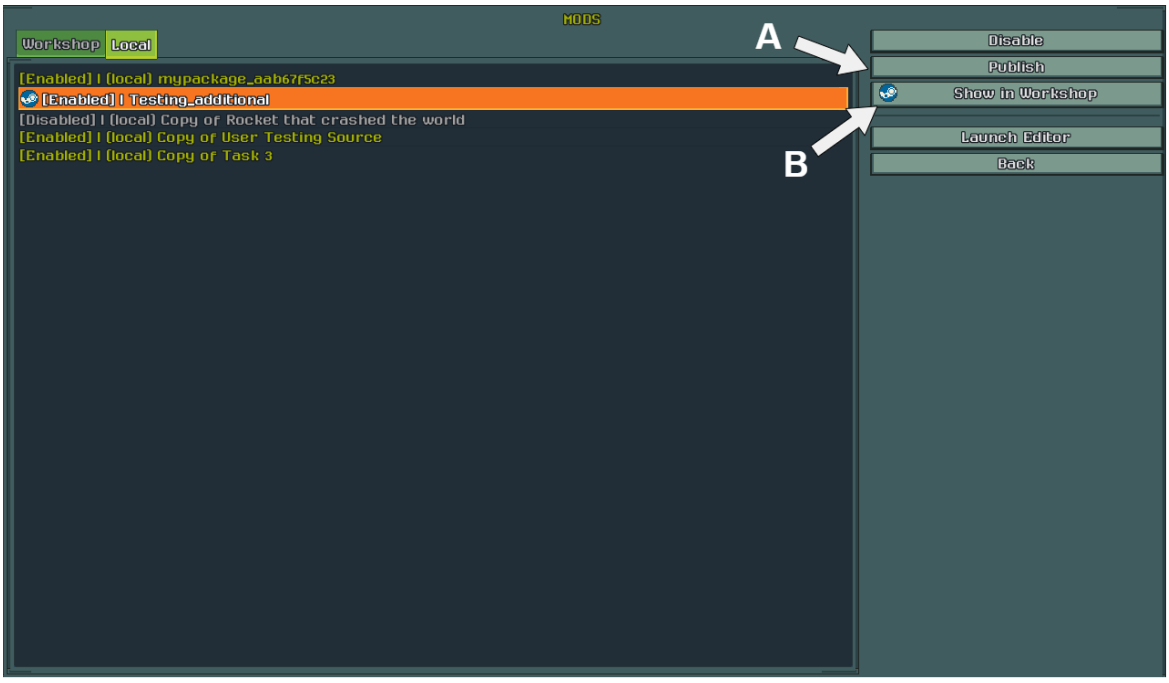

Figure 38. The mods menu "Local" tab. Buttons for publishing a content package (A) and viewing the published package's workshop page (B) are marked.

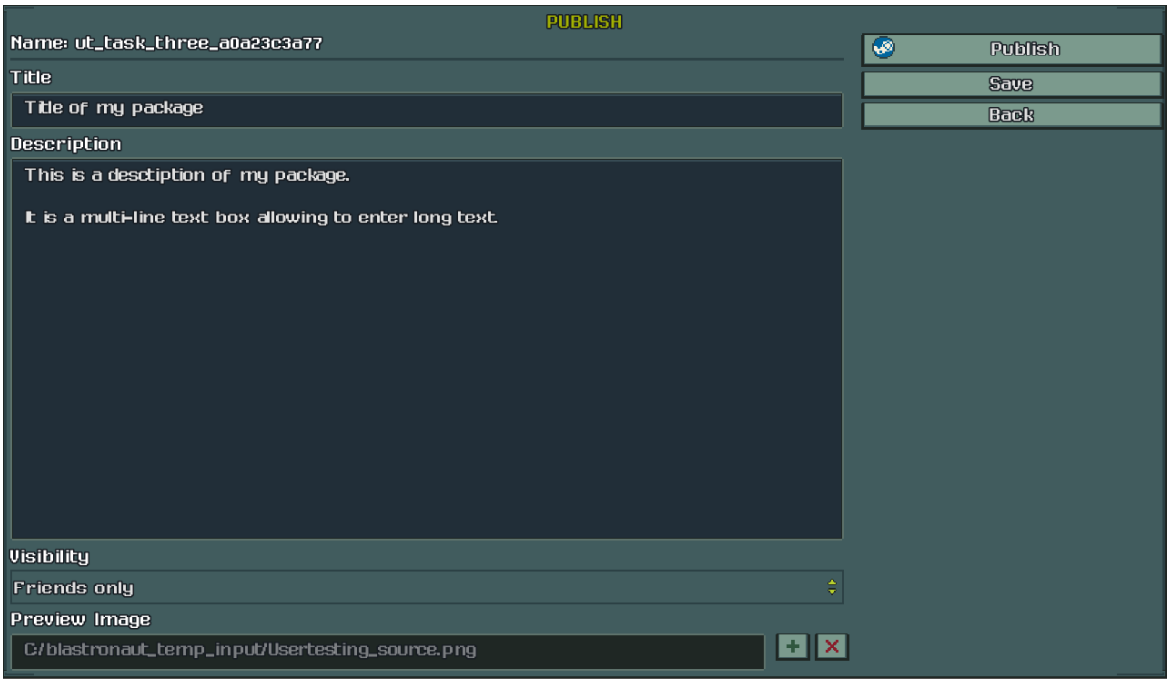

Figure 39. The “publish” window of the mods menu.

The publish menu allows the user to enter information about the package to be used in the workshop. The information includes:

- A **title** for the package, used as the name of the package in the workshop. The title does not need to be unique. This title is also used in-game if set, replacing the “name” property of a Package object in some UI elements.
- A **description** for the package. This shows up as the description of a package in the workshop.
- The **visibility** of the package. This determines who can see and download the uploaded package in the workshop. Options include “private”, “friends only” and “public”. The default is “private”.
- A **preview image** for the package. This is used as the cover image for the package in the workshop.

The title, description and visibility values are all saved to properties of the Package object to be persisted. After entering all required information, the “Publish” button can be pressed. This triggers an upload of the package’s contents to the workshop. On completing the upload, the user is taken back to the “mods” menu package list window.

To update a previously uploaded package the same actions are performed, albeit with a few differences. For example, when re-uploading an existing content package, the publish menu does not require that the user add a preview image. However, if one is added then it will be uploaded, and the preview image of the workshop item changed. Upon updating a package, the content belonging to it gets re-uploaded to the workshop as well.

A *steam_workshop_id* Integer type property was added to Package objects. If a content package has been uploaded to the workshop, then this property contains the workshop item's file ID. When a Package object is first created, this property contains no file ID. Uploading packages to the workshop follows a workflow shown in Figure 40. Steps numbered in the flow chart process are:

0. If the Package object does not have a *steam_workshop_id* property value defined, then a request to create a new empty workshop item is made. For this, the *createItem* method is called. Upon completion of the request, a file ID is returned and stored in the object's property. Step 1 then continues as if the workshop item had existed beforehand.
1. If the package already has a file ID attached to it, this value is used to update the item in the following steps.
2. The item update is started by calling the *startItemUpdate* method. The method returns an update handle ID value. This handle can be used to separately set the title, description, visibility, preview image and content for the item.
3. The update is submitted by calling the *submitItemUpdate* method. Upon completing the item update, a GDScript signal is emitted. This allows other places in the code to react to the completed upload - for example with the mods menu taking the user back to the package list page.

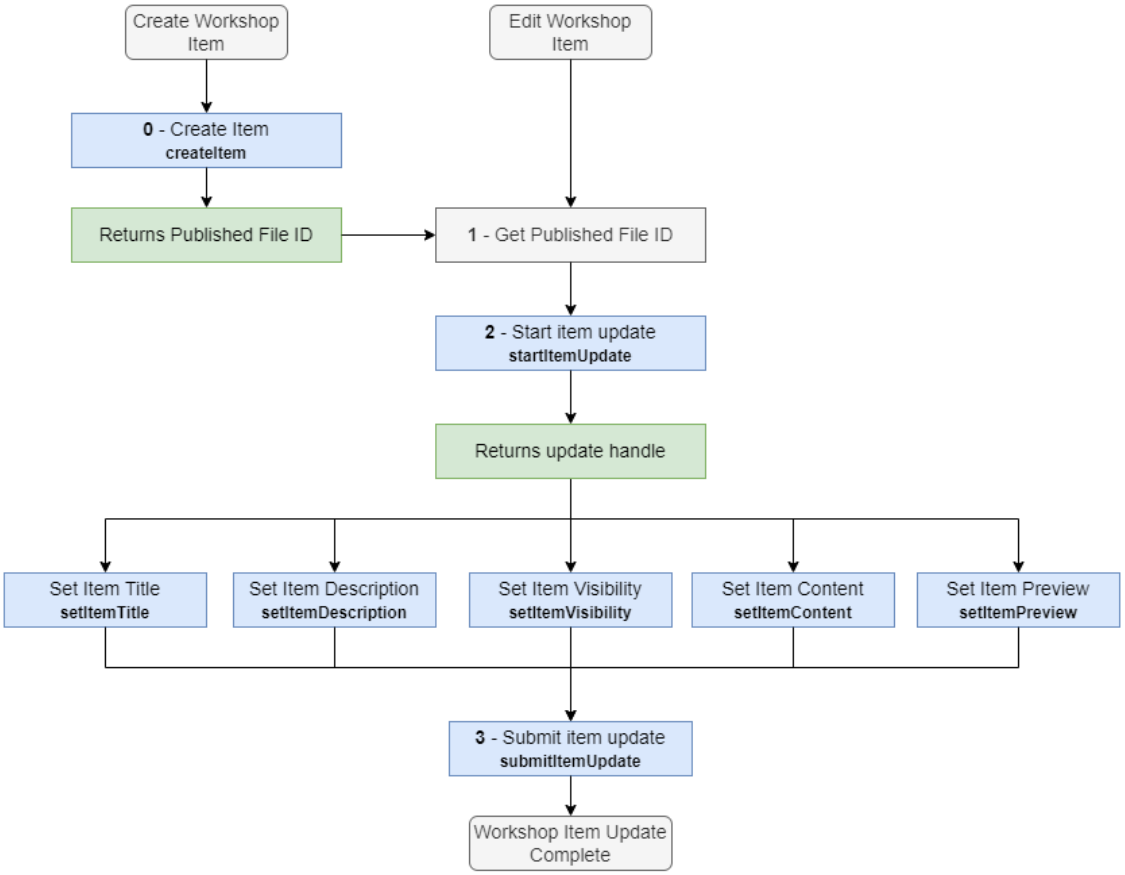

Figure 40. Flow chart for updating a workshop item. Steps in the workflow are marked

After the upload is completed, the player's content package is available for other players to download. Depending on the visibility settings, other players can see the content package listed in the workshop and subscribe to it.

5 Usability testing

The goal of the thesis was to develop the first iteration of a solution for distributing user-generated content in Blastronaut. Since the solution was intended to be used by players of the game, usability testing was conducted. Usability testing, sometimes referred to as user testing, is usually conducted in the software development field to give developers feedback on how well potential users of the software can perform the tasks a solution is intended to allow them to perform.

5.1 Methodology

A set of materials was prepared to conduct the usability testing. These materials, as well as testing results can be found in Appendix I.

Participants were sent a Steam key for the game and the preparation document (Appendix I, Usability_testing_preparation.pdf) in advance. They were instructed to follow all the instructions in this document. The document went through the process of installing the game and switching the game's installation to a separate *user_testin*g branch in Steam. Additionally, the document contained a link to the "Blastronaut user testing modding reference" document, where the participants were instructed to familiarize themselves with a few key aspects of the Blastronaut editor and modding tools in general. The participants were also instructed to try out the Blastronaut game itself beforehand, to get a better overview of the gameplay.

Each participant also needed to be invited to a Steam community group created for the purposes of usability testing. Steam allows a game's workshop page to be visible to both developers and testers. Anybody who is a member of a community group that is specified in the game's Steamworks configuration can visit the workshop page.

A guide document (Appendix I, Blastronaut_usability_testing_modding_reference.odt) was created as a reference guide for supporting the completion of the tasks the participants would be performing. This was intended to mitigate the technical nature of the tasks, as some form of documentation is common for all content creation tools. Additionally, the information contained in the document could later be used by the developers when creating an official public modding guide. In the scope of usability testing, the participants were not expected to read the whole document, but to keep it open as reference.

After the participants had completed instructions in the preparation document, familiarized themselves with the modding reference document and played the game itself, a voice call with

screen-sharing was set up between the facilitator and the participant. The facilitator would then send the participant the document containing the usability testing tasks (Appendix I, Usability_testing_tasks.pdf) and instruct them to begin with the tasks. There were 4 tasks performed by the participants:

1. Subscribing to and enabling an existing package in the Blastronaut workshop via the mods menu.
2. Duplicating the downloaded package, modifying a Display game object contained in it and publishing their copy to the workshop.
3. Creating a new content package, adding Template, Master Template and Sound objects into it, and publishing their package to the workshop.
4. Creating and publishing another new content package with a Master Template in it, but this time with more creative freedom and less instructions.

These tasks covered all three goals of the thesis - download, create and publish. As a result, the testing also covered all elements of the user interfaces involved - the editor and the mods menu.

Testing tasks were completed during a screen-sharing session intended to last approximately 1.5 hours. During the screen-sharing session, user comments and observations were written down by the test facilitator. However, the participants were also encouraged to later include any of their verbal comments in the feedback questionnaire as well.

The feedback questionnaire was provided to the participants as a Google Docs[44] form (Appendix I, Usability_testing_feedback_form.pdf). The link to the form was included in the beginning of the usability testing tasks document. The questionnaire contained questions about each of the 4 tasks, the menus and interface in general, as well as background questions about the participant's prior encounters with modding. No personally identifying information was recorded about any participant.

5.2 Testing

Testing was conducted at a late phase of development in May 2022, with the goal being to test the nearly finalized solution. Trial runs of the usability testing procedure were done with one person and final testing was performed with 5 participants. The small number of participants allowed for examining everybody's feedback in a more detailed manner. All the participants

[44] https://docs.google.com/

picked had prior experience playing video games and either downloading or creating game mods. Two participants cited specific experience creating mods for games in the past. The participants had no prior experience with the user-generated content tools in Blastronaut. People of varying experience with game content creation were picked to get feedback on different aspects of the developed solution.

The screen-sharing portion of the testing was conducted using the Discord[45] application. The task completion portion of testing for all participants lasted from the expected 1.5 hours up to 2 hours depending on how long the participants wanted to continue with the more open-ended fourth task. In one case a bug was encountered which the participant offered to reproduce for testing purposes of their own volition, despite the testing taking longer than expected.

Due to a critical bug being encountered by one of the earlier participants, a hotfix was created to address this issue before the next tests were conducted. The bug caused a crash when performing task actions in the editor in a specific manner and it was deemed productive to minimize potential crashes for the upcoming testing sessions. After the fix was applied, no further crashes were encountered during the completion of testing tasks by other participants.

5.3 Test results

Once testing was concluded, all answers for the open text feedback questions and the notes taken by the test facilitator were compiled into a single document (Appendix I, Feedback.pdf). The document contains tables specifying all suggestions and issues observed during testing. The estimated severity of these issues in the opinion of the author of this thesis was added. Additionally, the tables contain potential solutions for each issue derived from the opinion of the author and suggestions by the participants. Furthermore, the estimated complexity of the proposed solution was included. Severity and complexity are expressed as being either low, medium, or high.

Due to three of the four usability testing tasks requiring the use of the in-game editor interface, a large part of the feedback provided by users was about the editor interface and its ease of use. This was an expected side effect of the user-generated content distribution solution being built on top of the pre-existing editor. Any feedback on the editor and the game in general was also recorded in the feedback file as input for future development of the game. While the editor interface itself was not the primary focus of this thesis, some changes and additions had also

[45] https://discord.com/

been made in parts of the interface during the development of the user-generated content distribution solution. Consequently, some smaller issues with the editor interface were fixed among other issues after testing concluded. All issues fixed after usability testing are highlighted in grey in the feedback file and the solution described.

5.3.1 Task complexity

The complexity of the tasks was rated as expected considering the content of the tasks, with task 1 being rated the easiest on average and task 3 the most difficult (Figure 41). Task 4 was rated as being easier than the third, likely because the users already had a better concept of how the editor and mods menu worked and were not instructed to add as much content into their published package.

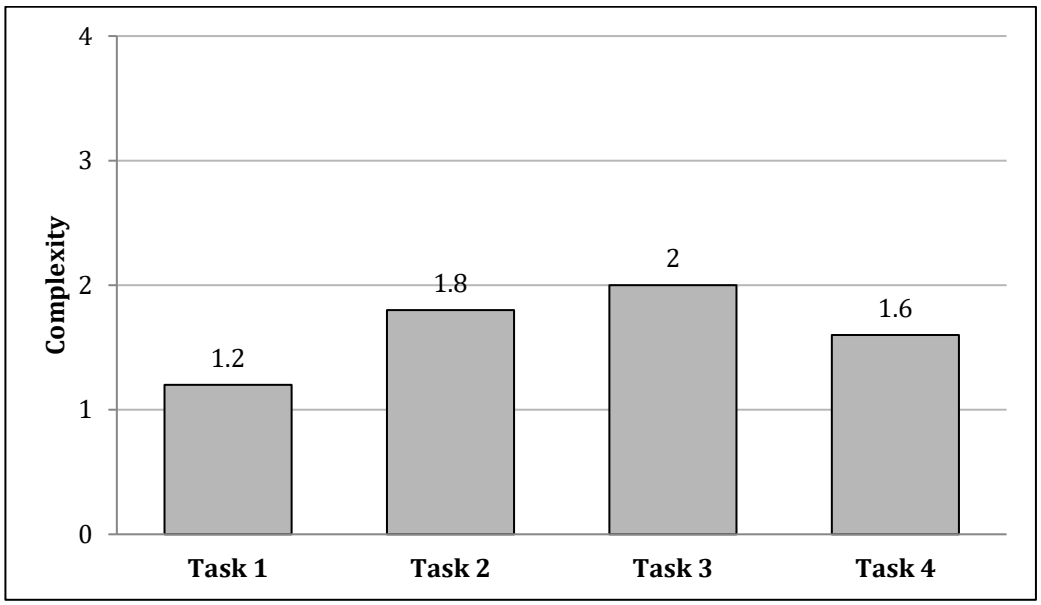

Figure 41. Perceived average complexity of tasks rated from 1-4.

Over all four tasks, the participants were asked to rate the difficulty of the activities they performed in the task compared to other games. In the more complex tasks, there were several participants who answered stating “I have not done anything like this in the past”. This was to be expected because the activities were specific to game modding and in some cases to Blastronaut.

5.3.2 Downloading content packages

In the feedback for task 1 (“Subscribe to a workshop package”), all participants answered that they had previously done similar activities in other games. The games mentioned included various titles with some having workshop support (like ARMA), and some not (like Minecraft).

When comparing the difficulty of the activities performed in the task to other games, all participants rated the difficulty to be the same or simpler (Figure 42).

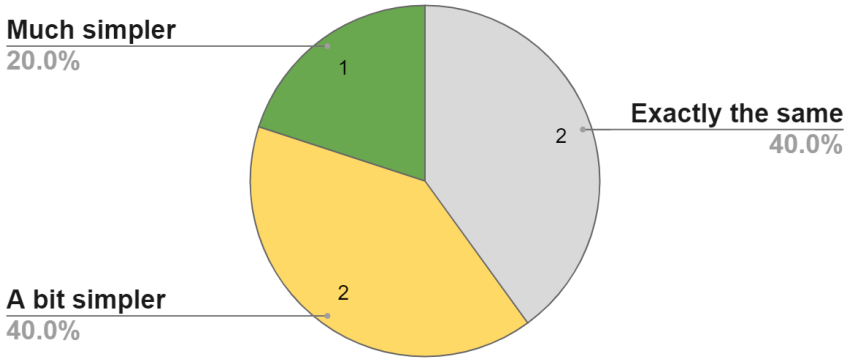

Figure 42. Estimated complexity of downloading mods compared to other games.

The participants had no notable difficulties navigating the menus or finding the package in the Steam Workshop that they were instructed to download. Overall, positive feedback was received for the simplicity of the solution. Participants generally liked being able to access the workshop page of the game inside the Steam in-game overlay. One improvement suggested was adding a clear load order for downloaded packages. This was a feature discussed at an earlier stage in development and decided to not be implemented for the initial version of the solution. Additional feedback focused on displaying the preview image and supported game version of a Workshop item in the mods menu. Finally, the “Manage workshop mods” button was pointed out as hard to detect in some cases.

5.3.3 Creating and publishing content packages

Since tasks 2-4 all focused on the creation and publishing of content packages, they were analysed together. For questions about comparing the task activities in Blastronaut with other games, participants cited no previous experience more often than in task 1. This is likely due to game content creation being a less common activity than downloading mods. However, in cases where participants had previous experience, the solution in Blastronaut was said to be much simpler or about the same as other games. Figure 43 visualizes the proportion of answers over the three tasks.

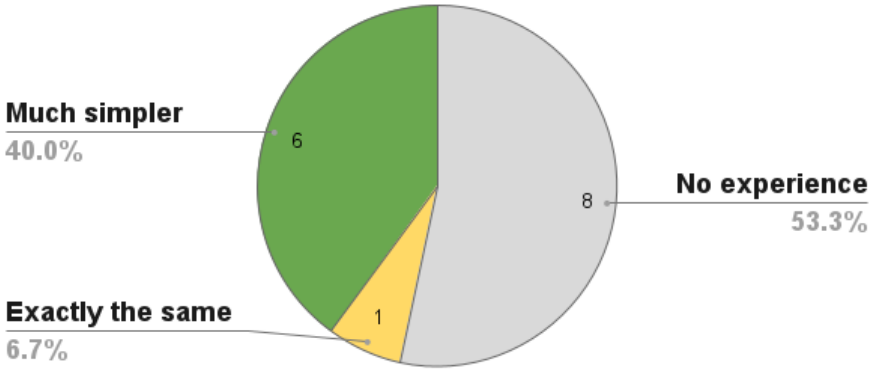

Figure 43. Estimated complexity of creating and publishing content compared to other games. Results across all participants.

Overall, the workflows for using the developed solution can be deemed suitably intuitive for players to use. Most feedback gathered focused on smaller ease-of-use issues, which can be worked on while development of the game continues. A few design issues were also brought out, like the inability to remove content packages from the mods menu. All such feedback was highlighted in the compiled feedback document (Appendix I, Feedback.pdf**)**.

However, all participants successfully completed the usability testing tasks even without any prior experience with the content creation tools in Blastronaut. Participants noted, that with everything integrated into the game, the basics were easy to learn compared to some other games' tools. On the one hand, this might be related to Blastronaut currently being a relatively small game in scope. On the other hand, it also indicates the value of integrated content creation and distribution tools in simplifying user experience.

It is also possible, that additional issues would be brought up if the solution is used by experienced content creators while creating large-scale modifications. These types of modifications can require much more time spent using the content creation tools, highlighting any usability issues. However, this type of feedback can only be gathered after the game has launched and gathered a substantial player-base and community following.

6 Results

The user-generated content solution implemented in this thesis fulfilled the implementation goals set. The solution provides a reasonably simple way to download content created by other users of the game. Additions were also developed to the content creation interface (editor) and background data management, which provide a basic but functional way to define content packages in Blastronaut. It is also possible to publish created content for other users to download using a workflow that is not overly complex. However, the solution in its current form has a few issues, which would need to be addressed before it can be officially released. These issues are analysed in the following Chapter (6.1).

6.1 Future developments

The usability testing feedback document created after usability testing was intended to serve as input for future development efforts. Issues were separated into three main categories:

- Issues and bugs related to functionality developed for this thesis.
- Issues and bugs related to the pre-existing in-game editor interface.
- Issues and bugs related to the gameplay itself.

The issues related to the functionality developed for this thesis will be focused on in more detail. For an overview of issues encountered during usability testing, see Chapter 5.3 for the test results. As feedback for the usability testing and observation predictably yielded mostly usability issues, some core issues or missing features of the solution were not detailed in the feedback.

6.1.1 Game saves and multiplayer

Firstly, the currently existing solution does not have separate logic for interacting with the pre-existing save system included in the game. Players can download new content for their game which could lead to unexpected behaviour when loading old saves. The same applies to the cooperative multiplayer mode of Blastronaut. When playing with other people in multiplayer, players would currently need to manually verify that their games are running the same mods as the host of the game to avoid unexpected side-effects. On one hand this is how the game functioned before - each player could already create local content which would make their game different from another player’s. In the first iteration of the content distribution solution this issue was discussed with the main developer of the game and deemed to be low priority for the initial solution. However, for a truly production-ready content distribution system,

checks would need to be in place to disallow such cases from happening. A simpler implementation could just involve the game checking which content packages are being used in a save game or by the host of a multiplayer game. These checks could forbid the player from loading a save or joining a multiplayer game with a different list of mods. In the long term more advanced solutions could be devised. For example, when saving a game, a list of workshop item ID-s could be included with the save instead. When loading the save, the game could automatically trigger downloads for each content package used. The same applies for multiplayer, where the client could receive a list of workshop item ID-s from the host.

6.1.2 Refactoring data management logic

The overall structure of the current solution's code is not completely ideal. In some cases, programmed logic can be convoluted and marked as such with comments in the code. This was discussed with the main developer of the game during development. It was decided that the initial focus was on getting the tools working in the context of the pre-existing data system. If the solution is to be made production-ready in the future, the code relating to user-generated content distribution could be considered when refactoring the game's data management logic. As it stands, the solution was developed with a mindset not to rework too much of the original data handling logic at an architectural level. Instead, changes were made where necessary and logic developed that would fit the existing structure. However, considering the new functionalities, a more thorough rework could still be warranted to simplify the architecture of the game's data management and make it even more focused on content packages.

6.1.3 Mod dependencies

A feature missing from the initial implementation of content distribution tools in Blastronaut is the tracking of content package dependencies. If one content package uses some objects from another, downloading a content package could trigger downloads for all its dependent packages as well. While this is not something that all games with mod support have, it could serve to make the whole solution even more intuitive to use. To detect these dependencies, each package could provide a method to return information about what packages' objects are referenced in it. This method could return the workshop item ID-s of all content packages depended on. The list of ID's could then be added with the GodotSteam UGC module *addDependency* method when creating or updating a content package. When a package is downloaded, the *getDependencies* method could be used to list these dependencies and download the necessary

content packages. Naturally, additional UI development might be needed to inform the user about the needed dependencies of a package and what was downloaded.

6.1.4 Scripting support

A feature not deemed critical to be included in the initial solution was the ability to add new gameplay logic as part of mods. This is often present in various other games in the form of integration with a scripting language like Lua script. Such integrations can enhance the ability of content creators to add interesting modifications for the game. Assessing the complexity and adding even simple scripting support could be a beneficial future development to investigate. At a basic level scripting support could be implemented in Blastronaut by allowing the user to attach an external script file to game objects in the editor. These scripts could be loaded from files similarly to asset files in the current solution. Some basic life-cycle methods would need to be implemented - for example support for an initialization method to set the initial parameters of a script and a "process" method which would be run periodically by the game. Specific methods can then be made available for the scripts, allowing these scripts to query information about the game and affect the game world. This includes methods like getting in-game objects' or players' positions in the game world or spawning new objects. However, implementing comprehensive scripting support is potentially a very long task, involving numerous iterations on the system in the future. If such scripting is introduced into the game, it would also be beneficial to then convert any built-in game logic that can be scripted to use the scripting system. An additional consideration is the continual maintenance of the system and documentation of the scripting API, both of which also require development effort. However, in the long-term scripting support can help with the longevity of a game. In some cases, like in Don't Starve, it has even allowed the game's community to fix bugs in the base game[46]. However, this is only possible if the gameplay logic in the base game is also largely implemented using the scripting support and can thus be modified.

[46] https://steamcommunity.com/sharedfiles/filedetails/?id=2767977231

7 Conclusion

Adding official tools for creating and distributing User-Generated Content has been shown to have various benefits for game developers. The main goal of this thesis was to review existing game content creation solutions and implement a similar easy to use solution in Blastronaut. Additionally, the implementation was validated by conducting usability testing on a small set of participants. The existing content creation tools were updated to allow creation of separately distributable content packages. Steam Workshop integration was added to allow players to distribute the content that they make. Usability testing was conducted with a small number of participants. Individual feedback was thoroughly analysed. Based on the feedback, the implemented solution is suitably intuitive to use even with no prior experience with the tool. However, before the solution is released, it would be beneficial to resolve high priority issues found in the consolidated feedback.

8 References

[1] Stefan Koch and Michael Bierbamer, "Opening your product: impact of user innovations and their distribution platform on video game success," *Electron Markets*, vol. 26, pp. 357–368, 2016.

[2] Dayi Lin, Cor-Paul Bezemer, and Ahmed E. Hassan, "An empirical study of early access games on the Steam platform," *Empirical Software Engineering*, vol. 23, pp. 771–799, 2018.

[3] Leo Poretski and Ofer Arazy, "Placing Value on Community Co-creations: A Study of a Video Game 'Modding' Community," in *ACM Conference on Computer Supported Cooperative Work and Social Computing*, 2017, pp. 480–491.

[4] Kyle Hulse. (2020, March) Why UGC is the Future of Gaming. [Online]. https://kylehulse.medium.com/why-ugc-is-the-future-of-gaming-faa9e8b1bf1a (17.05.2022)

[5] Valve Corporation. Steam Workshop. [Online]. https://partner.steamgames.com/doc/features/workshop (17.05.2022)

[6] Tom Sykes. (2013, August) PC Gamer. [Online]. https://www.pcgamer.com/dont-starve-gets-fed-steam-workshop-support-has-new-ruins-level-for-dessert/ (17.05.2022)

[7] Michael Heron, Vicki L Hanson, and Ian Ricketts, "Open source and accessibility: advantages and limitations," *Journal of Interaction Science*, vol. 1, no. 1, 2013.

[8] Valve Corporation. Steam Workshop Implementation Guide. [Online]. https://partner.steamgames.com/doc/features/workshop/implementation (17.05.2022)

[9] Katarzyna Bilińska-Reformat and Anna Dewalska-Opitek, "To Mod or Not to Mod—An Empirical Study on Game Modding as Customer Value Co-Creation," *Sustainability*, vol. 12, no. 21, 2020.

[10] Nathan Lovato. GDQuest. [Online]. https://www.gdquest.com/tutorial/godot/design-patterns/intro-to-design-patterns/ (17.05.2022)

[11] Kate Moran. (2019, December) Nielsen Norman Group. [Online]. https://www.nngroup.com/articles/usability-testing-101/ (17.05.2022)

Appendix I – Usability Testing Materials and Feedback

Materials created for usability testing and the results of the testing can be found in the attached **usability_testing.zip** file. Since the original files were provided to usability testing participants as Google Docs documents, inter-document hyperlinks have been removed in the included files. The original files and chapters linked are referenced next to each link in red.

Contents of the usability_testing.zip file:

- **Usability_testing_modding_reference.odt** – A reference guide created for supporting the completion of the usability testing tasks. Not included as a PDF due to containing animated GIF images.
- **Usability_testing_preparation.pdf** – Preparation document with instructions for usability testing participants to complete before testing. Goes through the process of game installation and accessing the correct beta branch of the game on Steam.
- **Usability_testing_tasks.pdf** – Tasks to be completed by participants during usability testing.
- **Usability_testing_feedback_form.pdf** – Feedback form filled out by participants during usability testing. PDF exported from a Google Docs form.
- **Usability_testing_feedback_form_responses_raw.ods** – Raw data with the responses to the feedback form.
- **Usability Testing Assets** folder – Files provided to the participants during the usability testing. Includes one image and one sound file used referenced in usability testing tasks.
- **Feedback.pdf** – Consolidated feedback and suggestions from the results of usability testing.

Appendix II – Gitlab Repository

The Gitlab repository containing the can be found on the following link: https://gitlab.com/perfoon/blastronaut.

To request access, please e-mail t.treikelder@gmail.com

Appendix III – License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Toomas Treikelder**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:
    1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
    1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Developing User-Generated Content Distribution Systems for Blastronaut Game,

supervised by Jaanus Jaggo,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **17.05.2022**