Adrian Ismail

Galactic Void



# Executive Summary

Galactic Void," originally conceived as a richly narrative 2D platformer, faced a developmental pivot due to the intrinsic complexities of the Godot engine and GDScript. Aimed at early teens and beyond, the project's core became refining gameplay mechanics and level design to ensure an engaging experience.

The shift to GDScript, though initially challenging, accelerated progress in character animations and movement mechanics. The development spanned weeks 4 to 15 and was marked by iterative phases: designing the game's framework, crafting a prototype, and developing an alpha version, all interspersed with feedback sessions for continuous improvement. Constraints led to a strategic focus on perfecting a platformer/side-scroller experience, setting the stage for future narrative exploration.

The project's scope was realigned in response to development insights, with the completion of one developed level instead of multiple. This strategic scope reduction was a practical adaptation to the learning curve and time constraints, ensuring quality over quantity.

Post-development, the plan is to expand "Galactic Void" by enriching its narrative depth and artistic expression. The goal is to launch on platforms like Steam or itch.io, leveraging user feedback for ongoing game refinement.

# Main Menu| Provides options to take as first action

From the main menu, a new game can be started whereafter the menu changes to allow player name input. The settings page can be accessed here where the master, music, and sound effects volumes can be changed. These changed values are restored when the settings page is brought up during the paused state of the game. There is also a section to explain the controls of the game before getting into game play.

A screenshot of a video game

Description automatically generated A screenshot of a music player

Description automatically generated

A screenshot of a video game

Description automatically generated

# Pause Menu | Provides the ability to pause the game and adjust the volume, save the game, view the game controls, or quit.

When the escape key is pressed, the game enters a paused state and presents the pause menu. From here, the game volume can be accessed just as from the main menu. There is a section to save the game that gives a pop up notification when the save completes. The game controls can also be viewed here. The pause menu can be exited through either pressing the escape key again, clicking the resume button, or pressing the quit button, which takes the game back to the main menu.

A screenshot of a video game

Description automatically generated A screenshot of a video game

Description automatically generated

# Gameplay Mechanics | Player interactions with the game

The player has the ability to “double-jump” to extend their jumping reach and get to areas they otherwise might not with a single jump. This action is performed after the initial jump. It can only happen after the player has left the ground and anytime before they land. Provided as a benefit, there is a “coyote-timer” that assists the player’s initial jump. When the player leaves a ledge, they are no longer on the ground and are unable to perform their initial jump. Knowing exactly when to jump is difficult since the games collision shapes are invisible and so a timer adds an additional 2/10ths of a second for the initial input of the jump command.

A video game of a pixelated robot

Description automatically generated

There are floating platforms through the level that are implemented with one-way collision, allowing a player to jump up through them, and then landing on top of them.

A pixel art of an object

Description automatically generated

To navigate the level, player movement can be controlled with the A and D keys or the LEFT] and RIGHT arrow keys on the keyboard. To attack an enemy, either the left click on the mouse or the F key can be used and an attack animation will be played. This animation will interrupt other animations as attacking takes precedent.

A video game with a person standing in front of a cloud of smoke

Description automatically generated

# Scenario | Program Usage Description

Use as many scenarios as needed to provide an overview of the system. The scenarios should give a screen shot and describe how the screen is to be used.

# Reports

There are currently no reports for Galactic Void.

# System Architecture

**res://:** is the root directory of the Godot project. When referring to any file within the project, the file path starts with **res://** to tell Godot that the path that at the project root. For example, **res://scenes/level1.tscn** would refer to the **level1.tscn** file located inside the scenes directory at the root of the project.

Below is the file system for Galactic Void, the Scene folder is showing the level1.tscn for demonstration purposes:

**res://**  **EXAMPLE**

├ Art/

│ ├Characters/

│ │ ├ Enemies/

│ │ ├ Health\_UI/

│ │ └ Hero/

│ ├ Fonts/

│ └ Levels/

│ ├ Animated Objects/

│ └ Backgrounds/

│ ├ 30 DARK- PARALLAX/

│ ├ DARK - Peaks Of Lightning/

│ ├ DARK Edition Parallax Background/

│ ├ Forest Parallax/

│ └ Tilesets/

├ **Scene/**  **EXAMPLE**

│ └ **Level1.tscn/**  **EXAMPLE**

├ Script/

├ Sound/

│ ├ Music/

│ └ Sound Effects/

└ default\_bus\_layout.tres

This structure organizes the various assets and scripts used in the game, providing an intuitive and accessible layout for developers and contributors.

## Source Code Structure

This section provides an overview of the source code directories and their respective contents for "Galactic Void." Each directory is organized to facilitate ease of access and modification, ensuring a coherent development environment.

|  |  |
| --- | --- |
| **Code Directory** | |
| **Directory** | **Usage** |
| res:// | Serves as the project root folder in the Godot engine, acting as the starting point for the project's resource file structure. |
| **Art**/ | Contains all game art assets, categorized for efficient resource management. |
| **Art**/Characters/ | Houses art related to the player and enemy characters. |
| **Art**/Characters/Enemies | Stores all enemy character sprite sheets. |
| **Art**/Characters/  Health\_UI | Contains the art for the player's health and lives indicators |
| Art/Characters/Hero | Contains the sprite sheets for the player character. |
| **Art**/Fonts/ | Holds the **.ttf** files for the fonts used in the game. |
| **Art**/Levels/ | Includes sprite sheets for animated objects, backgrounds, and tilesets. |
| **Scene**/ | Stores all the **.tscn** files created within Godot, which define the structure of various game scenes. |
| **Script/** | Contains all GDScript files for "Galactic Void." Each script is named corresponding to the scene it is associated with. |
| **Sound/** | Contains audio resources for the game. |
| **Sound/**Music | Stores the looping background music track. |
| **Sound/**Sound Effects | Holds all sound effect files used in the game. |
| *Highlighted rows indicate directories containing source code.* | |

# Executables

*Describe the executables that are a part of this project. If there are multiple executables break out each one into its own heading and give its name and a description of what function it plays in the system.*

### *EXE Short Name (EXEFileName.exe)*

*Description of the EXE and its usage in the project.*

### *EXE Short Name (EXEFileName.exe)*

*Description of the EXE and its usage in the project.*

There are currently no executables for this project. There are still certain aspects of the game that need to be addressed prior to its export.

# Code Architecture

## "Galactic Void" is built on a flexible and scalable code architecture thoughtfully designed to support the dynamic nature of 2D game development. This architecture uses a JSON-based save system and modular scene structure to improve gameplay through efficient data management and seamless scene transitions.

**Scene Management**

The game leverages Godot's powerful scene system, organizing gameplay elements into individual, reusable scenes. This modularity is critical for independent development and testing of game components.

**Scene Transitions**

* Transitioning from one game state to another, such as from the main menu to gameplay, is handled seamlessly, providing a smooth experience for the player. The main menu, once a standalone scene, is replaced by the GameGroup scene, which houses the gameplay mechanics.

**Scene Independence**

* Each scene is designed to function autonomously, enabling developers to run, test, and debug scenes independently as long as no external dependencies are required, with the exception of those from the global script. This independent scene structure facilitates a streamlined development and testing process.

**Global Script Role**

* The global script serves as the operational hub for "Galactic Void," overseeing the game state and mediating scene interactions. It retains the logic for scene changes and will eventually incorporate the logic for loading saved games. In addition to managing game pauses, it ensures the save directory is properly initialized at the game's commencement.

The architecture of "Galactic Void," with its emphasis on clear separation of concerns and the use of Godot's built-in systems, provides a foundational framework for the game's development and future growth.

## Database or Data Store

**Save System Logic**

The game features a save system integrated within the **save\_game.gd** script, which forms part of the GameGroup node hierarchy. This system captures and stores the game state, allowing players to save their progress and resume playing easily.

**Save Functionality**

* **Save Trigger and Feedback:** Through the in-game pause menu, players can save their current state at any time. A confirmation in the form of a "Game Saved" notification validates the successful action.
* **Data Captured:** Key player data, such as name, position, health, lives, and rotation, are serialized, **Figure 1**, along with enemy positions, thus providing a comprehensive snapshot of the game's current state. This data is structured into a dictionary and serialized to JSON format for storage.

A screen shot of a computer program

Description automatically generated

Figure save\_game.gd

* **File Handling:** Initially, data is saved in a non-encrypted format to facilitate debugging and can be seen in Figure 2. Future plans include encrypting the save file to bolster data security, utilizing a passkey managed by the separate global script. This file is stored in the app\_userdata file by default shown in Figure 3.

A screen shot of a computer

Description automatically generated

Figure Saved JSON file

A screenshot of a computer program

Description automatically generated

Figure Directory paths for user:// Source: Godot Docs. https://docs.godotengine.org/en/stable/tutorials/io/data\_paths.html

External Files & Data

There are currently no external files for Galactic Void.

Programming Language | GDScript

"Galactic Void" is developed primarily in GDScript, a dynamic, high-level programming language tailor-made for the Godot Engine. Its Python-like syntax makes it both accessible and efficient for game development, particularly in 2D environments like "Galactic Void"1.

**Key Components Developed in GDScript:**

* **Character Movement and Animation Control:** Scripts manage player actions, ensuring fluid motion and dynamic interactions within the 2D space1.
* **Enemy AI Scripts:** These scripts define behavior patterns and decision-making processes for non-player characters, which is important in a 2D platformer for creating challenging gameplay1.
* **Level Mechanics and Trigger Events:** The game's interactive 2D elements, such as traps, platforms, and environmental effects, are driven by scripts that respond to player actions and game dynamics1.
* **Collision and Physics Responses:** These scripts manage how game entities interact with the 2D environment, handling physics and responses to collisions1.
* **User Interface and Interaction Scripts:** They manage menus, heads-up displays (HUDs), and other interface elements, ensuring a seamless player experience in the 2D game world1.

**External Components and References:**

* **Godot Standard Libraries:** Utilized for their comprehensive functionalities in math, physics, and 2D rendering, enhancing the game's development1.
* **External Assets:** Incorporation of art assets, sounds, and music from third-party sources, integrated seamlessly into the game's 2D environment.
* **No External DLLs:** GDScript, combined with Godot's native tools, fulfills all core functionalities without the need for external DLLs. However, Godot's support for GDNative/DLLs is available for extending functionality if necessary1.

**Code Management and Dependencies:**

* **Scene Organization:** Each level and UI screen is constructed as a separate scene, reflecting the game's 2D structure, with scripts attached to relevant nodes within these scenes1.
* **Script Dependencies:** Minimizing tight coupling, scripts communicate through well-defined interfaces and signals. This approach is particularly effective in a 2D game where clear, organized script interactions are vital1.

The project's adherence to a clear separation of concerns and effective use of Godot's systems for organization and script communication ensures "Galactic Void" has a robust, scalable architecture. This structure is conducive to 2D game development, where ease of iteration and modular design are key1.

1 Godot (n.d.). GDScript: An introduction to dynamic languages. Godot Docs. <https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_advanced.html>

Project Classes

Classes within the project are used to abstract re-usable pieces of code. Classes are also used to group related values, known as properties.

Galactic Void" uses Godot's Object-Oriented principles, scene and script system as classes to encapsulate reusable code and related properties. The scene system is used to create modular classes while the script system defines their behavior. This approach improves code reusability and maintainability, facilitating the development and maintenance of complex game elements.

The project utilizes these classes:

### A Player class for direct user interaction | player.gd

This script extends CharacterBody2D, controlling the main player character. It handles player movement, jumping capabilities, attack mechanics, and damage reception. It also manages animations for various actions through an animation handler and signals for when the player takes damage or dies. The script utilizes variables for movement speed, jumping velocity, and game physics such as gravity.

### A base class for the enemy scenes | enemy.gd

This is the base script for enemy characters, extending CharacterBody2D. It includes properties like speed and health, handles enemy states (idle, chasing, attacking, taking damage, and death), and implements a simple state machine for behavior transitions. It allows for attacks with cooldowns and provides functionality to chase the player and take damage. The script is designed to be extended by specific enemy types for more specialized behaviors.

### A class that manages the overall game world and extends Node | game\_world.gd

This class handles the player's save function, reset mechanics when the player enters a death zone, and the display of the saved game notification. It controls the main game flow, including player resurrection and initial setup.

### A class that manages the pause menu functionality | pause\_menu.gd

Extending Control, this script controls the game's pause menu, including showing and hiding settings, game controls, quit confirmation popups, and managing audio feedback with button interactions. It ensures the game can be paused, settings adjusted, and game controls accessed or the game quit as needed.

### A class allows the player to adjust different sound volumes| volume\_slider.gd

The volume\_slider.gd script extends HSlider and is responsible for the audio settings in the game. It adjusts the game's volume levels through the Godot AudioServer. It handles the conversion between decibels and linear volume for the user interface slider and applies these settings to the audio bus for real-time audio level adjustments. It is also set up to adjust the sliders to the current volume when the scene is called to the scene tree.

### A class plays sound effects for the player and enemies| sfx\_audio.gd

This script extends Node and acts as a centralized sound effect manager for "Galactic Void" while also playing an uninterrupted loop of the background music. It preloads various sound effect resources, such as player actions (jump, double jump, attack, damage) and enemy actions (attacks and damage responses). The script contains functions to play specific sound effects for player and enemy events. When a sound effect is requested, it dynamically creates an AudioStreamPlayer node, sets the appropriate sound stream, and plays it on the "sfx" audio bus. After the sound finishes playing, it ensures the AudioStreamPlayer is freed from memory. This setup provides an efficient way to manage and play sound effects throughout the game.

Project Modules

"Galactic Void" leverages a streamlined approach to its module system, primarily focusing on audio management and global game settings. These modules contain procedural code and are utilized across the game without maintaining internal state, providing essential functionalities that are crucial to the game’s dynamics.

### A scene that manages the background music and sound effects | sfx\_audio.gd

This module centralizes sound effect management within the game. It dynamically handles the playback of various audio clips associated with player and enemy actions, allowing for a flexible and efficient sound design that can be easily expanded or modified.

### A global script that manages the game state | ModuleFileName3.cls

Serving as the game’s central hub, this module manages cross-cutting concerns such as scene transitions, game state management, and other core functionalities that need to be accessed globally. It ensures consistency and provides a backbone for the game’s logic and flow.

Program Start and End Flow

A player is presented with the main menu and if they have a saved file, can load their game. If not, they can start a new one. The correct scene is then loaded into the viewpoert and a player can navigate the level. During the level play, they come into contact with enemies and if they are hit, they take damage. If a player falls off the map, they touch the deathzone and are sent back to the beginning of the level. When a player takes damage, there is a check to see if their lives and health are less than zero, if so, then game over.

A diagram of a game

Description automatically generated

Summary

Briefly summarize the system documentation here. This section should be no longer than 1/3-1/2 of the page.

# APPENDIX B (BUILD AND RELEASE PROCESS)

Describe the build & release process required to implement an update.

# APPENDIX C (CLIENT INSTALLATION INSTRUCTIONS)

Detail how a client machine or device is prepared to utilize the project.

# APPENDIX D (DEVELOPER SETUP INSTRUCTIONS)

This project was developed in Godot, version 4.1.1, using GDScript.

Go to <https://godotengine.org> and navigate to the Download section.

Once there, navigate to the “download archive” section for previous releases.

A screenshot of a video game

Description automatically generated

Look for the 4.1.1 Stable release date **17 July 2023**

A screenshot of a computer

Description automatically generated

Once the program is downloaded, import the project.godot file and import that file to Godot.

A screenshot of a computer

Description automatically generated