Inland Norway
University of
Applied Sciences

Faculty of Audiovisual Media and Creative Technologies

Marcus Nesvik Henriksen, Stål Brændeland, Anders Petershagen Åsbø

Bachelor Project Report

‘Stranded: Astral Odyssey’:
On freedom of movement and game feel

‘Stranded: Astral Odyssey’: Om bevegelsesfrihet og spillfølelse

Game technology and simulation

SPIS2900

2024

Preface

Acknowledgements

The authors of this paper would like to extend a special thanks to our supervisor Jan Erik Haug for providing feedback on this report and assisting us throughout the development period.

Additionally, we extend our gratitude to the members of group 8, whose artistic collaboration helped bring the project to life. Their creative direction and distinct art style significantly enriched the visual appeal of our game, highlighting their essential role in its development.

Finally, a special thanks is extended to Einar Johannes Lindberg Birkeland for his advice on performance optimization in Unreal Engine, and Jakob Irian Grønbeck for his writing collaboration on story and lore.

Introduction

One of the most important aspects of modern game design is creating a satisfying game feel through the application of game mechanics. In this report we will detail our exploration into game feel through the production of a third-person platformer focused on freedom of movement and exploration as outlined in the project description document (Henriksen, Åsbø, et al., 2024). We chose to name our game ‘Stranded: Astral Odyssey’.

The Stranded: Astral Odyssey project is a collaboration between two bachelor groups. This report is written by the group consisting of programmers from the Game technology and simulations bachelor programme, while the second group consists of the artists Jan-Roar Tronvik Finseth, Viktoria Carlsen and Adele Wallace from the Animation and digital art bachelor programme. Thus, this report mainly focuses on the technical and mechanical aspects of the project, and the resulting product.

In addition to the project description document, we created a high-concept game design document (Henriksen, Brændeland, et al., 2024) to serve as a general reference for the prototype game we wished to develop. In the GDD we define our idea for the game as “A

third person 3D platformer with a jetpack/hovering mechanic [and fluid] movement. Genres: Adventure, Exploration, Metroidvania, Puzzle." (Henriksen, Brændeland, et al., 2024, p. 1). Both the project description document, and game design document have been handed in alongside this report as stand alone documents.

Problem statement

In the previous section, we highlighted the intent with the prototype itself. In this section we define the purpose of this paper with the following problem statement.

How can we create a satisfying gameplay experience and game feel, in accordance with Steve Swinks's definition (Swink, 2009, p. 6), utilizing player control and interaction with movement mechanics and other game systems?

The aim of this paper is to answer the aforementioned problem statement by documenting our production and development process, as well as the systems and mechanics we chose to implement. Furthermore, we aim to justify our development choices in the context of the problem statement.

Throughout this paper, the player will be frequently referenced, as central to our game development decision-making process. Jesse Schell emphasizes the significance of understanding the player's desires and preferences, illustrated with an anecdote about Einstein: "As much as he loved thinking and talking about physics, he knew that it wasn't something that his audience would be really interested in." (Schell, 2015, p. 116) As developers, we must be careful not to fall into the trap of trying to make players engage with the game strictly as we intended. Taking inspiration from Schell's insights, we aimed to prioritize player experience by actively listening to player feedback, adapting gameplay elements to resonate with their interests, and creating an engaging and open experience that allows players to explore and play the game according to their preferences.

Defining game feel

Central to our project is the idea of game feel, thus to define the term, we looked to Steve Swink's 'Game Feel: A game designer's guide to virtual sensation' (Swink, 2009). As

outlined by Swink, game feel is an aggregate experience built on several aspects of the game mechanics, game visuals, and controls (Swink, 2009, pp. 2–6).

The first aspect of game feel is real-time control, which Swink defines as the interaction between player and the game through precise and continuous control over a player character (Swink, 2009, pp. 3–4). The player makes repeated inputs to the game, which changes its internal state in reaction to the given input, and conveys the result of the input through audio, video and haptic feedback (Swink, 2009, pp. 2–4). Once the player senses the feedback, they process it and choose new input as the cycle repeats. The cycle should ideally be continuous, to not force a stop in player interaction (Swink, 2009, p. 26). Furthermore, the cycle should be precise so that the player maintains adequate control over the player character to achieve the goals of the game (Swink, 2009, p. 4).

Swink makes a distinction between games that do not fall under the definition of having real-time control, like StarCraft where each instance of control is a separate blip, in other words they are not continuous, and a third person game like Super Mario 64 where control is constant and uninterrupted (Swink, 2009, pp. 7–9). Stranded: Astral Odyssey is a game that falls into the latter category. In the game each input is processed and reflected in the player character on the screen instantly, and this process is continuous and uninterrupted, excluding when the player dies.

In order to display the results of the real-time control aspect of game feel, Swink posits that the aspect of a simulated space is needed (Swink, 2009, pp. 4–5). A simulated space is the constructed representation of a world in which the player character lives and moves. The simulated space consists of the visual models, terrain and effects that the player character moves through and interacts with, as well as the interactive systems like collisions and gravity, so that the player is given a reference frame in which to move their player character (Swink, 2009, pp. 4–5). Furthermore, the simulated space gives context to the feedback provided by the game, and provides a sense of familiarity between the game and the real world.

The final aspect of game feel, that Swinke puts forth, is polish. According to Swink, polish refers to cosmetic effects that enhance a player's experience of interactions without changing how the interaction is simulated (Swink, 2009, pp. 5–6). An example would be the flaming

exhaust of a jetpack thruster, or a gamepad vibrating when the player character takes damage. Together with real-time control and simulated space, these three aspects combine to create a definition of game feel that Swink formulates as “Real-time control of virtual objects in a simulated space, with interactions emphasized by polish.” (Swink, 2009, p. 6).

Methodology

Team structure

The team structure was a result of planning over time as well as the needs of the project. The general concept of the game was put together by the group leader before any team members were assembled, thus the required team size could be estimated beforehand. Due to the size and ambitiousness of the original concept it was known that a larger number of artists was required. It was decided to balance out on the programming side as well, which in hindsight ended up benefiting us.

The collective group knowledge spans wide, and members with specialized knowledge in their respective areas were chosen to avoid excessive overlapping of competence, as well as to cover the basic requirements for building a game. The main requirements that were thought of when choosing team members were world building, foliage, animation, texturing, movement programming and systems programming. That way we could have a basic grasp on what we needed to do and start developing a prototype, without first having to learn everything from scratch.

The team consisted of the three programmers from group 7: Marcus Nesvik Henriksen, Stål Brændeland and Anders Petershagen Åsbø, who are the authors of this paper. Furthermore, the team included the three artists from group 8: Adele Wallace, Jan-Roar Tronvik Finseth and Viktoria Carlsen, who wrote their own report.

Agile development approach

It was known that a managed work structure was needed to successfully organize a group of this size. During the pre-production phase, a considerable amount of time was spent by the

project manager researching different methodologies for work management. This was also driven by the desire to build proper competence in project management tools.

Agile Scrum is a project management framework used in software development. It emphasizes flexibility and iterative development through cycles called *sprints*. The agile scrum method allows for adaptive planning and continuous improvement based on feedback. The decision to adopt the agile scrum framework for our project was driven by two key factors. Firstly, scrum is well-suited for game development and widely embraced within the software and gaming industries for its effectiveness in managing complex projects (Kristiadi et al., 2019). Secondly, we aimed to gain practical skills for our post-bachelor's careers.

For project management and task planning, we opted to use Jira, due to its established reputation and use in the industry (SimilarTech, 2024). Using agile scrum on Jira enabled us to break down tasks into multiple manageable subtasks, which then could be assigned to different team members. It also enabled us to easily manage large numbers of tasks and organize them into sprints.

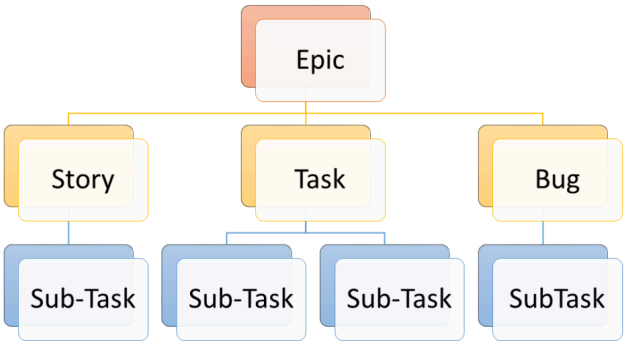

Figure 1: Hierarchical task structure on Jira. Each level can contain multiple subtasks to break down complex problems further. Tasks are divided into categories to improve readability.

We combined our sprint interval with our physical meetings, such that we could discuss task progress as well as set up the next sprint when we had physical meetings. During the pre-production phase we opted to have one bi-weekly physical meeting, due to the overhead of other subjects. In the production phase we increased the interval to once per week. We experienced that lack of physical meetings resulted in team instability and miscommunication, adding an extra physical meeting per week mitigated this issue. Additionally we added one digital stand-up meeting per week. Thus we met a total of two times per week.

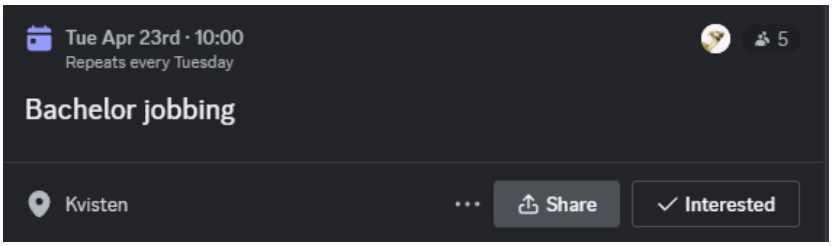

Figure 2: We leveraged Discord's (Discord Inc., 2024) event planner to schedule our meetings. This event represented a weekly physical work session.

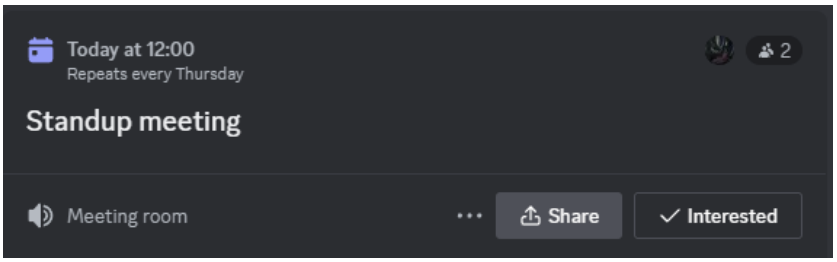

Figure 3: Weekly digital stand-up meeting over Discord.

The digital stand-up meetings followed the guidelines outlined by the Project Management Institute in the seventh edition of the PMBOK Guide for daily scrum meetings (Project Management Institute, 2021, sec. Glossary). This approach involved a concise session where team members reviewed progress since the last meeting, shared plans for upcoming work and identified obstacles, encountered or anticipated. We found that having meetings twice per week was enough to satisfy our needs for communication and collaborative planning.

Based on feedback from our supervisor and our meeting experiences, we learned the importance of keeping our meetings short, typically lasting 15-20 minutes. During these meetings, each member would provide an update on their achievements since the last meeting. This practice not only served as a progress indicator for the project manager and team members but also motivated individuals to accomplish tasks, in order to have something to show for the stand-up discussions.

Following the stand-up meetings, team members often engaged in breakout sessions where they would remain for further discussions, or join separate video calls to discuss specific implementation details. This post-meeting interaction is sometimes referred to as the "after party" in Agile methodology (Flewelling, 2018).

Concept timeline

The original concept

The original concept for the project was a large open world exploration game, where the player could switch their movement mode between a fast moving hover-vehicle, and slower but more nimble bipedal character. Thus the core mechanic of the game was the ability to switch quickly, more or less at will between the two modes of play.

The original concept drew inspiration from the game Hogwarts Legacy (Tew, 2023), specifically in the movement mode transition mechanics. In Hogwarts Legacy, the player unlocks a magical broom on which they can fly. Switching from on-foot mode to flying mode triggers a swift and seamless transition, which enhances gameplay by eliminating the disruption of animation sequences. Inspired by this, we attempted to develop a similar mechanic with a futuristic hover-vehicle and a character.

The premise of the story for the concept, was that of an interstellar explorer stranded on an alien world, where they would have to traverse the environment using the core mechanic in order to retrieve important spaceship parts from wreckage scattered about the open world, and use it to repair their spaceship and escape the planet. This story would serve as a guiding prompt for the player to explore the game world.

To challenge the player, the team planned to develop environmental puzzles that required the utilization of the core mechanics in order to get to the locations in which the spaceship parts could be collected. Such as a canyon filled with spikes, where the player must use their bipedal mode to reach a trigger which retracts the spikes for some duration, then use the hover-vehicle mode to quickly race across the canyon before the spikes extend again and harm the player character.

Second iteration

Midway in the pre-production stage on suggestion from one of the artists, both groups decided that there were a few problems with the original concept that meant it would be hard to develop into a full-fledged game in the time period allotted. Most importantly, an open-world game simply meant too much empty space that needed to be filled with content to

prevent the game from feeling empty or boring. It was decided that it would not be realistic to produce that many assets, or code that many encounters.

To solve this we decided to instead create a smaller game taking place on a series of floating islands. This allowed us to maintain most of the big picture story elements. The player still has a crash landing on an alien environment, but on a group of floating islands above the planet instead of on the planet surface (Henriksen, Åsbø, et al., 2024, p. 3). This still allowed the player to explore freely, but eased the development and planning of creating concrete content. This way we could clearly define workloads in manageable batches - one island at a time, and create more of these if we had time left over after creating the initial islands.

The second problem with the open-world concept was the main mechanic we wanted to captivate the player with: Transforming between a biped and a hovering vehicle would end up sidelined. Focus on the satisfying transition between the two modes of movement would be taken away, as we realized the player would spend most of their time traveling from point A to a target point B in hover-mode. The player would seldomly transform to bipedal mode when they needed to interact with the world to fulfill objectives.

Instead, what we wanted was for this main mechanic that was going to enable satisfying movement to be something the player would use all the time, integrated into the main gameplay loop. So in the updated idea we settled on something simpler, a jetpack (Henriksen, Åsbø, et al., 2024, p. 3). Not just a conventional jetpack that would allow the player to gain height, but one that would smoothly integrate with normal movement for a fluid experience.

Narrative and storyline

As mentioned in the concept section, the narrative is centered around an interstellar explorer stranded on an alien world. The focus lies on the player character having to collect wreckage from their crashed ship in order to construct an escape vehicle and leave the planet.

We took inspiration from Schell’s Lens of the Obstacle, where he mentions “A goal with no obstacles is not worth pursuing.” (Schell, 2015, p. 305). Crash-landing on the planet serves as a central obstacle, while the level design further challenges players' movement capabilities

with floating islands, traps, and turrets that facilitate the use of extended movement mechanics for evasion.

As a method of tying the game world into the narrative, certain concepts are introduced in sync with the player learning about them through story elements. For example, the first spike trap. The game operates in a linear manner, with the player exploring the islands one at a time, and this is an in-universe concept, meaning it was true for the previous explorer as well. So as the player follows in the footsteps of the previous explorer and comes along their belongings, soon after a blood splatter can be spotted next to a nearby spike trap, the first of its kind.

One concept we especially kept in mind when doing narrative design was Schell's 'Lens #4: The Lens of Curiosity'. In this part of the book Schell wants us to think about what questions we are putting into the player's mind (Schell, 2015, p. 30). One way of approaching this problem is limiting the amount of information available to the player, forcing them to think for themselves if they want to figure out what's going on. We give the player very little in the way of lore for just following the main quest, instead sprinkling in information naturally by exposition.

When exploring the player can come across remains of the previous explorer, triggering questions about their fate. They can also find pickups in the world in the form of data caches. Data caches are objects containing log recordings from the previous explorer, detailing their experiences and theories about the phenomenon of the floating islands. A second benefit of this more dynamic approach for delivering the story is a more natural form of "show, dont tell", where the player can take in the story without having to listen to minutes of expository dialogue or read multiple books.

Figure 4: A data cache lies hidden away from the main path, but still blinking and signaling its presence with a faint sound cue.

The data cache pickups don't give an outright explanation of the story either, as they are from the perspective of the previous explorer. Nowhere in the game is the full story as the developers intended stated outright in its entirety, but the player can figure it out from context clues in the world, and by reading the log recordings they collect. Ultimately, piecing together the story allows them to solve the final obstacle by destroying the source of the gravitational phenomenon, and so they can safely escape and finish the game.

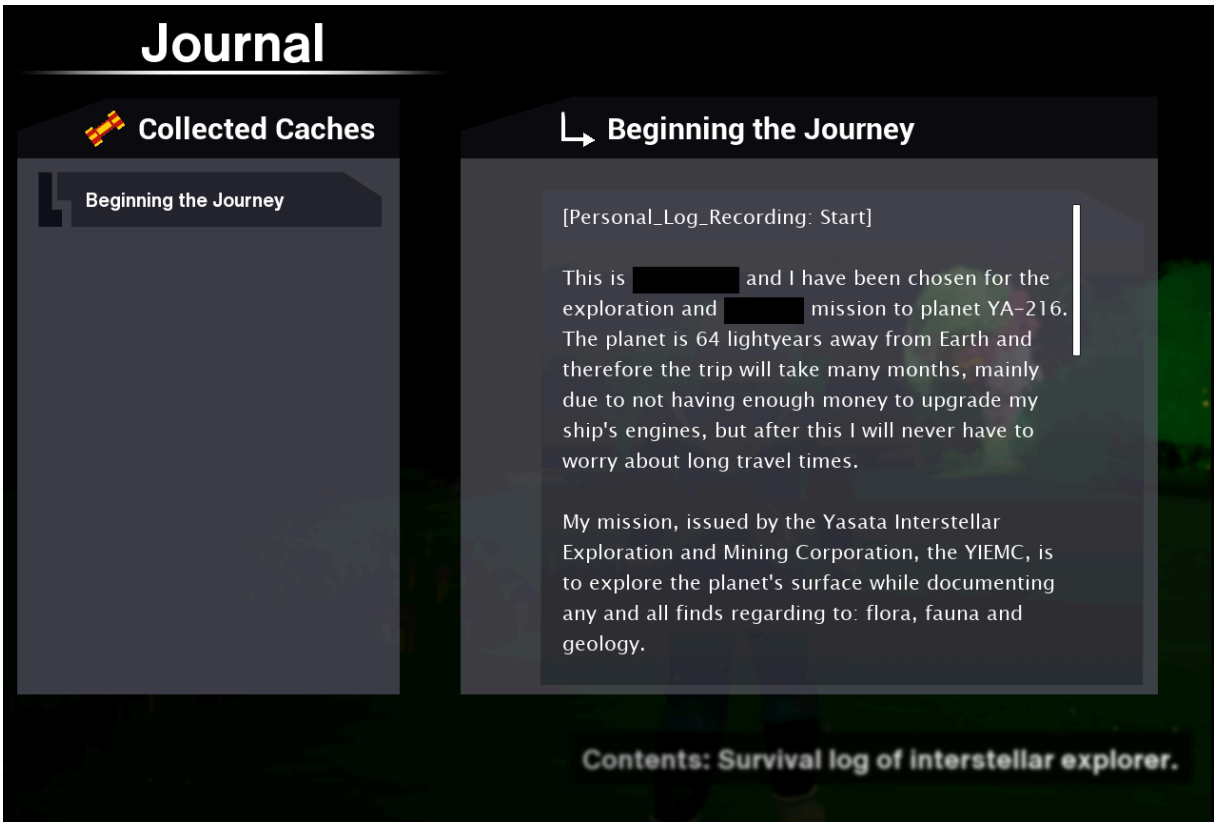

Figure 5: Journal menu. Collecting data caches progressively unlocks entries in the Journal, where the player can go back and look at them any time.

Furthermore, in order to feed the player necessary prompts and info, without continuously breaking the fourth wall, it was decided to have an in-narrative AI assistant belonging to the player character. This AI assistant would serve as a second character feeding the player information through dialogue, thus making tutorials and explanations of game mechanics diegetic.

For instance, we leverage the AI assistant to introduce new quests to the player through dialogue that contextualizes objectives within the storyline. In this way, the dialogue not only informs the player of their tasks, but also enhances immersion by aligning with the game's narrative theme. Additionally, we use the AI assistant to guide the player through different game mechanics. For example, on certain quests the assistant will mention that a marker has been placed on the heads-up display. Another method of guiding the player involves strategically placing trigger volumes between floating islands. If the player dies within one of these volumes, the AI might suggest, "Gear upgrade might be required to reach a new area.".

Through playtesting, we learned the importance of balancing guidance with player autonomy. Rather than forcing the player to adhere to the narrative advice, we apply subtle hints and reminders at strategic intervals, to assist without disrupting the player's flow state. "Distractions steal focus from our task. No focus, no flow." (Schell, 2015, p. 139) The goal with this is to allow the player to engage with the game's mechanics at their own pace, allowing for a sense of exploration and discovery.

Tools used

Unreal Engine

In this section we will discuss why we chose Unreal Engine 5.3.2 (Epic Games, 2023) as our game engine and how it enabled us to develop our project. Our initial concept called for a vast open world, and Unreal's existing functionality for this, 'World Partition', made it an ideal choice. To achieve our ambitious goals we would need tools like: procedural content generation, an existing character movement framework, and cinematic tools. All tools which Unreal Engine has right out of the box. Additionally, the collective experience of the programmers with the engine played a significant role. Lastly, Unreal Engine empowers greater artistic freedom, enabling the artists to develop a compelling art style and visual appeal for our game.

Unreal Utility

Unreal Utility (Henriksen, 2024) is a custom file management tool made specifically for this project. It was developed in Go, a lightweight programming language that compiles to a single standalone .exe file. Its purpose is to quickly delete temporary files, as well as generate Unreal Engine project files and compile the project. This is a process that usually takes about half a minute, however it is required frequently enough to take meaningful time out of the day. Unreal Utility was also developed to make this process effortless for the artist group, who are not as technically inclined in the subject. Throughout the development period the program has saved hours of development time.

Implementation

Model-view-controller User Interface

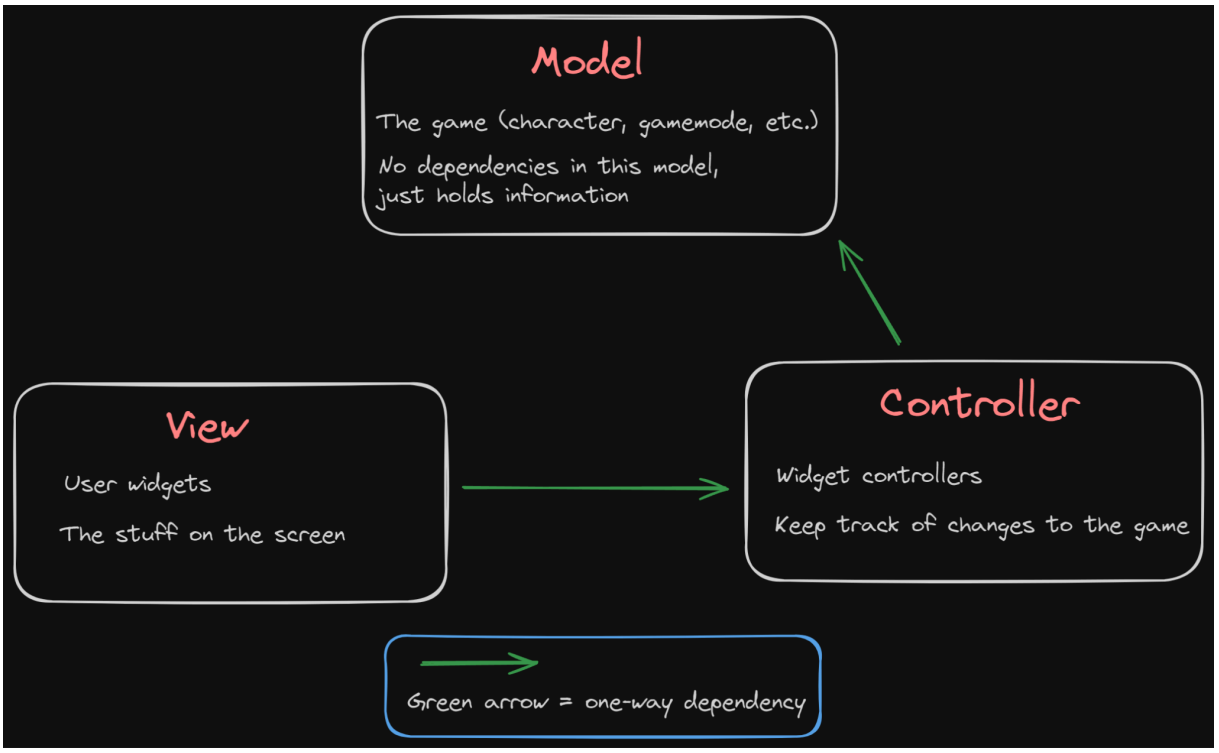

Figure 6: Illustration of model-view-controller pattern as used in our project.

When choosing a structure for the user interface C++ classes, the team landed on the Model-View-Controller design pattern. The idea is to separate the game as a whole into three layers, with two sets of one-way dependencies.

The simplest layer is the part of the game that is detached from the UI. Described here as the model, it's a catch-all classification for all the data we could want to retrieve from the game, like variables or events from the gamemode or character. The controller layer consists of widget controllers, classes that hold references to things in the game, and events to broadcast changes in the game to all their listeners. The listeners in this case are all the user widgets, making up the view layer.

The advantage of such a system compared with earlier interface designs where these layers were bundled together, is the added flexibility and reusability that comes with decoupling (Gamma et al., 1995).

For our use case this system was probably outside the scope of what was needed. Though present in the project, large parts of the UI does not rely on the framework and uses a more integrated two-layer system where the game acts as the model and controller combined, and communicates information straight to the view/UI.

Save system

A vital aspect of game progression is the ability to save the current state of the game, and load it at a later date, so the player does not have to restart the game every time they sit down to play. As such we implemented a flexible save system for storing game data to file.

The general idea of the system is to have a single instance of a save system manager class that handles the flow of data between game objects and disk. The save system manager also implements all methods for communication with, and usage from other systems.

As shown in the class diagram, the save system manager stores a reference to all objects that implement the ISavableObject interface. When the SaveGame function in the manager is called, it will instantiate a SaveGameData object which is then passed to each managed object. The managed object then stores all relevant data from itself into the saveGameData object. Once all data has been collected, the saveGameData object is asynchronously written to file in parallel with the game logic continuing to run.

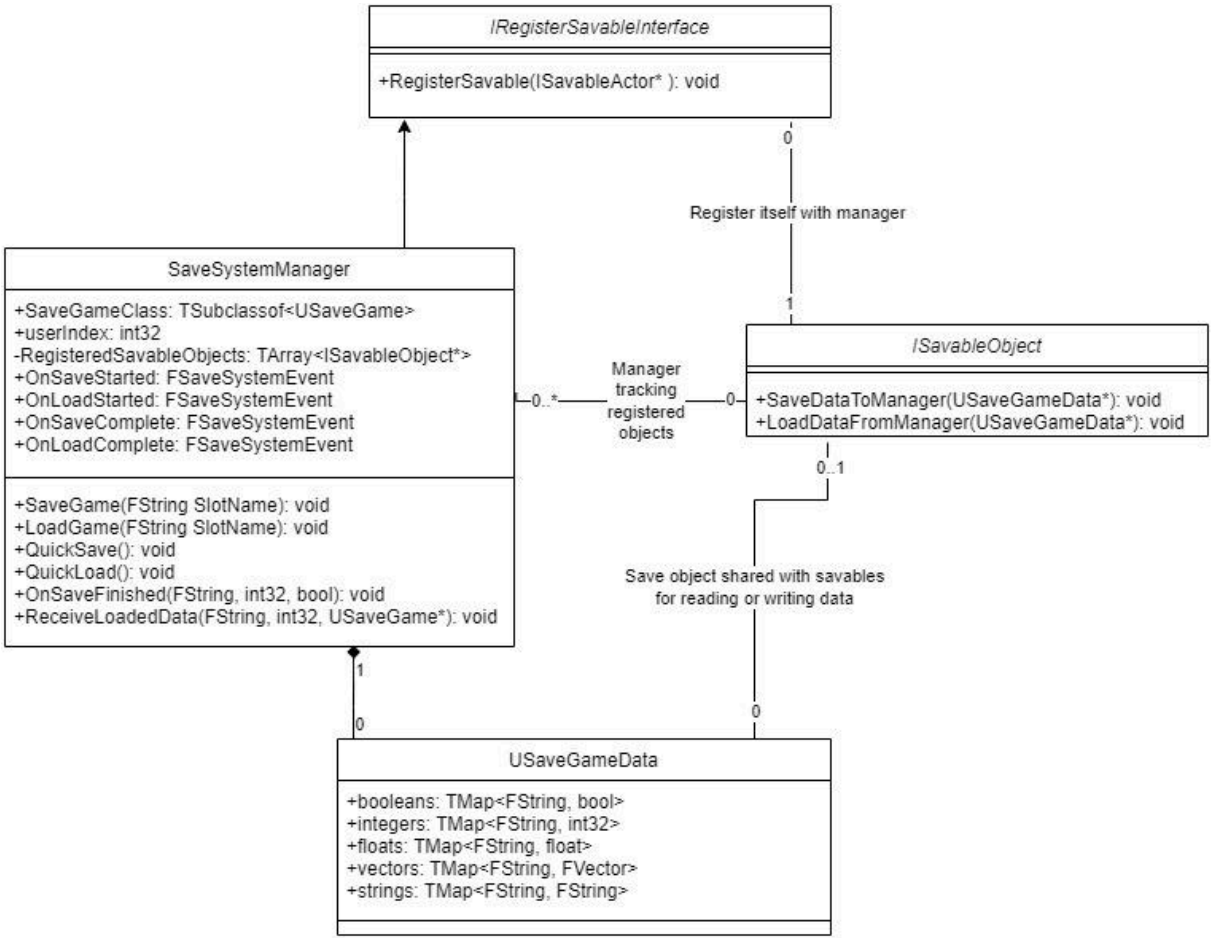

Figure 7: Class diagram showing interconnectedness of save system. Only methods and fields relevant to game state saving are included.

The LoadGame method is effectively a reversal of the SaveGame method, which starts an asynchronous read from file. Once the save data has been loaded into a saveGameData object, it calls a delegate in the manager which passes the data object sequentially to all managed objects, so they can set their internal state with the loaded data.
The save system manager has four events that other classes can listen to that fire when a save is started, when a save has completed, when a load has started, and when a load has completed.

The use of asynchronous programming for the save and load process, was made to minimize the time spent interrupting gameplay logic from running, thus preventing the player's experience of the game to become disjointed by periods of waiting. Such interruptions risk lowering the player's immersion and enjoyment of the game, breaking them out of flow-state

by offering the possibility of distractions from the game, and preventing the direct feedback that gameplay provides (Schell, 2015, p. 139).

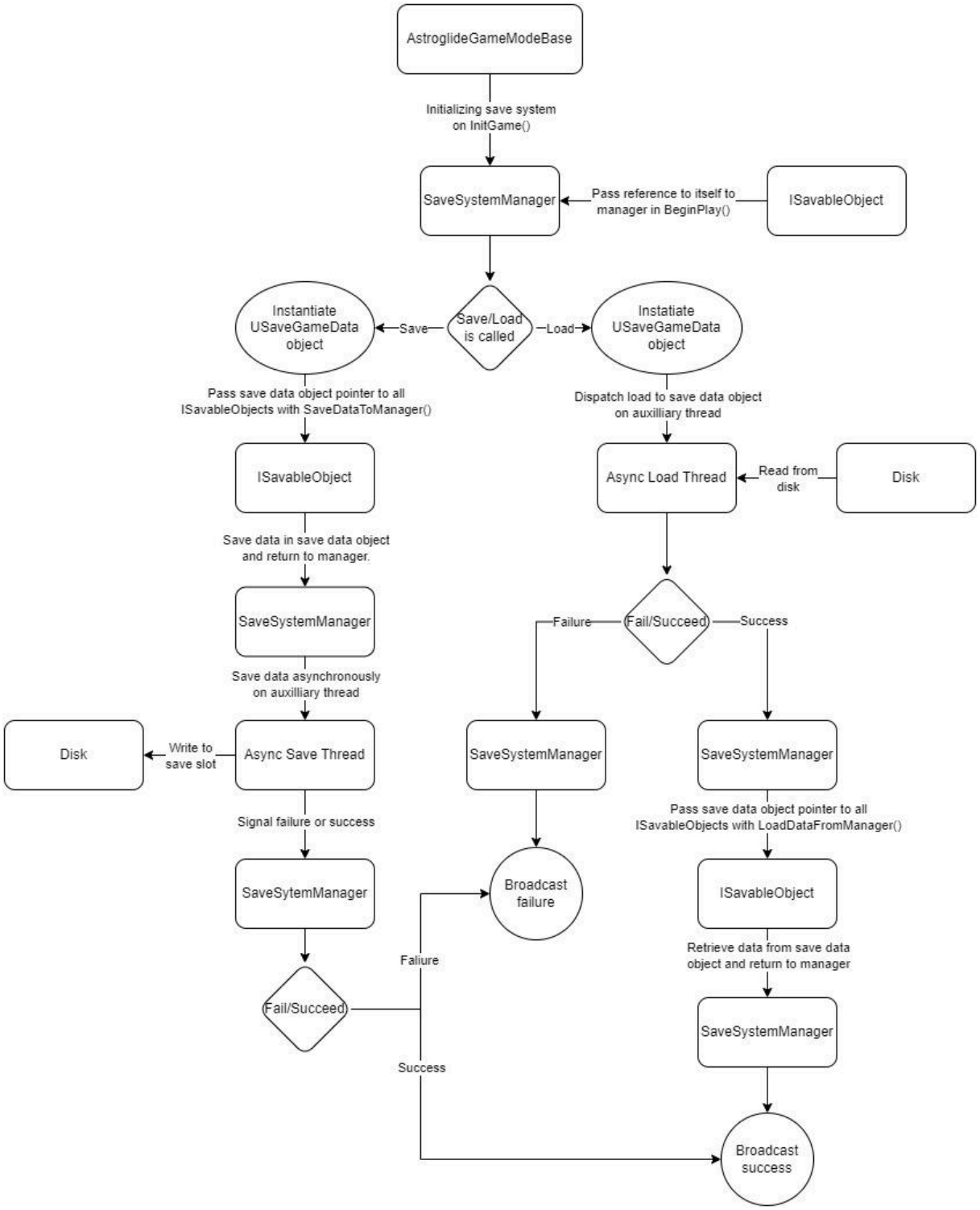

Figure: Diagram of program flow for save system. Shows initialization, as well as saving and loading of game state.

Furthermore, by keeping the delay caused by saving/loading within the human mean visual reaction time of 331 ms (Shelton & Kumar, 2010), the player can resume the game play experience before processing that a pause has taken place. This allows for continuous real-time control needed to sustain game feel (Swink, 2009, pp. 45–47). Such considerations of course, are only applicable in the case an automated save or load is performed during game play. Manual saving triggered by the player requires pausing the game experience and navigating to the save menu, and therefore depend less on maintaining a sense of continuity.

Quest system

In order to give the player a goal, and a direction to aim their playthrough, we decided to have a simple quest system. A quest system serves the game experience by defining concrete objectives that the player can strive to complete, thus providing the clear goals necessary to induce a flow-state in the player according to Schell (Schell, 2015, p. 139).

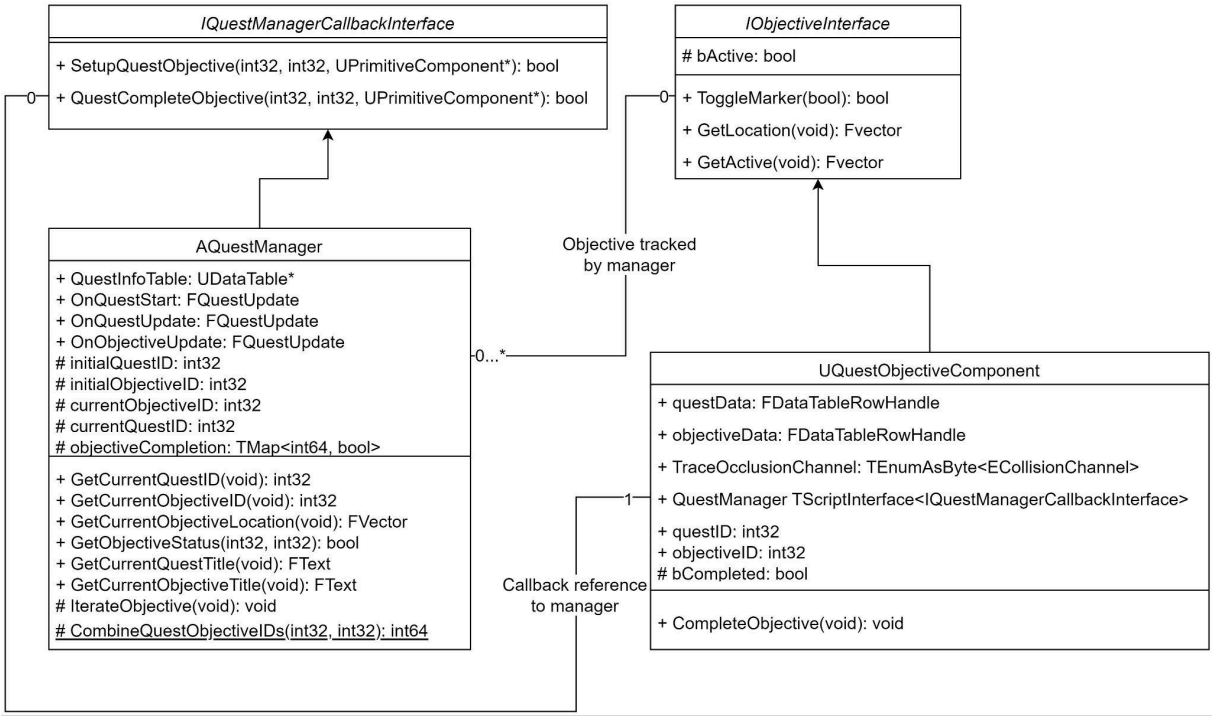

Figure 8: Class diagram for quest system.

The quest system relies on a manager class to handle quest logging, and quest progression. It is the main class that other systems interact with, providing methods for getting info about the current quest and objective.

All static data about quests, such as their names and descriptions are known at compile time, thus we elected to store them in Unreal Engine's provided data table structures (Epic Games, 2024b). The data is stored in a DT_QuestTable, where each row corresponds to a quest, and each column to a data field. Each quest is uniquely identified by a 32-bit integer serving as the quest ID of the form 1000X, where 'X' is a number denoting the quests position in the overall line of quests.

Each row in the quest table, also stores a reference to a DT_[QuestID]_ObjectivesTable, which holds static data about the objectives of the corresponding quest. As with the quest table, each objectives table holds objectives as rows of data, and a unique objective ID of the form A000B, where 'A' corresponds to the numbering of the parent quest, and 'B' is the objectives position in that quest's line of objectives. Thus, the fourth quest would have ID 10004, and its second objective would have ID 40002. From this, each objective tracked by the quest manager class is uniquely identified by a combination of their quest ID, and their objective ID.

The quest manager stores all existing objectives in a hashtable. However, Unreal Engine's TMap implementation of hashtable does not support composite keys. Thus we chose to make each key a 64-bit integer, where the 32 most significant bits are the quest ID, and the 32 least significant bits are the objective ID of said objective.

```
int64 AQuestManager::CombineQuestObjectiveIDs(const int32 questID, const int32
objectiveID) {
   // convert to 64-bit integers:
   const auto q64 = static_cast<int64>(questID);
   const auto o64 = static_cast<int64>(objectiveID);

   // bit-shift questID, then combine with bitwise or.
   return q64 << 32 | o64;
}
```

Figure 9: Code snippet of bit-manipulation used to create composite keys for hashtable.

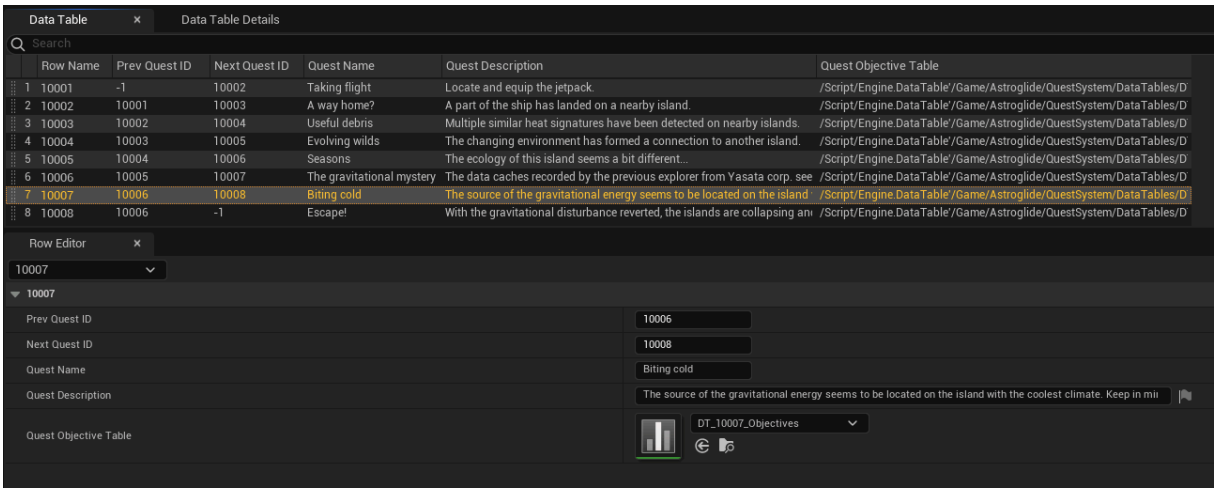

Figure 10: Data table holding static information about quests.

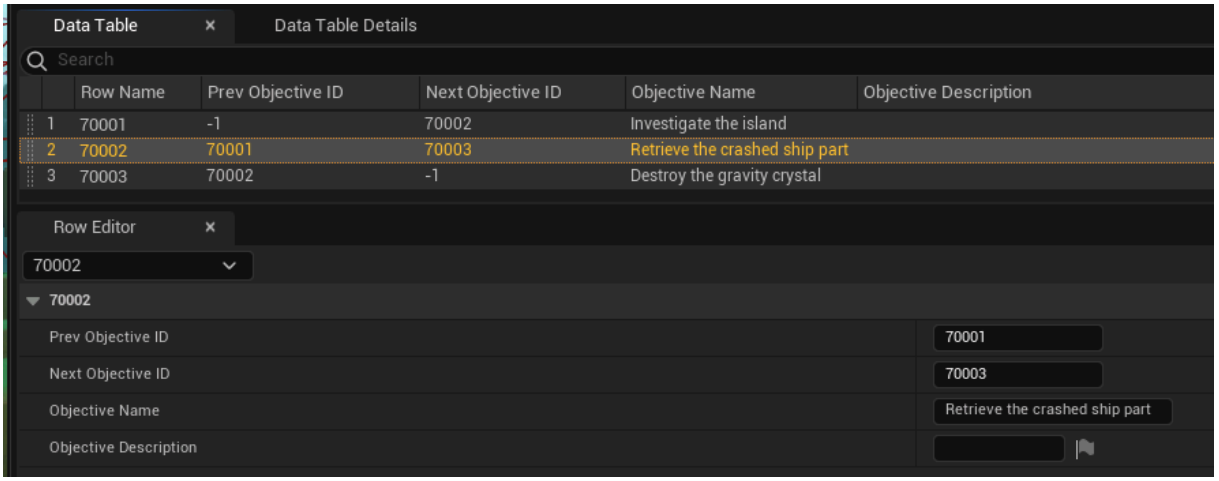

Figure 11: Data table holding static information about a given quest's objectives.

Figure 12: Quest objective marker. Viewed at location of objective. Left: translucent when the quest-panel is closed. Right: Pulsing with ring-effect when the quest-panel is open.

Figure 13: Quest objective marker attaches to edge of screen when objective is outside camera view.

Quest objectives must always be tied to specific gameobjects, by attaching a UQuestObjectiveComponent. Each objective component is assigned the rows in the quest and objective tables that correspond to the objective we want it to represent. Furthermore, this allows us to tie the objective to a physical location that can be indicated to the player using a visual marker.

The choice to utilize a visual marker was made to help the player from getting lost in the simulated space, while allowing for the player to stray from the path of progression. This prevents player frustration that could break them out of their flow state (Schell, 2015, p. 142). We also elected to keep the quest marker minimal in its visual language, both to not intrude on the player while exploring, and to avoid coercing the player to only chase objectives. However, to allow better visibility when the player decides, we chose to make the marker pulse when the quest panel is open.

Upgrade system

In ‘Players making decisions’ Hiwiller stresses the importance of the player having agency, so that they can have a meaningful decision-making experience (Hiwiller, 2016, p. 102). Due to time constraints and the resulting linearity of the quest system the player does not necessarily get a lot of choice in where to go at any given time, so when developing the game a conscious effort was made to make sure the player had choice in exactly how they progressed through the game.
This shows not only that our diverse movement system allows the player to move around in different ways, but also the branching tree structure of the upgrade system. At various points throughout the game the player is rewarded with Upgrade Schematics, which can be used to

enhance the player's movement options, or even unlock new modes of movement. These upgrades are unlocked from a tree-like structure, meaning different players can branch out in different directions depending on their preferred playstyle.

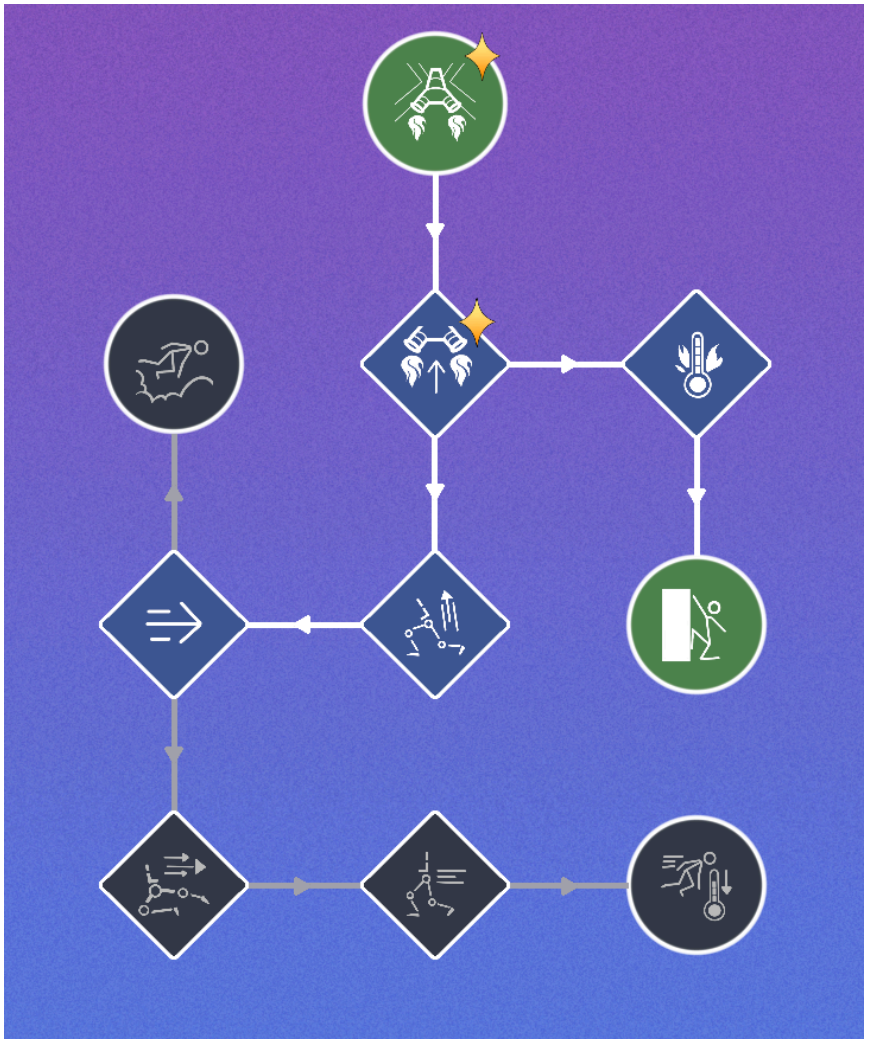

Figure 14: Upgrade tree Circles represent abilities (game changing abilities or new ways to move), diamonds represent upgrades (passive modifiers that enhance the existing movement). Each upgrade can be unlocked up to three times, each level increasing its effect. Purchasing one level in an upgrade unlocks the next one down the tree, and a fully unlocked upgrade or ability is marked with a star icon.

The game is not made so a specific upgrade order is required to clear certain obstacles, rather they aim to enhance the experience in different ways. There are also optional collectables in the world, hidden away from the main questline, that adventurous players can collect to get extra Upgrade Schematics. Such a player would be rewarded with the ability to eventually unlock every single upgrade, an extra completionist achievement that is not mandatory.

A possible pitfall of expansive options and choices in video games are so called blind choices. A blind choice is one the player is ill-equipped to make, because they lack information about what benefit such a choice might bring. Even if they know the effects of the choice, they might not be able to put it in context of the rest of the game (Hiwiller, 2016, p. 105). To avoid this problem in Stranded: Astral Odyssey, the player gradually receives upgrade points as they explore the map and get familiar with the movement mechanics. The hope is that the player may even begin to naturally wish for some of the available upgrades before they have the ability to acquire them, creating incentive.

User interface implementation

Common UI

We wanted to make our game as accessible as possible, which is part of the reason we chose to use Unreal Engine's Common UI (Epic Games, 2024a). Common UI is a plugin developed by Epic Games for Fortnite, and implements extra functionality for navigating UI elements. We wanted our game to support both controller input and standard mouse & keyboard. By default, Unreal Engine does not implement cardinal navigation in the UI. Cardinal navigation can be seen as "implicit" navigation, for example in a menu of vertically stacked buttons, you would expect to select the next button below when pressing the down key, or press the button when clicking 'A' on your Xbox controller.

Another reason Common UI was chosen was for Activation support. All widgets deriving from the Common UI implemented class CommonActivatableWidget are pooled in memory, such that they persist in memory and need not be instantiated and destroyed upon viewing/hiding. Widgets that are activated support input bindings, meaning we can bind certain keyboard or controller buttons to run functionality in our widget. An example where this would be useful is in a settings menu, where you could press TAB to apply your settings. In addition to this Common UI has classes for visualizing these actions, for example the CommonActionBar. This widget, once initialized, will display the keys needed to perform said actions. Often you will see this in games at the bottom of your screen.

Although Common UI has many time-saving functions, the learning curve was very steep. Unreal Engine already lacks in documentation, and the Common UI plugin is known for

having very basic, surface level documentation and little community content. Due to this, a sizable amount of time was spent researching how to use the plugin and implementing a base system for utilizing Common UI widgets. Lastly, Common UI facilitates the release of our game on various platforms like PlayStation and Xbox. This future-proofs our UI, and ensures a system which doesn't need to be reworked to function on other systems.

Main menu

We wanted to create a seamless experience when the player starts the game. Traditionally, games use the main menu as a splash screen, which you use to load into a saved game. Our approach took inspiration from Just Cause 3 (Lesterlin, 2015), where upon opening the game, immediately loads into the last save. The main menu is then overlaid on top of the game, meaning that pressing 'Continue' simply removes the main menu and focuses the camera back on the player character.

Figure 15: Just Cause 3 main menu. Upon pressing 'Continue', the transparent overlay disappears, and the player character transitions out of the resting pose, creating a seamless transition.

Context menu

Our game required various user interface menus to provide control and information to the player. We aimed to streamline the process of accessing menus, and keep them as simple as possible. During the pre-production phase, we drew inspiration from the menu layout in

Assassin's Creed Origins (Guesdon, 2017), in the way multiple menus are consolidated into a single, tabbed experience.

To achieve this, we created a context menu that integrates the pause menu, ability menu, control scheme and settings menu all into a cohesive interface. Pausing the game reveals the context menu, which offers a tabbed experience with the pause menu as the default tab. Players can switch between tabs using keyboard or gamepad buttons, or by clicking on the tabs directly. Smooth transitions between tabs were implemented, in addition to hover and click animations to maintain the games fluidity.

The ability menu received special attention in terms of graphical design, audio and animations. It was designed to enhance the player's experience when spending hard-earned upgrade schematics, aiming for a satisfying and engaging interaction beyond just implementation.

Custom movement behavior

A sense of control is integral to the overall game feel, with Swink (2009, p. 89) suggesting that the majority of this feeling stems from responsive gameplay. In other words, what buttons lead to what results on-screen. Freedom of movement was one of our main goals, and thus player input in Stranded: Astral Odyssey is mostly related to the movement of character. From playtesting we found that we needed more than the default character- and jetpack movement to make the game fun, and sell a feeling of freedom. The solution was to expand the player's options by adding additional unlockable movement modes.

Jetpack

The jetpack plays an essential role in creating *fun* in the game. As Schell defines in 'The Art of Game Design' as "Lens #17: Lens of the Toy" (Schell, 2015, p. 107), which prompts us to consider whether the game is inherently enjoyable to interact with, regardless of specific objectives. This lens focuses on the presence of mechanics that are engaging and entertaining in their own right, regardless of progression or goals. The design of the jetpack prioritizes simplicity, responsiveness and versatility, allowing players to play with it freely at any point during gameplay.

“When people see my game, do they want to start interacting with it, even before they know what to do?” (Schell, 2015, p. 107). Through playtesting, we observed that players were drawn to the game specifically to engage with the jetpack. Merely observing others play was enough to incite their desire to interact with the game. Schell challenges developers to ask themselves whether the game would remain enjoyable if all goals and objectives ceased to exist, and through the implementation of the jetpack and additional movement capabilities, we believe we have achieved this goal.

Considerable time was spent polishing the feel of the expanded movement. For our directional burst effect, a custom system was set up that handled the movement behavior as well as the camera effects. We used timelines in Unreal Engine to define movement over time, and used easing curves to smooth out transitions to refine the feel of the ability. In order to properly playtest the ability, all the values used in the code were made parametric and easily changeable, such that they could be tweaked both in the editor and at runtime to see the changes live.

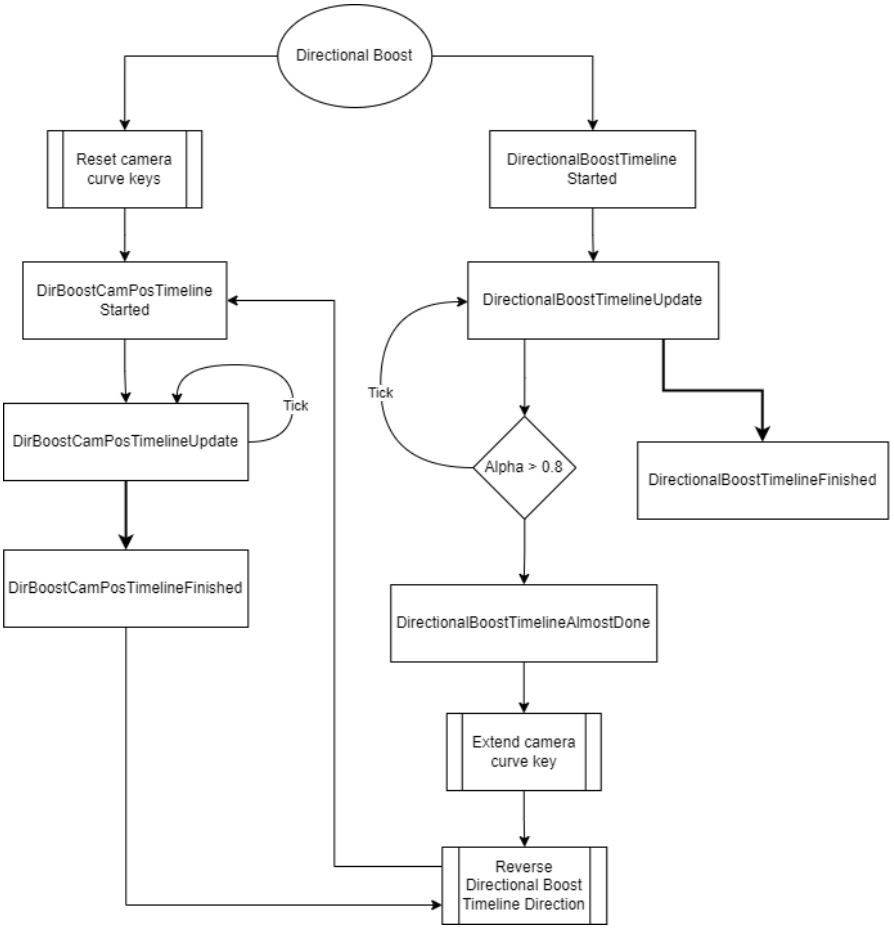

Figure 16: Directional burst execution diagram. Two timelines are triggered at the same time, one for movement and one for camera effects. When the ability is nearing finished, we edit the curve keys to get a smoother transition when reversing the camera effects.

Ground movement

Largely, the in-engine movement that comes with Unreal Engine for a third person game was deemed satisfactory by the development team and not set as a focus area to improve massively upon. While that remains true, with the goal of having an additional mode of movement in the air where the player could fly around with the jetpack, and with the air movement being a focus for a satisfying experience, a need arose to allow more transitions and interaction between the two modes than just jumping from the ground and then boosting.

Like the initial concept for the game where the player could swap between a hovering vehicle and a bipedal character, the goal was for transitioning between ground movement and boosting around to be smooth and fluent. This meant that with the addition of the ability to slide on the ground, it should also be easy and rewarding to transition to using the jetpack.

In summary: The expanded movement options given to the player on the ground, like sliding, boosting out of a slide, crouching and charging a boost jump, unleashing said boost jump, and also the option to hang onto walls in the air, were designed to seamlessly integrate the default third-person movement with the jetpack.

Development

For the technical challenge of implementing expanded movement, a custom movement component class was created and added to the character, to handle state logic when switching to new states like sliding, boosting, hanging, etc. As it consisted of a large portion of the code, the movement codebase was worked on by multiple programmers. This meant extra rigor in naming conventions, commenting, and writing understandable but also easily expandable code. Additionally, to help with playtesting and continuous interaction, all relevant variables were exposed to the editor to allow for quick adjustment to see what felt best during playtesting.

Wind system and foliage reactivity

As stated by Swink, the concept of a simulated space is one of the building blocks of game feel (Swink, 2009, pp. 4–5). Thus we sought to emphasize the authenticity of the game world by adding dynamic movement to foliage, such as grass, trees and bushes. Specifically two different systems for adding dynamicity to the game world, wind and foliage reactivity.

Wind

The first system is wind blowing through the game world, in order to build spatial immersion (Adams, 2014, p. 21). A repeated response to an early progress presentation of the project, was that our game world seemed static and artificial. As such we opted for wind-effects as one of the ways to make the game world appear more like a dynamic space, and less like a static stage. However, as Erik Zimmerman and Katie Salen Tekinbaş writes, “Given a phenomena to simulate, the problem is to decide what are its parts, how these parts can be represented [...].” (Tekinbaş & Zimmerman, 2003, p. 19).

Wind is moving air, which is a fluid (Gjevik & Fagerland, 2017, pp. 195–197), and can thus be realistically simulated with fluid mechanics. However, doing so is computationally expensive (Lawson et al., 2012). We therefore opted for a less expensive emulation of wind, by generating a computational noise field in unreal (Epic Games, 2024c) where each in-game world position in the horizontal plane corresponds to a location in the generated noise field. The noise field is then scaled and scrolled in a given direction to emulate the patterns of wind flowing through foliage.

Firstly, the normalized amplitude of the noise is used as a scale-factor for the deflection of each vertex, of the foliage mesh, in the direction of the wind. Furthermore, the deflection is scaled by the square of the normalized height from the root to the tip of the foliage, making the mesh curve along the shape of a parabola. Finally, the new position of the vertex is rescaled to maintain the original distance away from the root; minimizing stretching of the mesh. During the last progress presentation, as well as following play-testing sessions, we received positive feedback on the wind-system affirming that the game world became more immersive and more dynamic because of it.

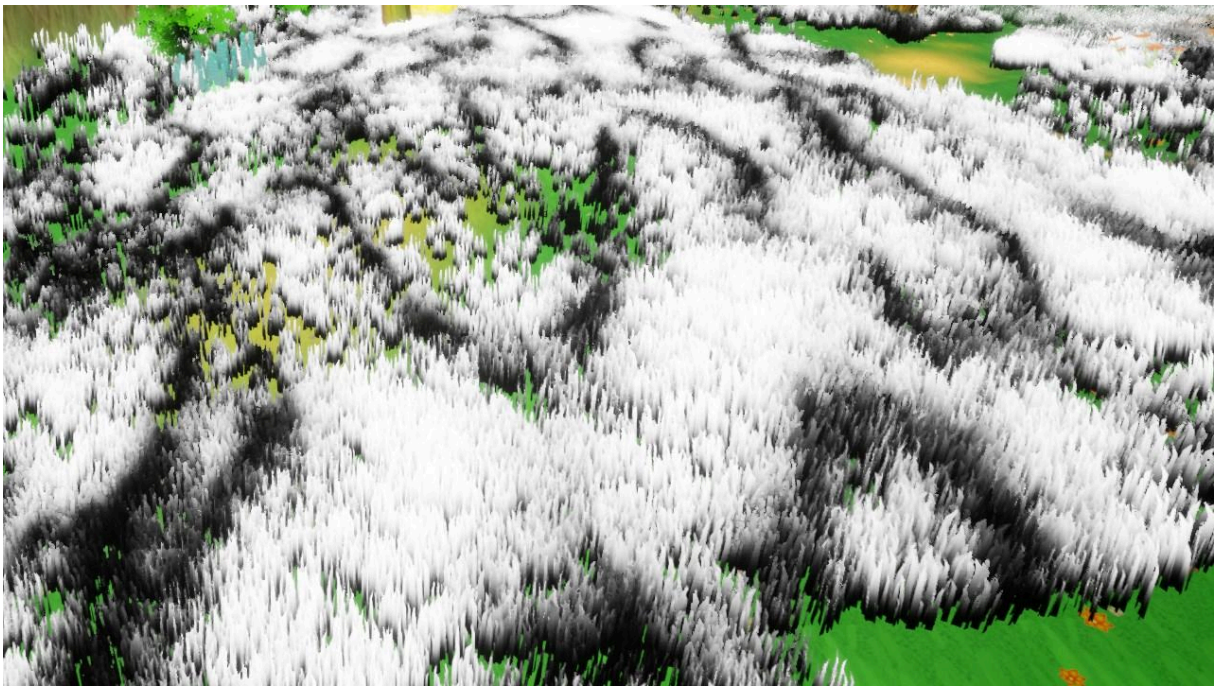

Figure 17: Illustration of noise-pattern emulating wind on grass. Black is zero vertex-deflection, and white is maximum vertex-deflection in the wind direction. The noise is moved in the wind direction with procedural changes to emulate turbulence.

The wind direction and speed can be customized for individual areas using WindController actors with detection volumes, which adds further dynamicity to the game world. The purpose is to build consistency with player expectation of how wind works, with hilltops being more windy and caverns being free of wind. The idea to use computational noise came from a tutorial by Visual Tech Art on YouTube (Visual Tech Art, 2022), and the height masking relies on a helper function created by Matt Oztalay (Oztalay, 2022).

Foliage reactivity

Reaction to player character

The second system implemented to make foliage more dynamic, was foliage reacting to the players presence and actions. Firstly, we created a spherical area around the player, where foliage is deflected away from the player character using the same height scaling and length-preservation described in the section regarding wind. Thus grass bends down and away under the player character, and bushes and tree-crowns rustle and bend when the player character moves through them. This implementation is based on a tutorial by SgtMcTarget on YouTube (SgtMcTarget, 2022).

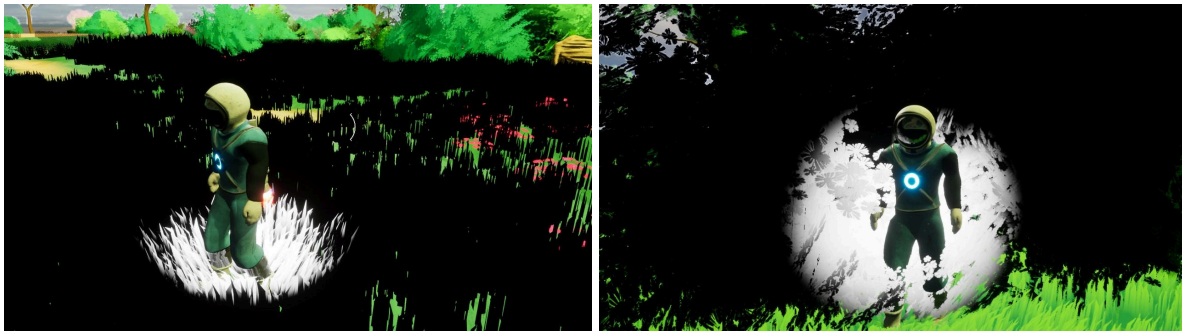

Figure 18: Foliage deflection from player position. White is maximum deflection, and black is zero deflection. Left: Deflection in grass. Right: Deflection in bushes and trees.

Reaction to jetpack

The central gameplay mechanic of ‘Stranded: Astral Odyssey’ is the jetpack allowing for various forms of flight. In order to enhance the game feel of the jetpack mechanics, we decided that foliage should react to its use. Taking inspiration from real-life jetpacks (Gravity Industries, 2020) the goal was to emulate a stream of air from the jetpack hitting the ground and spreading outwards in a turbulent manner that affects foliage.

To achieve a similar effect, we start with the plane wave solution

$$u\left(t,x\right)=A\cos\left(kx-\omega t+\phi\right)$$ (Vistnes, 2018, p. 139),

to a scalar wave equation

$$\frac{\partial^2 u}{\partial t^2}u\left(t,x\right)=\frac{\omega^2}{k^2}\frac{\partial^2 u}{\partial x^2}u\left(t,x\right)$$ (Vistnes, 2018, pp. 137, 140),

which can be made spherically symmetric by substituting the spatial coordinate x with the distance from where the wave is emitted to the vertex position in world space $|\vec{r}_{\text{vertex}}-\vec{r}_{\text{emitter}}|$. We then scale the amplitude A with the inverse square-root of the distance, such that the wave gets less intense the further it is from the jetpack, leaving us with the final expression for the strength of per-vertex deflection as

$$u\left(t,\vec{r}_{\text{vertex}}\right)=\frac{A\cos\left(k|\vec{r}_{\text{vertex}}-\vec{r}_{\text{emitter}}|-\omega t+\phi\right)}{\sqrt{|\vec{r}_{\text{vertex}}-\vec{r}_{\text{emitter}}|}}.$$

The amplitude A, wave-number k, angular speed ω and phase-offset ϕ can be tweaked per-vertex to change the look and behavior of the resulting ring-wave. Furthermore, we avoid division by zero by returning only the amplitude when the vertex position equals the emitter position.

The effect has an outer cut-off radius that starts at the jetpack position when the player begins boosting, and expands outwards with time. Similarly, an inner cut-off radius expands outwards from the jetpack when the player stops boosting.

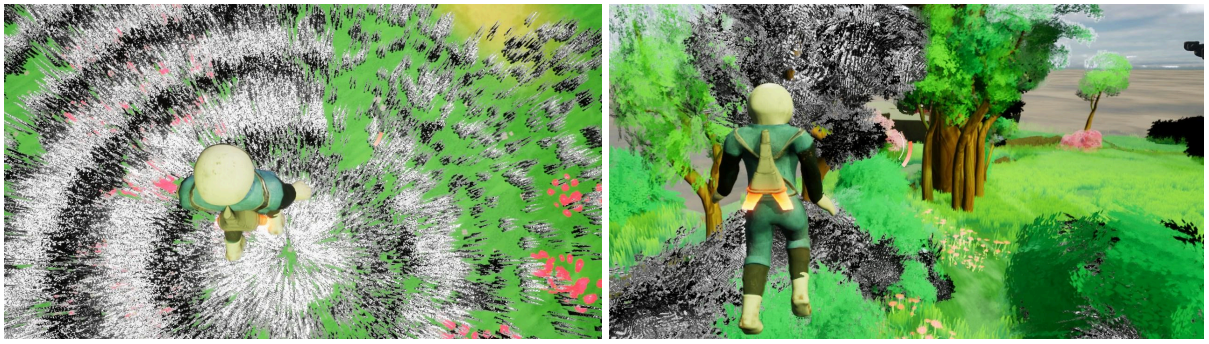

Figure 19: Visualization of ring-wave shaped vertex deflection in foliage. Black is zero deflection and white is maximum deflection. Right shows the effect on grass; left shows the effect on trees and bushes. Noise is added with a randomized phase-offset per vertex.

The purpose of foliage being reactive to player actions, and the player character's presence, is to increase the physicality of the player's interaction with the game. These effects don't serve

any practical function towards progressing in the game, such as the jetpack itself does, and therefore fall under what Swink refers to as ‘polish’ (Swink, 2009, pp. 5–6). Still, they enhance the player’s experience of the main mechanics by visually tying the player’s action with the simulated space the player character exists in.

Bobbing islands

Another form of reactive environment was added, by making some of the smaller floating islands deflect and bob up and down when the player steps on them. This effect was achieved by simulating the island mesh as attached to a spring anchored with an invisible component. The spring force F_{spring} is calculated using Hooke’s law

$$F_{\text{spring}} = -kz(t)$$ (Vistnes, 2018, p. 14),

where k is the spring constant representing the spring stiffness, and $z(t)$ is the island’s displacement from the anchoring point in the vertical direction. When the player character lands on top of one such island, the character’s downward momentum is used to set the initial vertical velocity of the island

$$\dot{z}_{\text{island}} = \frac{\dot{z}_{\text{player}} m_{\text{player}}}{m_{\text{island}}},$$

and the player character’s weight is calculated and added as a persistent force

$$G_{\text{player}} = -m_{\text{player}} g,$$

where g is the gravitational acceleration. The weight term is set to zero when the player no longer stands on the island, allowing the island to return to its original position.

To make sure the islands bobbing motion comes to rest, a damping force is applied

$$F_{\text{damping}} = -\mu \dot{z}_{\text{island}},$$

where the damping constant μ is calculated using the spring constant and a damping ratio c

$$\mu = 2c\sqrt{k \cdot m_{\text{island}}}.$$

Combining the forces, the islands motion can be numerically integrated per frame using the Forward-Euler method (Vistnes, 2018, p. 62) on the following differential equation

$$m_{\text{island}} \ddot{z}_{\text{island}} = -kz_{\text{island}} - \mu \dot{z}_{\text{island}} - m_{\text{player}} g.$$

The bobbing island allows for challenges where the player must jump from platform to platform as the islands drop down when the player jumps on them. Furthermore, switching the sign on the player weight term allows for the island to rise up, and be used as a dynamic elevator. By adjusting the damping ratio, and spring constant, the maximum displacement and speed of displacement can be changed according to need.

Audio implementation

“Atmosphere is invisible and intangible. But somehow it envelops us, permeates us, and makes us part of the world.” (Schell, 2015, p. 387) We wanted to create a game world that goes beyond visual aesthetics, establishing a rich and vivid atmosphere that puts the player *inside* the game. Recognizing the crucial role of atmosphere in game design, we used various tools and techniques to achieve an engaging atmosphere that amplifies the visuals and enhances the game feel.

For environmental sounds, audio clips from various environments were sourced, edited, and placed within the game environment at fitting locations. Methods like attenuation and reverb were used to create a living space that the player can hear in three dimensions. For instance, as you approach a waterfall, you will hear the roaring water growing louder. From different directions, you might also hear the rustling of leaves from nearby trees. Additionally, zones were defined to dampen the sound of the wind. Therefore, when walking into a cave or an area shielded from wind, you will hear the diminishing noise of the wind, similar to real-life.

Schell emphasizes the significance of selecting music for your game early on (Schell, 2015, p. 390). Planning our audio and music implementation early in the pre-production phase, music served as a guiding element to visualizing our environment and most importantly setting the pace for the game. To empower our level designers, we developed a system where music volumes can be placed in the world, each associated with a specific music category. When the player enters one of these volumes, they will hear music based on the category assigned to that volume. For example, tense music might be played in areas where the player can get attacked.

To gather the necessary audio files for our game, we used the following audio libraries: Pixabay (Pixabay GmbH, 2024), Zapsplat (McKinney, 2024) and Freesound (Freesound

Team, 2024). Additionally, the ElevenLabs (ElevenLabs, 2024) website was used to generate the raw AI voice lines used for our AI assistant.

Wwise implementation

Wwise (WaveWorks Interactive Sound Engine) (Audiokinetic, 2023b) is one of the leading audio middleware tools in the gaming industry. Developed by Audiokinetic, it's designed specifically to handle interactive audio in games. It has been used in games like Assassin's Creed (game series), Overwatch, DOOM and several other AAA titles (Audiokinetic, 2024).

The initial decision to use Wwise was based on wanting to gain experience with more sophisticated tools, and getting access to dynamic sound modulation capabilities. As a rather large and established software package, getting started with Wwise packed a steep learning curve. At times we regretted choosing Wwise for handling all our audio, however this opinion changed over time as we discovered its capabilities.

In our game, we leverage the Wwise sound engine to create dynamic audio which responds to the player's actions and modulates the sound at runtime. This enables a new horizon of feedback methods, which is why our opinion changed on using the engine. Wwise lets us define RTPCs (Real-Time Parameter Controls), which we can change at runtime with Unreal Engine. We can then use said parameters to define modulation curves in Wwise to change different aspects of our audio clips, like pitch, stretch, filters and volume.

An example of where we use this technique is on the player jetpack. When the heat builds up to over 60%, Wwise will modulate the sound clip to gradually increase the pitch until we reach 100%. To do this we defined an RTPC, and made a curve that increases the pitch when the jetpack heat ranges from 0.6 to 1.0. We use a similar technique to change up the footstep sound effects: each footstep that plays relies on a player speed RTPC, which stretches the audio clip depending on the player's movement speed. In that way, sprinting footsteps will be shorter in time and sound more realistic. Additionally, pitch randomization is applied such that no one footstep sounds exactly the same.

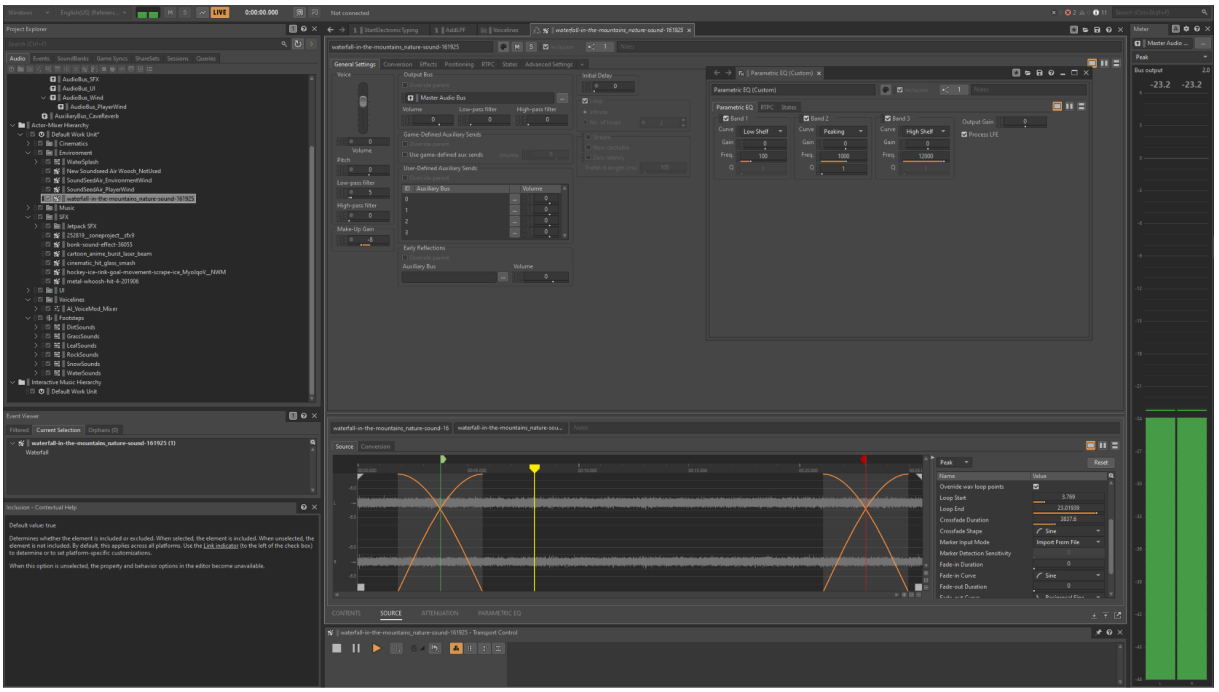

Figure 20: Modulating audio clips in Wwise.

After contacting the sales team at Audiokinetic, we were able to obtain a limited license for the SoundSeed Air plug-in (Audiokinetic, 2023a), which is a plugin that can generate wind sound dynamically. Making use of this plugin allowed us to set up a connection from the visual wind system to an auditory wind system, through use of gameplay parameters. The wind system in-game now updates the aforementioned parameters, which then update curve values in the Wwise engine. Thus, the wind sound changes parameters dynamically such as wind volume, speed, variability and gustiness.

Lastly, we leverage Wwise for our dynamic music system. Wwise has different types of audio containers that can hold several audio clips. These containers will perform different actions when they are triggered by an event via Unreal Engine. One we use a lot is called a random container, which plays a random clip from its children. It also includes functionality to avoid repeating the last couple of clips. For our music system, we use random containers in addition to switches, which is another type of container that chooses an active child based on a state. For our game we have three music states: Normal, Mystery and Tense. Thus, we have one switch container, and three random containers as children. Each random container contains music clips for one music state. Through Unreal Engine, we can then change the music state at runtime and hear the music change.

Audacity

Audacity is a free and open-source audio editing software (The Audacity Team, 2024), and served as a complementary tool alongside Wwise in our audio production workflow. Prior to importing audio files into Wwise, we utilized Audacity for preliminary processing tasks, particularly with the AI assistant voice lines requiring extensive audio adjustments and filtering to achieve a robotic sound. While Wwise offers virtual trimming capabilities, meaning selecting portions of the audio file to use, we leveraged Audacity to precisely trim unnecessary sections of audio clips to optimize space usage for our game.

Cutscenes and cinematics

From the initial concept document, storytelling cutscenes were planned, recognizing their importance as a precursor for our story (Henriksen, 2023). We believe that the narrative should be shown to the player, not told, therefore we invested time in visual storytelling to introduce it. Our goal was to immerse the players in a compelling narrative, showcasing that there is depth to the game beyond the gameplay alone.

When implemented, cutscenes were incorporated strategically within our game to provide players with a sense of understanding over the narrative. It was important for us to make the player feel at one with the player character and understand the situation they’re in.

The introductory cutscene sets the stage by informing the player of the main conflict, and leading with visual storytelling depicting how the mission fell into chaos. This cutscene concludes with a glimpse into upcoming gameplay and objectives, hinting at future objectives as the camera pans through various locations containing scattered wreckage. In this way, we leverage cutscenes to give the player a better understanding of their main objective.

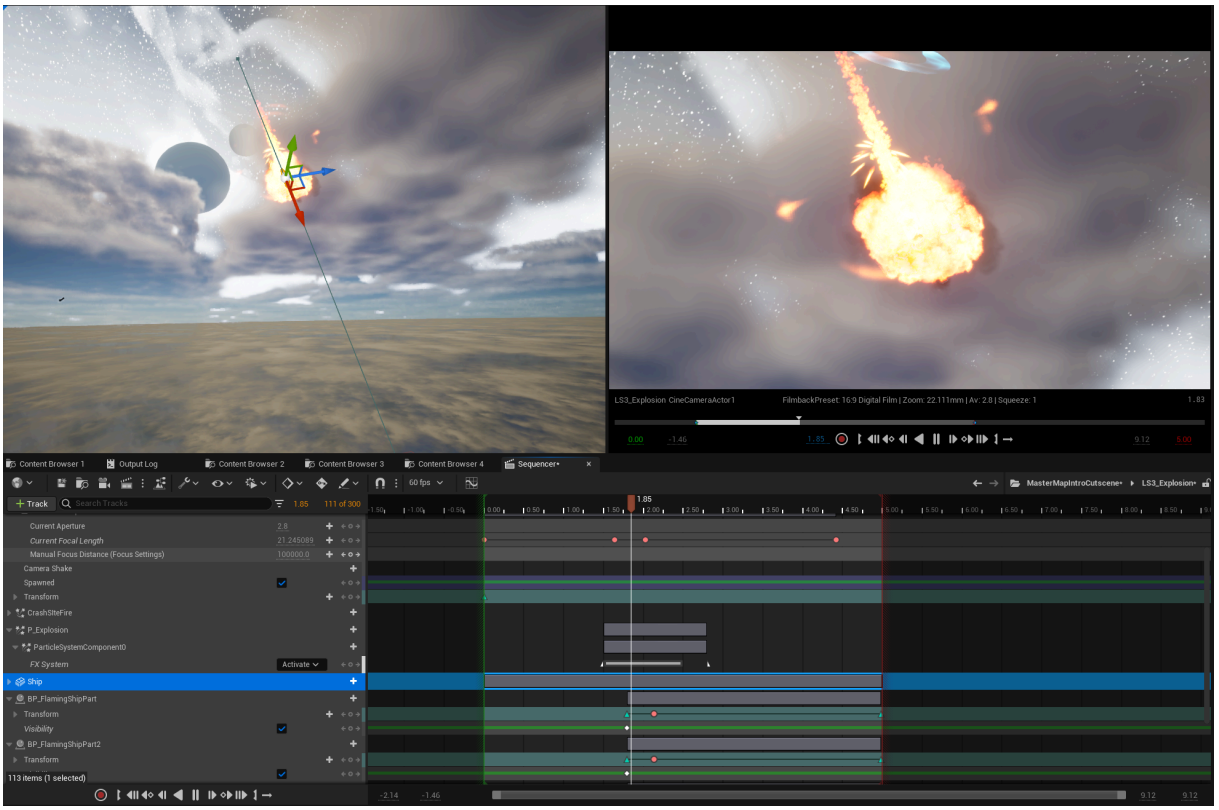

Figure 21: Cutscene creation using the Unreal Engine sequencer

Performance issues and optimization

When we initially developed the concept, we wished to utilize Unreal Engine's new features, such as Nanite (Epic Games, 2022; Henriksen, Åsbø, et al., 2024, p. 12) which handles automatic polygon reduction in high poly meshes, and Lumen (Epic Games, 2022; Henriksen, Åsbø, et al., 2024, p. 12), which handles real-time ray-tracing for reflections, shadows and lighting.
These were both systems that offered simplified work-flows, with high-quality visual results.

However, when finalized assets were added to the game world during the latter half of development, it became clear that we faced major performance issues on weaker hardware, such as laptops, and older graphics cards. On the main island, we experienced frame rates as low as 25 frames per second, with the average being 33 fps. Furthermore, there was noticeable stuttering and freezing on our test laptop.

CPU: AMD Ryzen™ 7 6800H Mobile Processor
- **8-core/16-thread**
- **20 MB cache**
- **up to 4.7 GHz max boost**

GPU: NVIDIA® GeForce RTX™ 3060 Laptop GPU
- **1752 MHz* at 140W**
- **1702 MHz Boost Clock+50 MHz OC**
- **115 W+25 W Dynamic Boost**
- **6 GB GDDR6**

RAM: 16GB DDR5-4800 SO-DIMM

Table 1: Technical specifications of test laptop.

While originally suspecting the number of triangles drawn at any one time, or the number of draw calls to the GPU, we found that these numbers were within reason. With a maximum of three million triangles drawn on screen at any one time, and a maximum of two thousand draw calls per frame, these were ruled out as the cause of our performance issues.

The performance issues were narrowed down to the excess texture memory used by Nanite and Lumen on the GPU, as well as Nanite's additional reliance on the CPU for mesh processing, causing increased reliance on memory transfer between CPU and GPU. Disabling Nanite and Lumen and instead relying on the older hierarchical level-of-detail and screen space lighting systems, resulted in an approximate 66% reduction in video memory usage. Thus, we achieved a frame-rate increase to the 50-70 fps range on our test laptop.

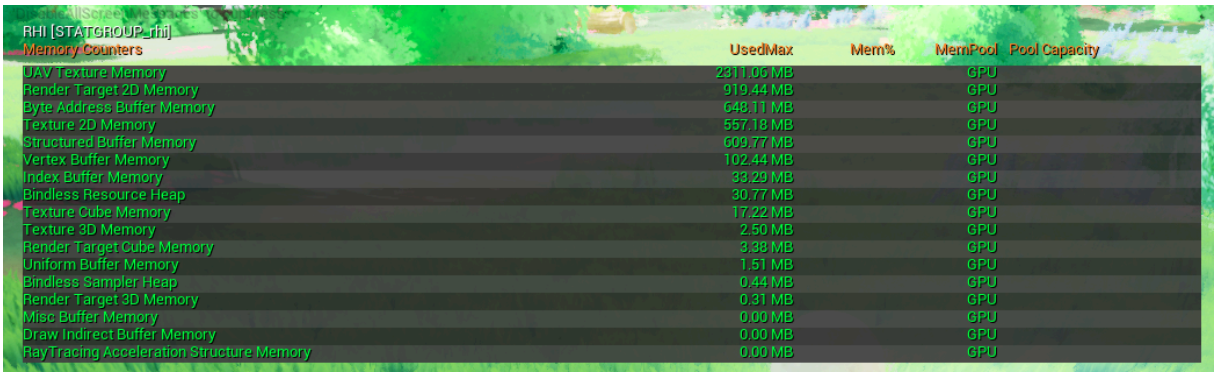

Figure 22: Video memory usage with Nanite and Lumen enabled. Totaling ca. 5.3 GB of memory in use.

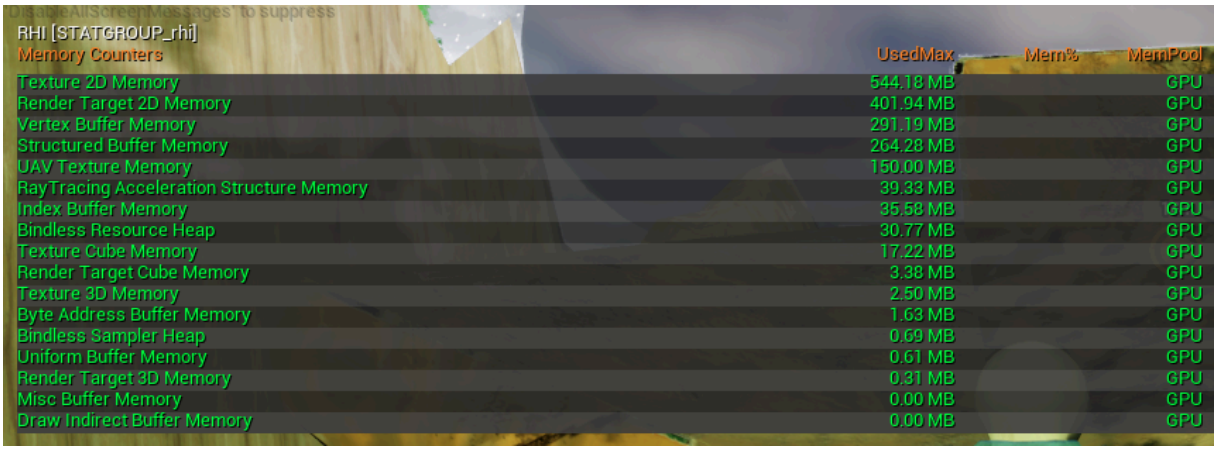

Figure 23: Video memory usage with Nanite and Lumen disabled. Total memory usage ca. 1.8 GB.

Further optimizations include fading out foliage with distance to reduce rendered triangles, and setting a cutoff distance on the foliage wind, and interaction effects, such that foliage sufficiently distant from the camera no longer simulates mesh deflection.

Results and Discussion

Playtesting and feedback

To evaluate our prototype, we got several people to try playing our game. The main playtest was carried out on the 23rd of April 2024. However, additional playtesting was carried out, both internally and externally, during development. From these sessions, valuable feedback was received.

According to Isbister and Schaffer in their book ‘Game usability: Advice from the experts for advancing the player experience’, the focus in playtests is on whether the game is fun to play, but also where players may be getting stuck or frustrated (Isbister & Schaffer, 2008). The feedback players had about the game fell almost exclusively into the latter category. This painted an overall picture of a situation where the main mechanics and assets had clearly come together to form an enjoyable experience, but sometimes the medium through which it was experienced (a linear progression of quests where the player is traveling from A to B to C) was missing some direction. Some of the players reported feeling lost, or unsure where to go next at different points throughout the game. Hoping to reconcile this, we made changes to

the UI and gameplay to more clearly convey the next quest, and the artists worked on designing the islands in such a way that the player would naturally understand where they were supposed to go.

Regarding our goal to create a satisfying and engaging movement system for the players to use seems to have been met, as the majority of playtesters reported being satisfied with the player character movement. During the playtesting session on the 23rd, most playtesters spent a minimum of 15 minutes playing the game, and two people spent over 40 minutes. Both of the longest playing testers reported having a positive experience, and being overall satisfied with the movement. Still, we received some minor critical feedback. For example: one tester commented on transitions between certain movement modes not being smooth enough, which was promptly addressed, and also easy to solve for.

Development process

Towards the end of the development cycle, a consensus emerged between all members of our project - we had underestimated the task of level design. Something that should have been its own dedicated task that was worked on from the beginning of development had fallen to the wayside, and been mostly done on the fly. Keeping in mind the fact that the guidelines for the goal of this project was to create a ‘playable tech demo’ and not necessarily a full-fledged game, it still felt like something integral to the overall experience had been lost by not focusing enough on the bigger picture of how the game would look. Especially for a game like Stranded: Astral Odyssey which is played on one big level, level design is more than just deciding where to put things. Level design becomes a process in which you have to consider all the mechanics of the game, where what you are building is the game experience itself.

This is reflected in the feedback from the players as well. We are confident we created the mechanics for a satisfying and well-rounded game experience with Stranded: Astral Odyssey, but if there is one direction we could continue working on to make the game come together in a holistic way it would be level design.

In the project description document, we performed some rudimentary risk analysis of our team and project (Henriksen, Åsbø, et al., 2024, p. 10). Furthermore, we set up a planned

timeline for our project. At the end of our project, the actual development timeline had deviated somewhat, as overhead from our other courses caused our finalization stage to be pushed from early april to late april and early may.

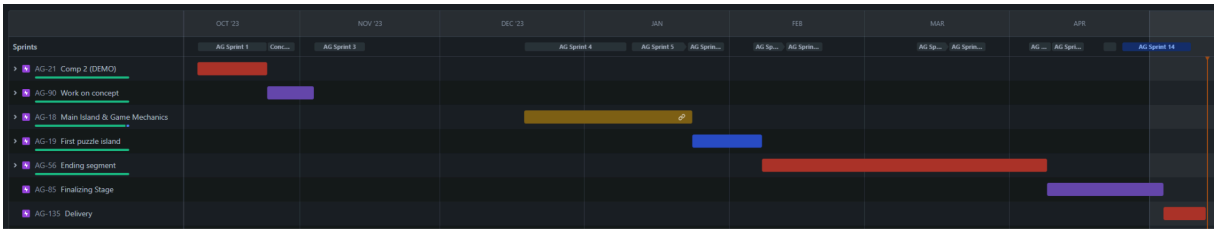

Figure 24: Overview of final development timeline. The titles of each segment does not necessarily reflect the actual focus of said segment, as assignments were reshuffled according to need.

Outside of the shifting dates, the overall timeline matched what we expected at the start of the project. For further info about our pre-production and project planning, refer to the project description document (Henriksen, Åsbø, et al., 2024, pp. 7–11).

Future improvements

Regarding future improvements and feature expansion, one of the prime mechanics to change would be the linear nature of the quest design and tracking. To better facilitate and encourage exploration, a branching quest tree, with multiple simultaneous objectives would be desirable. Such a system would also allow greater player choice in how to explore and experience the game world. During development, we decided to maintain a linear structure to avoid overengineering the quest-system for the prototype, and to avoid excess development time being spent on a system that may not be fully utilized.

Conclusion

The goal of this project was to create a prototype for our game idea outlined in the project description (Henriksen, Åsbø, et al., 2024, p. 3) with a focus on creating a satisfying game feel in accordance with the definition put forth by Swink (Swink, 2009, pp. 2–6). In order to achieve this, we built a technical prototype of a platform based exploration game with jetpack mechanics.

Our project was a collaboration between our group of three programmers, and another group of three artists. To manage such a large team and effectively distribute the workload, we adopted an agile scrum work methodology. We used regular meetings and workshops to coordinate efforts. Though communication and motivational challenges were met during development, we managed to learn from our experience and solve most of our problems.

We created a quest system to offer directed progression. Implemented an upgrade system for the movement mechanics and jetpack, in order to control the speed of progression and add dynamicity to the gameplay experience. Furthermore, we spent a considerable effort on input and user interface, to facilitate a smooth and painless user experience.
Additional systems such as wind, foliage reactivity, and advanced audio were utilized to build a sense of authenticity and dynamicity in the simulated space of the game world. While some systems and components of the systems were cosmetic in nature and served as polish, some aspects, like the dialogue of the AI assistant, served the function of providing vital information to the player.

As discussed in the section on Playtesting and feedback, the overall reception of our prototype was satisfaction with the game mechanics. We therefore conclude that we achieved our goal of creating a satisfying set of mechanics constituting our system of movement. Furthermore, as argued throughout the report, we have designed our systems and mechanics in accordance with the principles put forth by Steve Swink in ‘Game Feel: A game designer’s guide to virtual sensation’ (Swink, 2009). Taken in consideration with the positive feedback we received regarding our movement system, we conclude that we achieved our goal of creating a satisfying game feel.

References

Adams, E. (2014). *Fundamentals of game design* (Third edition). New Riders.

Audiokinetic. (2023a). *SoundSeed Air* (2023.1.0) [C++]. Audiokinetic Inc. https://www.audiokinetic.com/en/products/soundseed/

Audiokinetic. (2023b). *Wwise* (2023.1.0) [C++]. Audiokinetic Inc. https://www.audiokinetic.com/en/

Audiokinetic. (2024). *Audiokinetic*. https://www.audiokinetic.com/en/wwise/powered-by-wwise/

Discord Inc. (2024). *Discord* [TypeScript, Elixir, Python, Rust, C++; Windows, macOS, Android, iOS, iPadOS, Linux, web browsers]. Discord Inc.

ElevenLabs. (2024). *AI Voice Generator & Text to Speech* [Artificial intelligence]. ElevenLabs. https://elevenlabs.io

Epic Games. (2022). *Unreal Engine API Reference* [Documentation]. Unreal Engine. https://docs.unrealengine.com/4.27/en-US/API/

Epic Games. (2023). *Unreal Engine* (5.3.2) [C++, C#, Python; Windows, Linux, macOS]. Epic Games.

Epic Games. (2024a). *Common UI Plugin For Advanced User Interfaces In Unreal Engine | Unreal Engine 5.4 Documentation*. Epic Developer Community. https://dev.epicgames.com/documentation/en-us/unreal-engine/common-ui-plugin-for-advanced-user-interfaces-in-unreal-engine

Epic Games. (2024b). *Data Driven Gameplay Elements* [Documentation]. Unreal Engine Documentation. https://docs.unrealengine.com/5.3/en-US/data-driven-gameplay-elements-in-unreal-engine/

Epic Games. (2024c). *Utility Material Expressions in Unreal Engine* [Documentation]. Unreal

Engine Documentation. https://dev.epicgames.com/documentation/en-us/unreal-engine/utility-material-expressions-in-unreal-engine

Freesound Team. (2024). *Freesound* [Audio clip sharing]. Freesound. https://freesound.org/

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Gjevik, B., & Fagerland, M. W. (2017). *Feltteori og vektoranalyse: Forelesninger og oppgaver* (1st ed.). Farleia Forlag.

Gravity Industries (Director). (2020, September 29). *Paramedic Mountain Rescue!* https://www.youtube.com/watch?v=gtvCnZqZnxc

Guesdon, J. (2017). *Assassin's Creed Origins* [PlayStation 4, Windows, Xbox One]. Ubisoft Entertainment. https://www.ubisoft.com/en-us/game/assassins-creed/origins

Henriksen, M. N. (2023). *Astroglide: Concept Document* [Unpublished paper]. Innland Norway University of Applied Sciences.

Henriksen, M. N. (2024). *Unreal Utility* (0.0.1) [Go; Windows]. https://github.com/henriksen-marcus/unreal-utility

Henriksen, M. N., Åsbø, A. P., Brændeland, S., Finseth, J.-R. T., Carlsen, V., & Wallace, A. (2024). *Project Description of 'Stranded: Astral Odyssey'* [Unpublished paper]. Innland Norway University of Applied Sciences. https://docs.google.com/document/d/1KZHLQp2lgRL1qWjytsQrAdS9haztcyuPzusNE76NyXs/edit?usp=sharing

Henriksen, M. N., Brændeland, S., Åsbø, A. P., Finseth, J.-R. T., Carlsen, V., & Wallace, A. (2024). *Stranded: Astral Odyssey—GDD* [Unpublished paper]. Innland Norway University of Applied Sciences. https://docs.google.com/document/d/12iQXY-BNzJJt8eTVIdSJ7wZKkZcWXgspTGz

gHht9rzU/edit?usp=sharing&usp=embed_facebook

Hiwiller, Z. (2016). *Players making decisions: Game design essentials and the art of understanding your players*. New Riders.

Isbister, K., & Schaffer, N. (2008). *Game usability: Advice from the experts for advancing the player experience*. Morgan Kaufmann ; Elsevier Science [distributor].

Kristiadi, D. P., Sudarto, F., Sugiarto, D., Sambera, R., Warnars, H. L. H. S., & Hashimoto, K. (2019). Game Development with Scrum methodology. *2019 International Congress on Applied Information Technology (AIT)*, 1–6. https://doi.org/10.1109/AIT49014.2019.9144963

Lawson, S. J., Woodgate, M., Steijl, R., & Barakos, G. N. (2012). High performance computing for challenging problems in computational fluid dynamics. *Applied Computational Aerodynamics and High Performance Computing in the UK*, *52*, 19–29. https://doi.org/10.1016/j.paerosci.2012.03.004

Lesterlin, R. (2015). *Just Cause 3* [C++; PlayStation 4, Xbox One, Windows]. Avalanche Studios. https://www.square-enix-games.com/games/just-cause-3

McKinney, A. (2024). *Download FREE Sound Effects at ZapSplat* [Stock sound effects]. ZapSplat - Download Free Sound Effects. https://www.zapsplat.com/

Oztalay, M. (2022, July 14). *Instance-Local Bounding Box UVWs | Unreal engine Code Snippet* [Forum]. Epic Developer Community. https://dev.epicgames.com/community/snippets/X0w/unreal-engine-instance-local-bounding-box-uvws

Pixabay GmbH. (2024). *Pixabay* [Stock media]. 90,000+ Free Sound Effects for Download - Pixabay. https://pixabay.com/

Project Management Institute (Ed.). (2021). *The standard for project management and a guide to the project management body of knowledge (PMBOK guide)* (Seventh

edition). Project Management Institute, Inc.

Schell, J. (2015). *The art of game design: A book of lenses* (Second edition). CRC Press/Taylor & Francis Group.

SgtMcTarget (Director). (2022, July 3). *How to Create Interactive Grass in Unreal Engine 5—Step by Step Guide—Stylized Grass (2023)*. https://www.youtube.com/watch?v=84BeP1sqNY0

Shelton, J., & Kumar, G. P. (2010). Comparison between auditory and visual simple reaction times. *Neuroscience and Medicine*, *1*(01), 30–32.

SimilarTech. (2024). *Jira Market Share and Web Usage Statistics* [Statistical Database]. SimilarTech. https://www.similartech.com/technologies/jira

Swink, S. (2009). *Game feel: A game designer's guide to virtual sensation*. Morgan Kaufmann Publishers/Elsevier.

Tekinbaş, K. S., & Zimmerman, E. (2003). *Rules of play: Game design fundamentals*. MIT Press.

Tew, A. (2023). *Hogwarts Legacy* [C++; Playstation 5, XBox Series X & S, Windows]. Avalanche Software. https://www.hogwartslegacy.com/en-us

The Audacity Team. (2024). *Audacity* (3.5.1) [C, C++; Windows, macOS, Linux, Other]. Muse Group. https://www.audacityteam.org/

Vistnes, A. I. (2018). *Physics of oscillations and waves: With use of Matlab and Python* (R. Naqvi, Trans.). Springer. https://doi.org/10.1007/978-3-319-72314-3

Visual Tech Art (Director). (2022, July 5). *HYPER-REALISTIC Grass Wind Animation [UE4, valid for UE5]*. https://www.youtube.com/watch?v=gfQNjC69PCM