Mars Lander

CS39440 Major Project Report

Author: Jack Philip Partington (jap93@aber.ac.uk)
Supervisor: Dr Laurence Tyler (lgt@aber.ac.uk)

5th May 2023
Version: 1.0 (Release)

This report was submitted as partial fulfilment of a BSc degree in Computer Science (G400)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, U.K.

Declaration of originality

I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Registry (AR) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work I understand and agree to abide by the University's regulations governing these issues.

Name Jack Philip Partington

Date 4/05/2023

Consent to share this work

By including my name below, I hereby agree to this project's report and technical work being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name Jack Philip Partington

Date 04/05/2023

Acknowledgements

I am grateful to my family and friends for their support through this process.

I'd like to extend a thank you to my play testers; Jack, Mulberry, and Kyameron

I'd like to thank my fiancée for sanity checking my report and providing emotional support thoughout.

Abstract

In today's world, where technology plays an increasingly important role in our lives, there is a growing need for educational games that provide engaging ways to teach STEM subjects. Mars Lander is one such game, designed to teach players about physics and space exploration while providing an entertaining gaming experience. In this report, discussion will focus on the process of developing a game of this nature, with respect to the considerations that had to be made to provide educational value to the player.

Extensive research was conducted prior to the development process to comprehensively understand the various factors that would need to be taken into consideration. The research primarily focused on software and games, with a specific emphasis on games that utilize interesting gameplay mechanics to teach the concept of space exploration. The second section of this report outlines the findings of the research.

The game Mars Lander was created using the Free Open Source Software known as Godot. The development process followed an Agile methodology, incorporating principles and concepts from various processes such as Lean and Kanban. The project spanned a period of twelve weeks, during which the game was designed to enable the player to swap out parts on the lander and then descend onto the Martian surface, with the objective of successfully landing the lander.

Contents

List of Figures

Contents

Chapter 1

Introduction

1.1 Introduction

'Mars Lander' is a game that aims to teach players about the decisions that are made through the course of a mission to Mars. This game intends to demonstrate the consequences of these decisions. To do this, an approximation of the physics on Mars will need to be created. An approximation of these physics can only be carried out through research into the real-life physics on Mars. It is hoped that through research and understanding of the concept of "gamification" among other similar technologies currently on the market, an entertaining yet educational game can be created.

Chapter 2

Background, Analysis, & Software Process

2.1 Background

2.1.1 Study of Development Tools

To begin this project, establishing background knowledge is crucial in identifying what the project will entail. From the onset, the decision was made to use a game engine. A game engine briefly summarised is a piece of software that provides a toolkit of predefined objects. These all have predefined functionality that a developer can then extend upon. This means that developers from all across a wide spectrum of skill can utilise game engines to make something impressive.

Initially, Unity looked to be a promising engine to build the project in with it being often cited as one of the easier pieces of game development technology to pick up and use [18]. Despite this, research into which engine to use continued, with engines such as Unreal Engine and Godot being considered.

2.1.1.1 Unreal Engine

It was decided to constrain the scope of the project to a 2D game. Constraining the scope of the project would allow for more preparation and testing to be carried out. This in turn would lead to a better understanding of both the engine and what could be accomplished over the course of the project.

Unreal was ruled out early in the research process as it would be excessive for the aims of the project. This is further backed up by a blog post by Ben Tristem for the website Udemy. In this blog post Ben talks about how ”Unreal can be used for 2D game development, but Epic Games aren’t prioritizing this feature set as much as Unity.” [43]. It should be noted that Epic Games are the company that develop and maintain the Unreal Engine [7].

2.1.1.2 Unity & Godot

With Unreal Engine no longer an option, the focus shifted to Unity and Godot. Unity and Godot both share similar features when it comes to 2D game development. Either of these engines would suit the project's needs, however, Godot is free open source software or FOSS.

Since Godot is FOSS, if the project requires the modification of the engine or the creation of a plugin this can be explored as a possible option. Godot utilises a specially built language called GDScript. GDScript is syntactically similar to python, in that it is formatted using indentation. The author is most suited to programming in Python, and has knowledge of languages such as C#.

2.1.2 Study of Existing Technologies

To gain a better understanding of what would be necessary for the project it was decided to study other software/games that were based around the simulation of physics and Mars. To avoid developing a bias for one approach compared to another, both commercial and educational games/software were considered.

2.1.2.1 Lunar Lander

Whilst not entirely relevant to Mars, Lunar Lander was researched. This was because the project brief mentions this game as an inspiration for the project. Lunar Lander is a ”game that simulates landing a manned spaceship on the moon” [20]. It is important to recognise that this game was not the first of its kind. It was ”mostly inspired by” an earlier game called Moonlander [44].

Lunar Lander was created to be played on an arcade machine. Therefore, a true to life simulation of the physics involved had to be abandoned in order to ensure the game was simple enough to enjoy in arcades [22]. Howard Delman, one of the developers of Lunar Lander, recounts of these limitations in his interview with Technologizer, ”We did all these things to make it easier to play.” [22].

It is important to consider the technological and commercial context that exists with Lunar Lander and not use these to justify restricting the scope on this project. For example, realism in this project should be simulated to a higher degree than Lunar Lander since the game was intended to: make money, capture the attention of customers, be easy to play for all ages, and to keep customers engaged. All of these points, except the last are irrelevant for this project, which is why Lunar Lander had lower importance in the analysis phase of the project.

2.1.2.2 Mars Horizon

Another game to consider is Mars Horizon. Releasing in 2020 from a company called Auroch Digital, this is their first foray into the space simulation genre. Auroch Digital consulted with the European Space Agency (ESA) and the UK Space Agency when developing Mars Horizon, even staying at ESA facilities such as the European Space Operations Centre (ESOC) in Germany and European Space Research and Technology Centre (ESTEC) in The Netherlands [16].

In Mars Horizon, the player starts in 1957. The player has to manage several different facilities as well as the budget in order to achieve milestones that mirror the real life space race. These milestones include but are not limited to; first satellite in orbit, the first person in space, and landing a manned mission on the moon. As the player advances through the various milestones, they will reach a point where they are completing missions that have not been currently completed such as a sample return from Mars and the first manned mission to Mars.

Mars Horizon does not focus on the in mission control of the vessels like in Lunar

Lander, rather Mars Horizon focuses on the decisions and dilemmas that can arise during a mission. An example of one of these dilemmas can be seen in Figure 3.1. This game has a lot of depth to it and can teach a player a more or less complete history of space exploration, that is, if the player remains engaged and completes the game.

Figure 2.1: A screenshot of one of the dilemmas presented to the player [25]

2.1.2.3 Kerbal Space Program

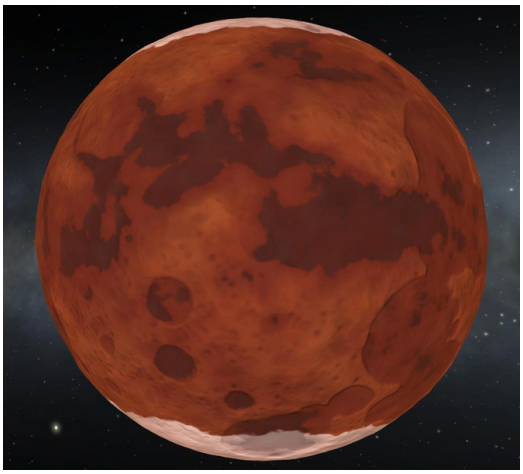
A unique take on the genre of space simulation is Kerbal Space Program (KSP). KSP sees the player take control of the space program of a species of people called kerbals who inhabit a planet called Kerbal. The player is able to build their own custom space crafts, including everything from Space Shuttle-esque spaceplanes to fully autonomous rovers. This also means that the player has to build the launch vehicles that are capable of getting the payload, whether that be a planetary landing vehicle or communications satellite into space, and finally to the desired destination/orbit.

KSP puts more emphasis on trial and error rather than strict calculation and adjustment of parameters. However tools and readouts are available to a player who wants to. For example, fine tune the Thrust-To-Weight ratio of a specific stage of the vessel. Once a player has built a vessel possibly consisting of an engine, fuel tanks, aerodynamic control surfaces, and a control module, the player can then deploy their vessel to the launchpad. From the launchpad, the player is able to take off and guide their vessel into orbit.

Private Division, the development team behind KSP, has worked with ESA to provide self-contained mission spaces that allow players to jump in and take control of past ESA missions [17]. Another aspect of this collaboration is the ability to construct Ariene 5 launch vehicles and use them in player missions [17]. The major downside with KSP is that the games' solar system is fictional. However, loosely based on our solar system. Therefore, all of the physical details of the celestial bodies in the game are wildly different from their real life counterparts.

In KSP, the martian equivalent is called Duna. Duna has been designed to offer the same challenges to players that Mars does to space agencies in the real world. Duna has a significantly thinner atmosphere than Kerbal, lower gravity and no breathable atmosphere. From the authors experiences with the game, drogue chutes, main chutes, and powered descent are the bare minimum required to successfully land on Duna. This is why KSP should not be discounted as a valid inspiration to draw from when considering the gameplay element of the project.

Figure 2.2: A screenshot of the planet Duna as it appears in Kerbal Space Program [13]



2.1.2.4 NASA Perseverance Pre-Landing Simulation

As part of publications released prior to the landing of the Perseverance rover in 2021, NASA made a pre-landing simulation available to the public [3]. This simulation allows a user to watch as Perseverance separates from its cruise stage, descends through the atmosphere, deploys its parachute, identifies a safe landing spot, and then finally touches down safely on the martian surface. The simulation provides various readouts of the crafts current velocity, altitude, and distance from landing zone. The only interaction the user has with the simulation is the ability to orbit the camera around Perseverance and fast forward or rewind the simulation to a specific stage. The Pre-Landing Simulator provides a graphical representation of the Perseverance mission and distinctly defines the different stages involved in a successful mission.

2.1.3 Study of Gamification

Gamification refers to the ”application of game mechanisms in non-gaming environments” [21]. An analysis of several pieces of literature concluded that gamification ”...techniques are being adopted to support learning...” [21]. Generally, this analysis covers how gamification in education is allowing students to become more engaged with their learning and how it is especially useful when working with ”very specific applied courses like graphic arts and gardening” [21].

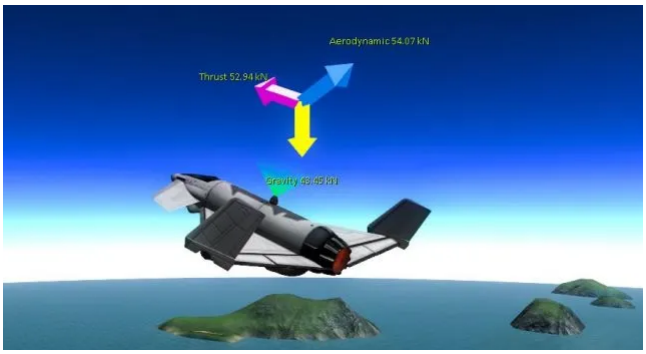
2.1.3.1 EDU Games

Often when a game is popular and has potential educational value, the developers of these games will create whats called an EDU version. This version is intended for education, and generally is limited in content when compared to the commercial version. However, they will often include functionality for teachers to develop their own lessons within the game.

EDU versions are generally free for educational organisations or heavily discounted. It is interesting to note that ”SimCity EDU...KerbalEDU...MinecraftEDU” [14] were the games that gamedeveloper.com reached out to in their article discussing the increase in popularity of EDU games.

As can be seen, both in the quote and Figure 2.2, the second game mentioned is KerbalEDU. Looking at an article written in 2014, it can be seen that KerbalEDU was less about being the sole way of teaching physical sciences in the classroom and more about getting students engaged [41]. Santeri Koivisto, CEO of TeacherGames, the company responsible for KerbalEDU said in an interview with pcgamesn.com ”A lot of schools have one hour per week of computer lab time... So what can you do other than get them excited... so that they learn at home?” [41] This quote from Koivisto perfectly demonstrates one of the best purposes of EDU games, engagement.

Figure 2.3: A screenshot of KerbalEDU [14]



2.1.4 Study of Entry, Descent, Landing on Mars

To understand what the project will be attempting to simulate, it is important to first study the environment that shapes real life missions to Mars.

Firstly, the atmosphere of Mars poses a significant challenge to vessels wanting to land on its surface. The Martian atmosphere is incredibly thin, with it being theorised that billions of years ago, this atmosphere was thick enough to enable liquid water to flow on the surface. NASA's MAVEN mission was able to clarify that most of the Martian atmosphere has been lost to a process called "sputtering" [15]. Sputtering is when solar winds pick up ions, and then these ions collide with atoms in the top layers of the atmosphere. This collision leads to these atoms flying off into space.

This thin atmosphere means that parachutes do not yield the same benefits on Mars as on Earth. On Earth, it is possible to slow and subsequently land a payload with a parachute alone. The same can not be said for Mars. On Mars, parachutes are used to slow a payload as much as possible, but in order for landing to be successful a secondary landing method is required. These methods include powered descent, bouncing gasbags, deadbeat gasbags, and the Skycrane maneuver. Different missions require different methods with the mass of the vessel playing a huge part in deciding which landing method should be used.

The Martian atmosphere may be thin, however, it still poses a threat to vessels upon entry. Atmospheric heating on vessels occurs when vessels are travelling through an atmosphere at high enough speeds to cause the compression of gas in front of the vessel. This compression of gas combined with the friction of the atoms on the vessel results in high temperatures. If a vessel does not have some form of heat-shield that can sufficiently disperse heat, the vessel will be destroyed.

2.1.5 Motivation and Interest

The authors motivation and interest in this specific project comes from a lifelong interest in two subjects, gaming and space. It is hoped that this motivation and interest fuels the learning that is required in order to provide a product that imparts knowledge and enjoyment to the player.

2.2 Analysis

Before deciding on what the outcome of project should look like, the tools used to develop the project needed to be established. With the research carried out in the previous section, its clear to see that both with the authors experience in Python and the FOSS nature of Godot, Godot is a good choice for the development of this project.

One of the most significant challenges in developing a project of this nature pertains to determining the level of accuracy achieved through the physical simulation in the game. The game is intended to be used in an outreach capacity, therefore long term engagement with the game is not necessary. Players will not be sat at a computer playing this game for hours. Due to this, accuracy can be sacrificed if it is found that it is hindering the games ability to impart knowledge upon a player. It is hoped that with this context, a user can play the game and generally understand the principles and decisions that enable the success of real-life missions.

This project will rely heavily on prototyping due to inexperience in game development and a lacking familiarity with replicating physics in code. Therefore, it is reasonable to expect many iterations of different functionality before an solution is reached.

As previously mentioned in the background section, due to the limiting factors mentioned above constraining the scope of the game to 2D will be for the best. As stated in an article from Starloop, a game development company, ”Due to the lower level of complexity, game developers invest less time and money to create 2D games” [2].

It is important to have something at the end that is feature complete. A 3D game may look better and provide a better experience for a player, however, would take longer to develop and therefore complexity would need to be dropped from the final product. The advantage with 2D is that there is time to learn and understand the different systems that exist in Godot whilst also maximising the chance to produce a feature complete, finished game.

With that in mind, the type of 2D game needs to also be considered. A Lunar Lander style game is definitely the first thing that comes to mind, however, there are other approaches. The project brief states the game should show the decisions and choices that go into mars missions. One way to do this is by following the example set out by Mars Horizon and make the game an almost ”pick your own adventure” type game. These games follow the formula of a character who has choices in front of them, e.g ”go to the castle” or ”continue on the road”.

Applying this formula into the context of the project, decisions such as the ones above become decisions like ”delay the launch till the next launch window” or ”launch anyway”. The benefit of this method is the replayability the player gets, because they may want to investigate another branch of the ”adventure”. The huge downside of this approach is that you don’t actually show the player any of these physics in motion, you merely break the mission down into step-by-steps. The best approach may be a combination of these two, like ”Lunar Lander” in that you manually control the lander through some aspects of its descent, and then ”pick your own adventure” in what happens at certain stages of the descent.

The choice of which platform to provide the project on will be investigated through the course of development, however, it is most likely that the game will be built for the Windows operating system.

With an idea of what the gameplay should resemble, this can then be broken down into distinct tasks to be completed.

2.2.1 Task Breakdown

2.2.1.1 Learning Godot

Before any other task can be completed, Godot needs to be learnt to a sufficient standard in order for the first task to be completed. It is hoped that sufficient knowledge and proficiency will be acquired in each task, which will enable the completion of the next. To start, completing the ”Getting Started” section in the Godot documentation will give a good basis to build upon [35].

2.2.1.2 Core Mechanics

Replicating the mechanics in Lunar Lander is a good place to start for this task. Controls such as thrust and rotation should be implemented similarly to how they are in Lunar Lander. As well as this, adding a conditional statement that takes the speed of the lander upon collision with the surface, and if the speed is higher than a certain amount the lander is ”destroyed”. These mechanics form the core of a Lunar Lander game, the player has a thruster to slow their speed and should they hit the ground too fast, they will destroy their vehicle. The aim being to land on the surface successfully, which may require a few attempts. The physics at this point will be determined by defaults set in the physics engine of Godot.

2.2.1.3 Additional Mechanics

These mechanics will work towards moving the current realisation of the game towards something that is distinctly different from Lunar Lander. In this task, mechanics such as a heatshield, parachute deployment and fuel should all be investigated and implemented. Addition of these mechanics may lead to significant restructuring of the core game.

2.2.1.4 Physics Recreation

With these core and additional mechanics implemented, work can be carried out to replace some of the systems that are handled by the physics engine. The biggest issue of this will be determining how the physics system handles functions like drag on objects travelling through air. From a cursory glance at the physics system, a property called `linear_damping` handles an approximation of air resistance/drag. A higher value is synonymous with a denser altitude with the opposite being true for a thinner atmosphere. The default value of this property is 0.1. In conclusion, this isn't very accurate and relies on what the developers of Godot have used as a basis for this default value.

2.2.1.5 User Interface (UI) Elements

Up until now most information important to a player is likely to be displayed in the console for debug purposes. At this stage, it can expected that UI elements are implemented in a presentable manner. This UI does not have to be final, but it should show that some consideration has went into the design of the user interface. This task should include the addition of menus and the proper navigation controls to go with them.

2.2.1.6 Graphical Assets

At this point in development, placeholder assets will have most likely been used for all objects in the game. This task doesn't necessarily represent the creation or acquisition of assets, however, instead the implementation of these assets into the game. Assets may be discovered or created as part of other tasks, such as in the last task.

2.2.1.7 Addition of Educational/Tutorial Material

This task refers to any educational material that is to be added as part of the educational aspect of this project. This could simply be the creation of help dialogues, a tutorial, or an ingame wiki/encyclopaedia.

2.2.1.8 Finishing Touches

Any finishing touches on the game can be applied here. These may be graphical additions, physics tuning, tweaking of educational material, or any additional changes made to previous steps. If any finalising changes are made, these should be recorded with the reason for the change documented.

2.3 Software Process

An Agile approach will be taken for this project. This is largely in part due to the development process requiring a high amount of prototyping/spike work for each task to be correctly realised. Most software processes that make up the Agile approach are built around work that is completed by a team. The individual nature of this project makes choosing one of these processes difficult. For this reason, it is valid to 'cherry pick' features from these processes that work well for the individual.

2.3.1 Gitlab

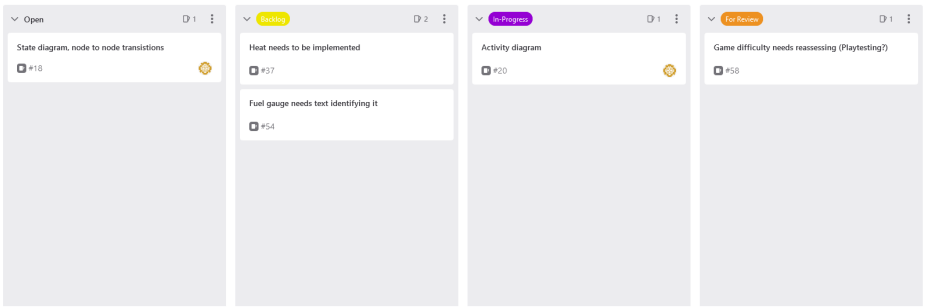
For this project, using the departmental instance of Gitlab will be a useful way of maintaining a coherent version history for the project. A version history is important for tracking changes that have been made and also rolling back these changes should something go wrong. In theory, this allows for work on the project to be conducted on any device, anywhere, with the caveat that these devices must be set-up with the University VPN service.

2.3.2 Gitlab Issues & Kanban

Using the issues feature on Gitlab, a Kanban-esque board can be established. Issues that are posted on this board will be formulated from the main task currently being undertaken. These issues in practice behave like user stories. A task such as 'Core Mechanics' has some requisite subtasks that need to be completed in order to call the parent task completed so that work on other tasks can commence.

When looking at the 'Core Mechanics' task, the sub-task of thrust could then itself be broken down into several 'sub-sub' tasks. Tasks such as "Take user input for thrust", "add upward force on Lander", and "add a throttle control for thrust" would become the user stories entered into the issues board. This 'Divide and Conquer' approach to completing tasks is likely to work when considering the main tasks laid out in the analysis section. There is the danger that when considering smaller tasks with this

Figure 2.4: A screenshot of the Kanban board in use.



approach that the further abstraction of these tasks leads to unnecessary work being

carried out. It is important to keep this in mind when working on tasks and regularly check if the work being done is unnecessary. For the purpose of this report, issues that are posted on this board will be referred to as user stories.

This board also allows the tracking of any unexpected issues that may arise as a result of research or development into the tasks. It is likely that through carrying out work pertaining to these user stories, a previously overlooked aspect of the story may be discovered. Having a board that is tightly linked to the version history of the project allows for these issues to be documented through commits and subsequently turned into user stories.

2.3.3 Delivering Value

Working off a principle of Lean, it is intended that value is to be delivered in each meeting with the project supervisor, who in this context could be seen as the 'customer/client'. For the purposes of the project, showing development of previously outlined theory and technical work in each meeting is valuable. Upholding this principle is likely to result in positive developments being made on the project, as the desire to demonstrate value in each meeting will be present.

2.3.4 Iterative Working

The aforementioned principle works well with an iterative work schedule. The meeting day serves as a good day to conclude an iteration. Utilising the Kanban board, items can be assigned to a 'backlog' list which then can be pulled through into an 'in-progress' list. The fact that tasks may be broken down into smaller user stories, means that it may be more efficient to assign this stories to a day of work, or rather a 'micro-iteration'. In theory this would mean that if each iteration is a week long, each day serves as a 'micro-iteration'. This method will have to be evaluated through the process, as it has the potential to be harmful to the development to the project. However, the desired effect is that this change to the agile iterative cycle allows for a more efficient work flow.

This altered workflow could look like this. Sub-tasks from 'Core Mechanics' are assigned for the week, thrust and lander rotation. These are broken into six separate user stories, which can then be assigned into three-four hour daily sessions, consisting of the 'micro-iteration'. It is likely that in this example, work may be completed early, in which case more items could be pulled into the 'backlog' from a pool of 'open' stories.

2.3.5 Testing

Throughout the course of development it is highly likely that unintended behaviour will be observed. To avoid getting stuck into troubleshooting of this issue, it will be best for the time being to ensure that the issue is an isolated one. In the context of development this means that the issue exists independent of continued development or currently implemented mechanics.

An example of what would be considered an isolated issue is the lander object reacting strangely when colliding with the surface whilst the heatshield reacts as expected. This implies that a property of the lander is causing the issue and nothing intrinsic to the game e.g. the physics engine. In the instance that an issue impedes the progress of further work, brief testing can be carried out to identify a possible cause.

If no solution is found, the feature housing the issue is to be commented out until more thorough testing can be accomplished. This testing will take the form of a testing table document, in which test inputs are recorded, the expected outcome, and the actual outcome. This enables the identification of edge cases and unintended behavior which in turn should allow for the fault to be better identified.

Chapter 3

Software Development and Implementation

3.1 Learning Godot

This section serves to provide a background on the specifics of Godot and what was learnt during this task.

3.1.1 Scenes, Nodes, and Signals

The learning process was begun with reading the beginning of the Godot documentation. The documentation has a section that provides an overview of the key concepts present in Godot. The first concept presented is ”Scenes” [34]. Scenes represent items in a game. A scene can be anything from ”...a character,...a menu in the user interface,...an entire level...” [34]. A Scene is made up of nodes. Scenes can also be nested, for example, a house scene might have a nested bathroom, kitchen, and bedroom scene encapsulated within.

The next concept presented is nodes. Nodes are the ”smallest building blocks” [34]. Focusing on the house example mentioned above, in the bathroom scene the expected nodes for this scene would be; a toilet node, sink node, shower node, door node, etcetera. The Scene Tree is the next concept to be presented. In this tree there is always a root node. From the root you have other nested scenes that make up the branches in the scenes, with the nodes inside of these scenes forming the leaves. It should also be noted that a parent/root node can always access the properties and methods of its children.

The final concept presented is Signals. Signals are emitted by each node whenever a certain event occurs. Using the house example, imagine the house’s lighting functions using Godot Signals. When the light switch node is pressed, a signal is emitted. This signal is then interpreted by the light bulb node as ”turn on” turning the light bulb on. These examples are abstractions of these systems. However, they demonstrate a clear

idea of the structure of games developed in Godot.

3.2 Core Mechanics

This section will discuss the work completed, the scenes created and the code written that allowed for the completion of this task.

3.2.1 Lander Scene

To develop the mechanics, an understanding had to be formed of how the Lander object interacts with the environment and responds player inputs. In Godot there exists two primary nodes used for player objects, RigidBody2D [38], and CharacterBody2D (KinematicBody2D in Godot 3.1) [28]. These two nodes are both intended to be interacted with, whether that be through the physics engine or through code. Some experimentation was carried out into the possible use of the CharacterBody2D node. The conclusion of brief experimentation is that this node is heavily focused on the interaction with a platform the player moves the node across. This is exactly the mechanic that is crucial to a traditional platformer, however, not so much in the context of Mars Lander. For the project, a RigidBody2D node fits the intended purpose much better.

The initial iteration of this scene featured a RigidBody2D node, with two children nodes, a CollisionShape2D and Sprite2D node. For the meantime, Sprite2D used the icon.png (The Godot Logo) as a placeholder image. The CollisionShape2D was used to draw a shape around the Sprite2D, this allows other objects to properly collide with the object. All object scenes that interact with other objects will be arranged this way as default.

The Godot documentation states that when working with RigidBody2D nodes, the assignment of a bodies position or `linear_velocity` should be avoided whenever possible [38]. Instead the Godot documentation recommends applying forces to the node. This allows the physics engine to calculate the change in position based on physical properties assigned to the object. With the Lander scene setup, work began on the implementation of the core mechanics such as thrust, rotation, and collision with the surface.

3.2.2 Rotation of Lander

To affect the rotation of the lander, torque has to be applied to the lander. This is done through the `apply_torque()` function. This method takes a float as its only parameter. This value controls the strength of the rotational effect. To allow the player to control the rotation of the lander, the change in torque needs to respond to the players input. The implementation of this can be seen in Listing 3.1.

Listing 3.1: The function responsible for handling rotation

```
func rotating(body):

        if Input.is_action_pressed("left"):
                body.apply_torque(10000)

        if Input.is_action_pressed("right"):
                body.apply_torque(-10000)

        body.apply_torque(0)
```

The `Input` singleton is responsible for handling all 'action events' that occur in the course of the program. Action events are triggered when an input is received. The following code snippet, is listening for the event "left" `Input.is_action_pressed("left")`. In this case "left" is the event that has been set in the Input Map, a menu dedicated to setting up keys that trigger action events. When these events are detected by the `Input` singleton, they will return true. Therefore, fulfilling the conditional statement and then applying the torque. The method `body.apply_torque(0)` is used to reset the currently applied torque. Without this at the end of the function, torque would only ever be cancelled out when the user presses the opposite direction.

3.2.3 Thrust Mechanism

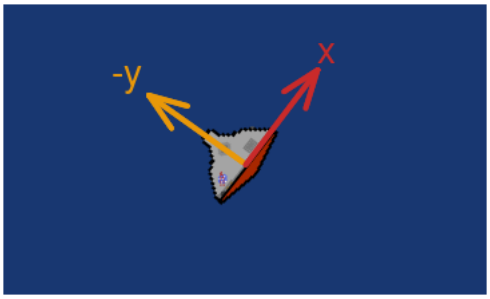
Applying thrust upon the lander was completed in a similar way to how the rotation was implemented. Using the physics function `apply_central_impulse()` a single impulse of force can be applied to an object. By putting this method in a function that is called every frame, the method will apply an impulse continuously. To fully implement the thrust mechanism, a throttle system needs to be established. The idea early on was to use a variable that increased whenever the player held the "up" arrow key, and decreased when the "down" arrow key was held.

This idea was further developed by using a UI node called VSlider. VSlider by default responds to the up and down arrow keys, but only when it has focus [31]]. Taking the current value of the VSlider as a percentage of the thrust allows for the user to operate the thrust function anywhere between 0 and 100%. The benefit of this solution is that no code has to be run to ensure consistency between the actual thrust and the displayed

thrust.

The final iteration of the thrust mechanic utilises the current value returned by the VSlider. This is done through the use of a signal, that passes the current value to the Lander script whenever the value changes [36]. This thrust is then applied to `-transform.y`. By applying force at this point, force is always applied upwards from wherever the bottom of the Lander is. Figure 3.1 demonstrates how this system works in the context of the game.

Figure 3.1: A diagram demonstrating where the force is applied with respect to the lander.



To get the desired effect, the value received from the VSlider was then divided by 80. This number was reached after trial and error as it made the force applied sufficient enough to land whilst still offering a challenge to player. Lowering this value would result in a stronger thruster, whilst increasing it would result in a weaker thrust. This is one setting that could be tweaked to make the game harder or easier.

3.2.4 Lander Collision

Once these two systems were in place, it was time to consider how the game would end. In Lunar Lander, when the lander hits the surface too fast it gets destroyed. When the lander lands within a certain speed threshold the player succeeds. Something similar was pursued in the context of Mars Lander. The current and only failure condition would be colliding with the surface too fast. The only way to win would be colliding with the surface whilst going under a certain speed.

The initial prototyping of this looked into recording when the Lander had collided with Surface, through using an Area2D node that extended slightly above the surface object. When the lander object entered the Area2D node a `body_entered` signal would be sent from the Area2D node. Underneath this signal function would be the conditional statement to evaluate the current lander speed.

In testing this performed adequately, however, it was discovered that if an object that had been attached to the lander was detached this in turn would trigger the `body_entered` signal upon collision with the surface. A more elegant way of implementing this system was to instead push the collision detection onto the Lander object. By default the setting to accomplish this behaviour is disabled.

The `contact_monitor` once enabled, was adjusted to one contact. This was due to

the lander object being on different collision layers to other objects that may be attached to it. It would only share a collision layer with the surface object. This method ensured that nothing else could interfere in the collision with the surface. Using the contact monitor allows for the same `body_entered` signal to be sent into the script attached to the lander object.

In the lander script, a check is then performed to ensure that the body entered is the surface object, which when the conditional statement evaluates as true, emits a custom signal `collided` to the main script. Passed with this custom signal is the Lander's current speed upon collision. Originally this feature didn't work. Through the usage of debug messages that got printed to the console, it could be seen that the lander speed was being recorded as 0 upon collision with the surface.

After discussions with users on the Godot discord server [5], an avenue of investigation was offered, the `_physics_process` function. This inbuilt function runs asynchronously to the `_process` which runs for every rendered frame. This means that any code placed in the `_process` function is hardware dependant, faster hardware will render more frames per second. The `_physics_process` is limited to running 60 times a second. This is done so that any code executed under this function is in line with the physics engine which updates 60 times a second also.

The collision detection functionality provided by the contact monitor feature is tied to the physics engine. The issue had arisen from recording the current lander speed inside of the `_process`. This meant that when the lander collided, the recorded speed may have been the speed a frame or two after the lander had collided and came to a stop. Rectifying this issue was trivial, however the lessons learnt about how the `_physics_process` and `_process` functions work, were certain to be valuable moving forwards.

3.2.5 Finite State Machine

As these three components were combined to form the core mechanics of the game, they were housed in a main script that was conditional in nature. The overall design featured conditional statements to trigger and track the events in the game. This current design was clunky and cumbersome to work with, as small changes at one part of the code led to large changes in how code behaved. At this point in the design it was clear that an alternative was required.

Discussion with the project supervisor revealed that there was an alternative method that could be implemented to handle this system. This alternative method was a Finite State Machine or FSM. Research was carried out into whether or not this design pattern could replace the current conditional system in place [40]. This research yielded promising results, with the most promising concept being that behaviour could be separated into states. By having behaviour under these states, the only conditional statements required in theory were to adjust the game state. At this point in development a formal design had not been established.

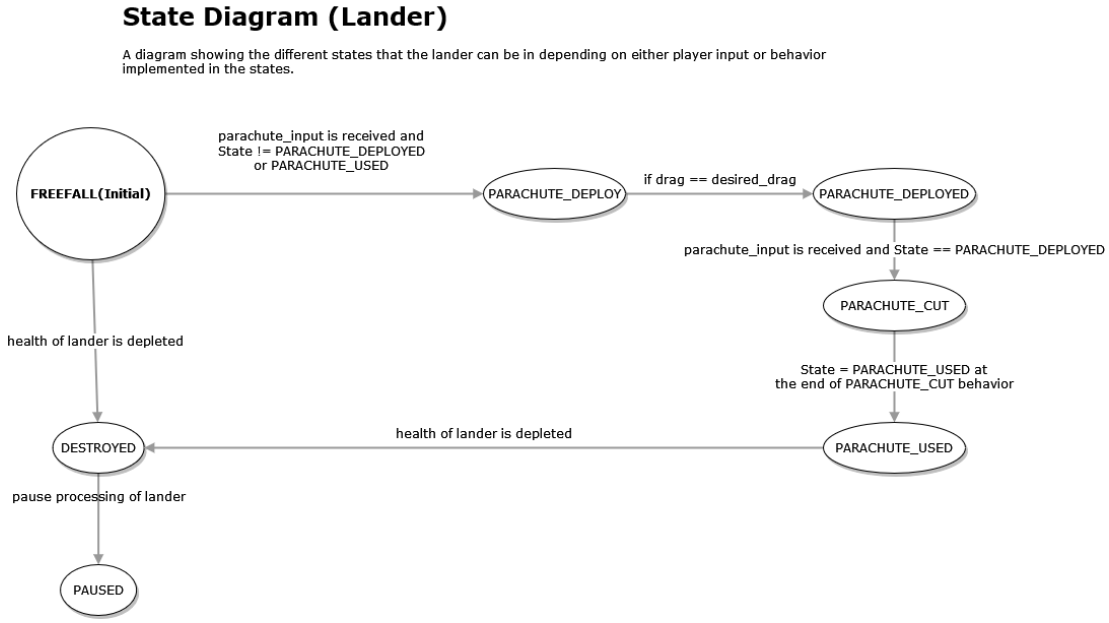
3.3 Additional Mechanics

In this section, investigation into what will make Mars Lander different to Lunar Lander will be carried out, with this resulting in the implementation of additional mechanics.

3.3.1 Finite State Machine Design

At this point in the project, work had already been undertaken to move the structure of the game away from that of conditional statements. This work was continued under this task. As a preface to the work that would later be carried out, a state machine diagram was established. It should be noted that at this point in development most of the states shown on the diagram are subject to change and some are simply placeholders for planned features.

Figure 3.2: State diagram for the state machine in the main script.



The design as shown in Figure 3.2 clearly shows the shift from viewing the game as a series of conditional statements to a collection of states. In the following sections, this state machine design will be used to define more failure conditions which will put the game into the `DESTROYED` state. How this design was adapted can be seen below in Figure 3.3.

Figure 3.3: The code adaptation of the design shown in Figure 3.2

```
## Enum which holds all of the States the game can be in.
## A state variable is utilised to keep track of the current state of the game.
enum State {
    FREEFALL, ## Default state. The game will start off in the freefall state. The behavior attached to this
    ## state tracks the speed of the lander, and relays information back to the user when it is safe
    ## to deploy the parachute.

    PARACHUTE_DEPLOY, ## Generates the parachute instance. If the speed is currently too high for the parachute
    ## to be deployed, the game transistions to the PARACHUTE_CUT state.

    PARACHUTE_CUT, ## Removes the connection between the parachute and lander.Sets the node b of the
    ## PinJoint2D to "", nullifying the connection between the two rigidbodies.

    PARACHUTE_DEPLOYED, ## Relays textual feedback to the user. Acts as an "idle" state whilst the
    ## parachute object is connected to the lander.

    PARACHUTE_USED, ## After parachute is either cut, or detached, this state becomes the new FREEFALL.
    ## This is to insure that the PARACHUTE_DEPLOY state can only be accessed once.

    SUCCESS, ## Upon colliding with Surface, the landers velocity is evaluated. If the velocity is below a
    ## certain value, this state is transistioned to.
    ## This state is passed to the PauseMenu script which displays the appropriate information
    DESTROYED, ## Upon colliding with Surface, the landers velocity is evaluated. If the velocity is
    ## above/equal to a certain value, this state is transistioned to. This state is passed to the
    ## PauseMenu script which displays the appropriate information is displayed.
    }
## State variable to track current game state.
var _state = State.FREEFALL
```

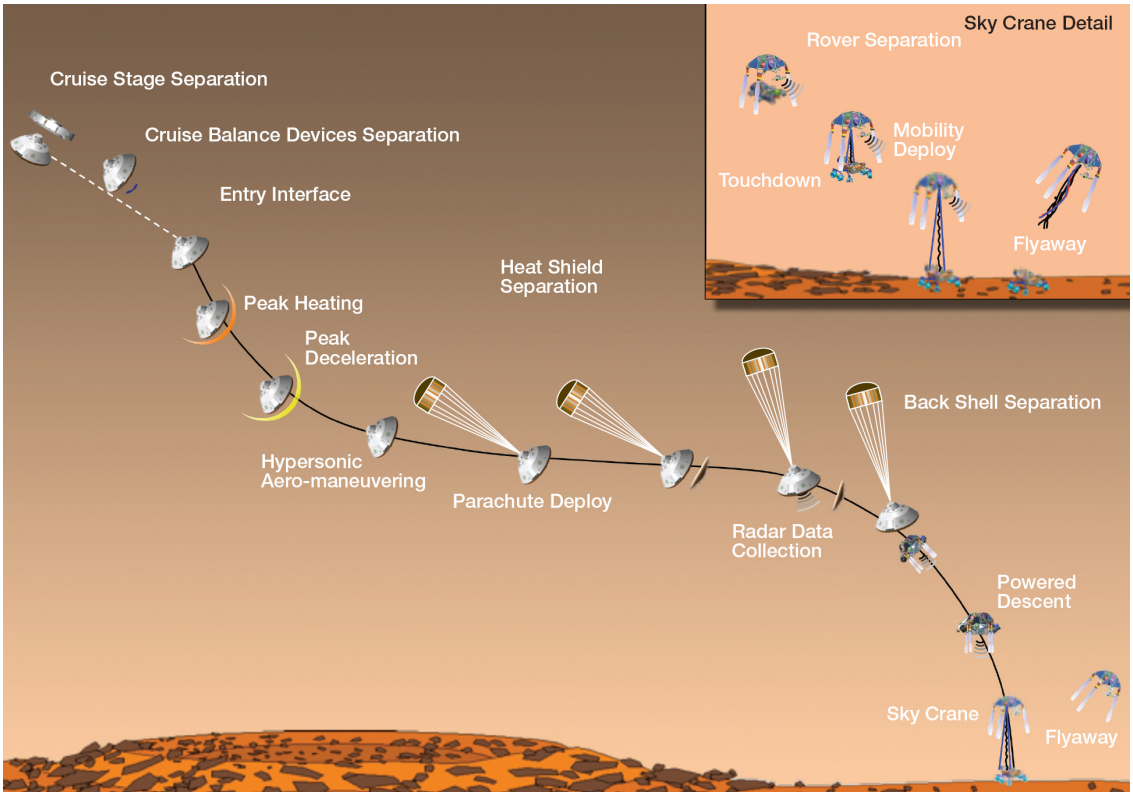
3.3.2 Recreating Mission Stages

Upon reaching the orbit of Mars, typically three main stages need to be considered in order to land successfully. These stages are as follows; Entry, Descent, and Landing (EDL). An aim of this task is to devise how these stages will be recreated in the game.

3.3.2.1 Entry

Referring to Figure 3.4 it can be seen that the EDL sequence starts with the separation from the cruise stage, after which cruise balance devices are jettisoned from the craft. The figure doesn't make it clear, but these two features of the entry phase take place around two thousand kilometers above the 'Entry Interface' stage. The time it takes to go from separating from the cruise stage to interfacing with the atmosphere is too high for a game that is intended to be played at an outreach event. Focusing on the 'Peak Heating' phase of the EDL process, by implementing a heatshield and heating mechanic will satisfy this stage of the game.

Figure 3.4: An image showing the Entry, Descent, Landing stages of the Curiosity mission [12].



Following the process outlined in Section 3.2.1, a heatshield scene was created. This scene would then be instanced inside of the Lander scene, and connected to the bottom of the Lander through the use of PinJoint2D nodes. This node allows the connection of

two RigidBody2D nodes. Using one PinJoint2D would mean the heatshield pivot around the single joint. Utilising two ensures the connection between the two bodies is stable.

With the heatshield in place, work on a heating mechanic could begin. Discussions were had with the project supervisor over the level of accuracy that should be provided by the heating mechanic. From this discussion it was agreed that the accuracy of this mechanic wasn't highly important as it was something the player wouldn't necessarily be able to impact. Nevertheless, the conclusion of the supervisor was that if realism can be established without impacting the players experience then this should be sought after.

As had been established in Section 2.1.4, atmospheric heating occurs due to the compression of gas in front of the heatshield. To give the same effect, a calculation which returned a value proportional to current density of the atmosphere and the velocity of the heatshield was required.

Listing 3.2: A code excerpt showing how a heat rate was established

```
var velocity = state.get_linear_velocity().length_squared()
var density = state.density
var heating_value = 0.5 * density * velocity
var heat_rate = 0.001 * heating_value
```

As can be seen in Listing 3.2 above, the physics basis for this is loose at best. An attempt was made to try and mirror real-life heating calculations. However, in practice alterations were made to the calculation so that desirable values were returned that could be used in the game. The complex nature of these concepts was mentioned in a meeting with the project supervisor. Both the author and supervisor agreed that it is a complicated concept to recreate.

For the purposes of the game, this calculation serves its purpose. It generates a heat rate which can then be subtracted from the variable `heatshield_health`. This variable serves to track the current status of the heatshield. Should this value drop below or equal zero, then it is regarded as 'compromised'. This is where a failure condition can be defined for use in the state machine outlined in Section 3.3.1.

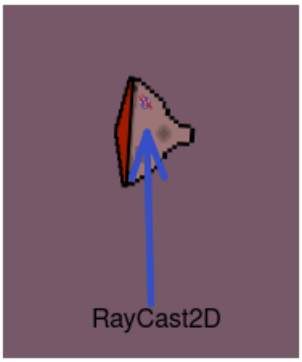
When the heatshield is compromised, a conditional statement placed in the `process` checked for when a boolean variable named `heatshield_destroyed` evaluated to true. Upon which, the code in Listing 3.2 was then ran on the lander object. If the `heat_rate` returned by this code was higher than 0.1, then the lander would also be considered destroyed, leading to the state variable of the game being assigned to `State.DESTROYED`.

During the peak heating stage of a mission, the heatshield bares most if not all of the heating effect of atmospheric entry. Due to this, another failure condition needs to be added. Should the player rotate the lander so that the heatshield is no longer facing downwards, the backshell of the lander would be exposed to the heating effects. In real life, the backshell of the lander would be designed to withstand some heating. However, to emphasise the importance of the heatshield, punishing the exposure of the backshell is necessary.

A method to determine whether the backshell was exposed needed to be developed. The first attempt at devising a solution used a RayCast2D [37]. This node casts a

directional ray, which then reports where and what the ray has collided with. The concept was to use this node, 100 pixels below the Lander object, and have the directional ray pointed at the Lander.

Figure 3.5: An image showing how the RayCast2D node would be set up.



The RayCast2D should report the object it collides with the theory being, if it collides with the backshell, then the backshell is exposed to the atmosphere. This is demonstrated by Figure 3.5, in which the RayCast2D node is independent of the rotation of the lander. In theory this solution should have worked, however, in practice the added complexity of collisions meant that the RayCast2D node did not always return the correct body. Because of this, a simpler solution was devised.

There exists a method that can be called on objects that inherit from Node2D. This method is called `get_global_rotation_degrees` and it gets the rotation of an object in degrees rather than radians. If the value returned from this method is bigger than 45 or smaller than -45 degrees, the backshell is seen as being exposed.

When the backshell is exposed, and `heat_rate` is higher than 0.1, a variable called `backshell_visible` will add another variable `delta` to itself. The `delta` variable comes from the `_process` function and represents the time in seconds from the last frame being rendered. When `backshell_visible` is greater than or equal to 5, the game is put into the `DESTROYED` state.

3.3.2.2 Descent

Once the lander has made it through the peak heating stage and subsequently the peak deceleration stage a parachute stage is required to additionally slow the lander down before a safe landing can be attempted. Early on in the process it was decided that the parachute should be deployed and cut with the same button. This was done in an effort to cut down on the possible keys the player would need to learn/remember over the course of the game. Doing this proved to be more difficult than previously thought. This was due to attempting to implement this mechanic through the usage of conditional behaviour.

Listing 3.3: Pseudocode that shows how the system of parachute instancing was carried out before use of FSM

```
if input_pressed(parachute) == true OR parachute_deployed == true
   parachute_deployed = true
       if instancing == true
           generate instance of parachute
           instancing = false
```

What resulted was a chain of conditional statements and usage of boolean variables to track the occurrence of events. Frustration with how this system worked and how difficult it proved to change behavior was the inspiration to seek another system, such as the previously mentioned FSM. How this system functioned before usage of the FSM is shown in Listing 3.3. This pseudocode was written by looking at early prototypes of the game through Gitlab version history. The pseudocode shown in Listing 3.3 primarily works to ensure that only one instance of the parachute can ever be generated through the course of the game.

This chain of conditional statements highlighted in Listing 3.3 was able to be reduced down to a single conditional statement. This statement checks that the current state of the game isn't a state that would directly conflict with the `PARACHUTE_DEPLOY` state. This check is performed once the input for parachute deployment is received. After this state has been accessed once, the game state would remain in `PARACHUTE_DEPLOYED` until the spacebar was pressed again.

At this point a second conditional statement would check if the game was in `PARACHUTE_DEPLOYED` and put the game into `PARACHUTE_CUT` state, which housed the behavior to disconnect the parachute from the lander. Following this the game would then be placed into the `PARACHUTE_USED` state, a state that conflicts against the `PARACHUTE_DEPLOY` state.

3.3.2.3 Landing

As most of the landing phase has been previously covered in Section 3.2, this section is going to focus on the additional mechanics added to this stage. Currently, the player is able to use the thrust indefinitely. The aim of this task is limit the amount the player can use the thrust feature. This will be done through the implementation of a fuel variable. By subtracting the current value of the thrust from a `lander_fuel` variable with a value of 1000, a vague system of a thruster which burns fuel was created. When `lander_fuel` was less than or equal to 0, then the force was no longer applied.

The surface that the lander lands upon is an important aspect of the Landing phase. Currently in the game, is a StaticBody2D node. This node inherits from the same PhysicsNode2D as RigidBody2D, however, has no methods for applying forces associated with it. In order to make the game more interesting, this surface scene will be changed from being a flat rectangle. The goal is to create a randomly generated map each time the game is launched. Work was carried out into utilising the `FastNoiseLite` class to provide noise generation functionality [29]. This class is able to generate a 1D array of noise, which can be accessed via the `get_noise_1d()`

method. The for loop shown in Listing 3.4 was used to assign points in a Polygon2D shape. The Polygon2D shape accepts a `PackedVector2Array([])` or essentially an array full of positions in vector space.

Listing 3.4: A code excerpt showing how the surface was randomly generated

```
for x in i:
            width += 1
            height += noise.get_noise_1d(x)
            polygon.append(Vector2(width,height))
            x +=1
```

It was possible to assign the position of x to whatever the currently iterated number was and then the y to whatever number was pulled from noise generator. CollisionShape2D accepted the same `PackedVector2Array([])` meaning creating the collision for the surface involved copying the array onto the CollisionShape2D node. This system was dependant on the built-in pseudorandom number generator being reset upon startup, as otherwise the seed for the noise generator would remain the same, and likewise the output would always be the same. The end result of this system can be seen below in Figure 3.6.

Figure 3.6: The randomly generated surface scene.

3.4 Physics Recreation

As mentioned in the task breakdown, Godot does have an approximation of drag. Investigation was carried out to identify whether or not this approximation was appropriate for the project, or whether it would be worth creating a bespoke approximation of drag.

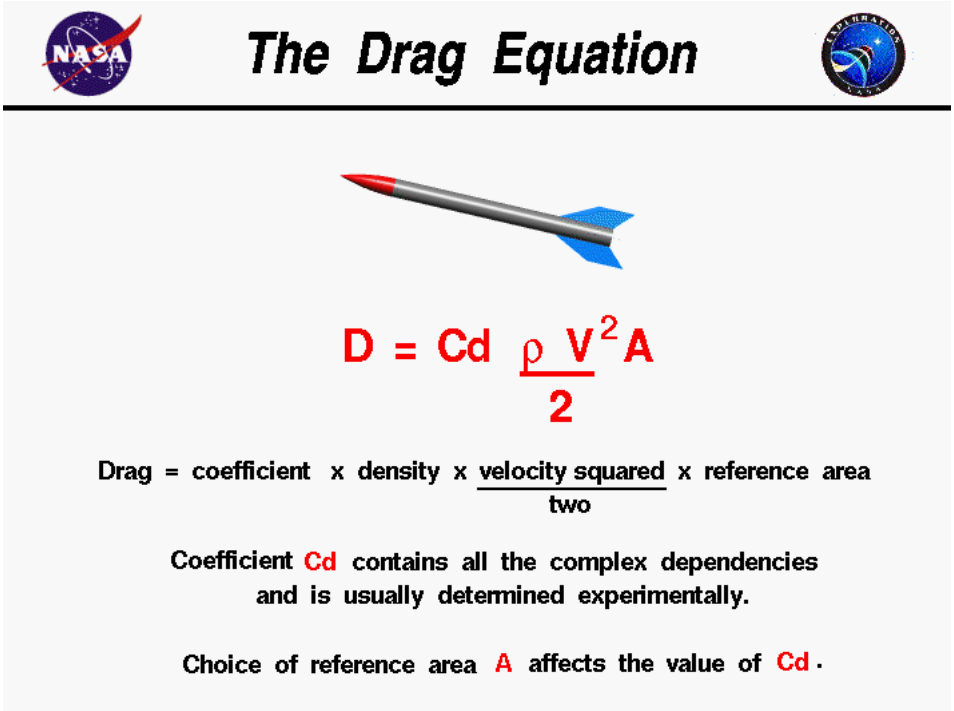
3.4.1 Drag

When using Godot with basic parameters the gravity will be set to 9.8 and `linear_damping` will be set to 0.1. The property `linear_damping` is utilised by Godot to slow an object down over time. In order to fully test the limits of any possible drag representation, a parachute was first made. With a parachute, the effect generated upon the lander due to the drag is known. The parachute upon deployment should slow the lander down. A tutorial was followed to create rope from several RigidBody2D nodes upon which the Lander and Parachute would be attached [24].

The first method that was investigated involved altering the `linear_damping` value inside of the `PARACHUTE_DEPLOYED` state. Increasing the value of this property resulted in the lander slowing upon deployment of the parachute. Resetting the value to the default inside of the `PARACHUTE_CUT` state made the Lander object accelerate again. However, it also made the Parachute object accelerate as well. As a casual representation of drag this method works. To simulate anything more complex i.e the Lander and Parachute object descending at different rates due to them being detached from each other, a bespoke system would be required.

Investigation into this second method began with analysis of the drag equation. As can be seen in Figure 3.7, the drag equation is made up of four parts. A coefficient, a density, the velocity squared (divided by 2), and a reference area would be required to translate this equation into code.

Figure 3.7: An image showing the drag equation, [11]



3.4.1.1 Density

The surface density of the atmosphere on Mars is 0.020kg/m. To use this value without any changes, assumptions had to be made about other units in the code. For the purpose of simplicity, pixels were assumed to be meters, with a thousand pixels being synonymous with a kilometer. If the player object is starting at the outer atmosphere, before heating occurs, the density at this altitude would not be equal to the aforementioned surface density. For this reason, a solution was required which would return the density for a given altitude. Initially, an interpolation was going to be performed in order to acquire the density for a given altitude.

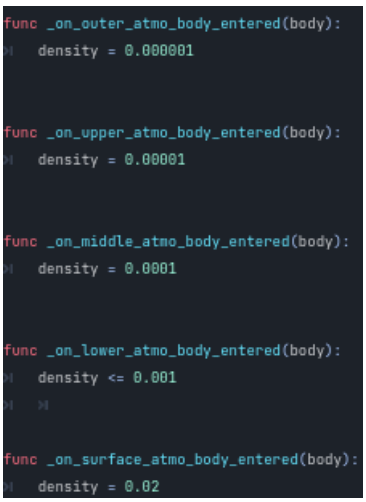
Listing 3.5: The function responsible for interpolating the density given an altitude

```
func calc_density():
        while current_density <= surface_density:
                current_density = surface_density*(EULER**(-1 *
        (object_altitude-0) / 11))
                return current_density
        if current_density > surface_density:
                        current_density = 0.2
        return current_density
```

Listing 3.5 shows how this interpolation was implemented into code. This implementation was reached after talking to users in the Godot Discord server. Initial testing utilised an exponential interpolation to acquire the density. This implementation worked. However, it was computationally heavy, as each frame this calculation had to be performed and then assigned to the three to four RigidBodies that could exist at any one time. Furthermore, due to the way the Surface was generated, tall hills and deep valleys could be generated that would make the interpolation incorrect. Due to these issues, the decision to move towards a simpler solution was made.

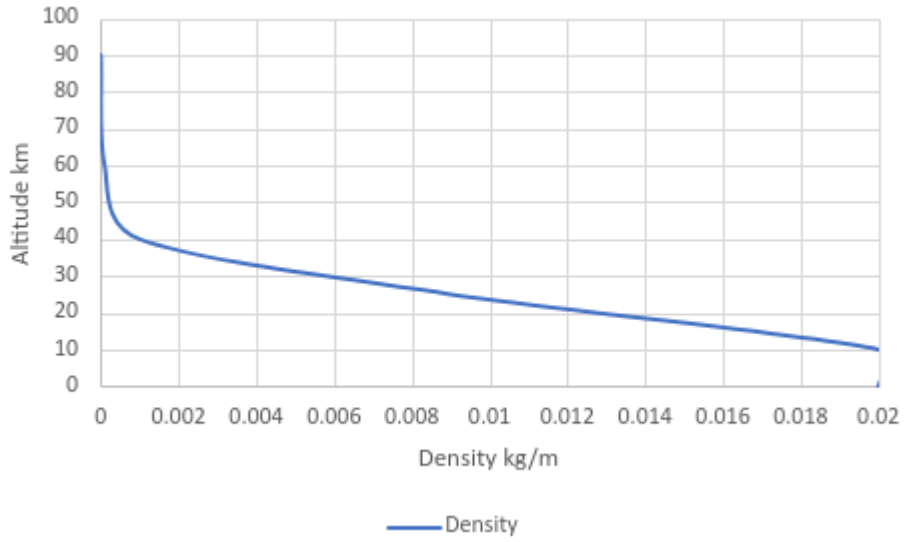
Utilising Area2D nodes, five zones were established in the game. Upon a RigidBody2D node entering the Area2D node, a `body_entered` signal was emitted into the respective RigidBody2D's attached script. Figure 3.8 shows how this system worked with all of the atmospheric zones.

Figure 3.8: A code screenshot showing how the concept of atmospheric zones worked



Using the values shown in Figure 3.8, a curve representing the in-game relationship between altitude and density was made. NASA stated that density decreases as altitude increases [9]. This trend can be seen in Figure 3.9.

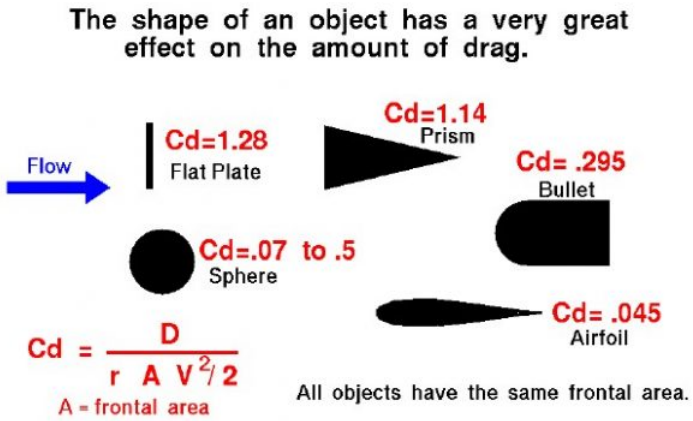
Figure 3.9: The relation between altitude and density in the context of Mars Lander.



3.4.1.2 Drag Coefficient

Drag coefficients are typically formulated from extensive experimentation. Publicly accessible information on the drag coefficients of different Lander components was sparse, so an appropriate analogue had to be found. NASA has an article covering the drag coefficients of different shapes [23]. An image from this article stood out and is the image featured in Figure 3.10. Using this image, a rough approximation can be made for the drag coefficients of each physics object.

Figure 3.10: An image showing the drag coefficients of several different shapes [23]



3.4.1.3 Area

The final variable was derived from research that was centered around the different dimensions of the InSight mission. This mission was picked as there was a lot of information based on the specifications of each component part of the landing stage. Looking at the InSight mission press-kit, it can be seen that the diameter of the heatshield is 2.64 meters and the parachute diameter is 11.8 meters [26]. Using these numbers to generate areas of these objects gives 5.47 meters and 109.36 meters respectively. This forms a basis for which reference areas of objects can be estimated.

3.4.2 Translating Drag into Code

Listing 3.6 below shows how all of the the aforementioned sections come together, except the velocity squared which can be seen being retrieved at the beginning of the function. Similarly to how thrust is applied, `-air_resistance` is applied via the `apply_central_impulse` method. It should be noted that the minus in the variable exists so that the drag force is applied opposite to the direction of travel.

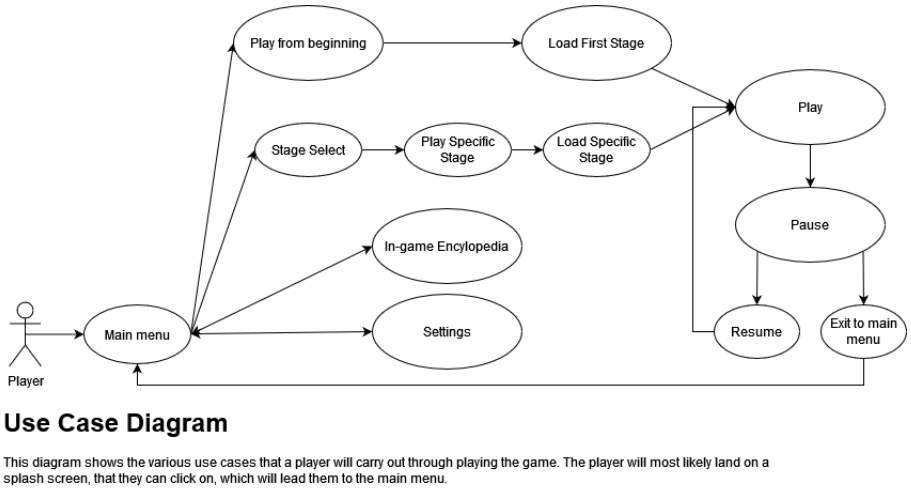
Listing 3.6: The drag equation as it functions in the game.

```
func drag(state):
        var x =  int(state.get_linear_velocity().x)
        var y = int(state.get_linear_velocity().y)
        x^2
        y^2
        var squared_velocity = Vector2(x,y)
        air_resistance = (CD * density *
    (squared_velocity/2) * area)
        state.apply_central_impulse(Vector2(-air_resistance))
```

3.5 User Interface Elements

This section aims to briefly cover the most important aspects of the UI that were created. In the initial stages of the project, the use case diagram shown in Figure 3.11 was created. Whilst not all of these use cases were implemented in the final release of the game, the diagram provided valuable insight into what UI needed to be created.

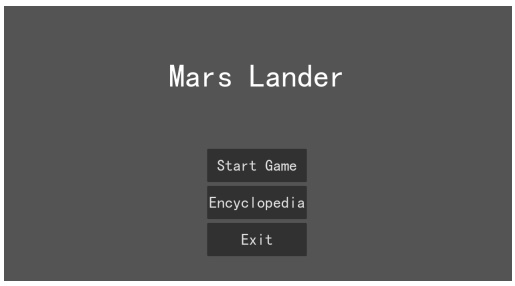
Figure 3.11: An image showing a use case diagram, that covers a lot of the UI elements.



3.5.1 Menus

It was decided that a standard menu layout should be created which can then be adapted for different situations. This method ensures consistency between UI elements. The first menu created was the Main Menu. This can be seen in Figure 3.12. Creating this menu allowed for a second menu to be created, the Pause Menu. The Pause Menu is instanced whenever the player presses the ”Escape” key.

Figure 3.12: The main menu of Mars Lander



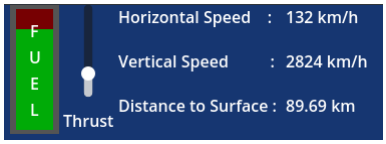
On the Pause Menu, the only difference to the Main Menu is the title of the page. This Pause Menu was then used to create the Game Over screens the player would see when they failed the game or succeeded. The title of the page changed depending on

what state the Main script was in. This was done so that unique information about why the player had failed the mission could be displayed to the player.

3.5.2 Telemetry Data

To give the player a better idea of where they are in relation to the surface, it was decided to display telemetry information to the player.

Figure 3.13: The instruments panel implemented into Mars Lander.



This telemetry data would include Altitude, Horizontal Speed, and Vertical Speed. This is clearly shown in Figure 3.13. Both speed indicators were implemented through getting the current `linear_velocity` of the lander. The velocity of this property is returned in pixels per second. This could be converted to kilometers per hour by multiplying the returned value by 3.6 and then rounding the value to 1 decimal place.

Implementing the altitude was a more complex issue. Using the `distance_to()` method on two objects would yield the distance in pixels between the two centers of each object. This solution worked when the lander was a long distance away from the surface node, but when landing, the returned altitude would be incorrect. Investigation was carried out on the Godot subreddit to ascertain a better solution to this problem [6]. The consensus was that a RayCast2D node should be used to return a collision position, which could then be subtracted from the position of the Lander. This resulted in an altitude that reflected the true distance to the surface. This was especially important with how the surface was generated.

3.6 Graphical Assets

This section aims to highlight the method used to create graphical assets. Whilst carrying out prototyping of scenes, using placeholder textures is essential to get a feel for how these scenes will work in the actual game space. Creation of these graphical assets was done through pixel art. Pixel art is an art style that the author has experience working with. This art style also has the added benefit of being very light on performance due to the small sizes of sprites, limited colour palette, and decreased complexity. Figure 3.14 shows assets that were created using this art style.

Figure 3.14: Assets created for Mars Lander

3.7 Addition of Educational/Tutorial Material

At this stage in development, the game is considered functional. However, it is lacking in educational content. In this section the different methods of implementing educational material will be investigated.

3.7.1 Part Selection

In order to add variation to each landing attempt and to teach that considerations are need to be made about what parts get used, a part selection screen was created. This screen is overlapped on the Main game scene. When the continue button is pressed the overlay is made invisible.

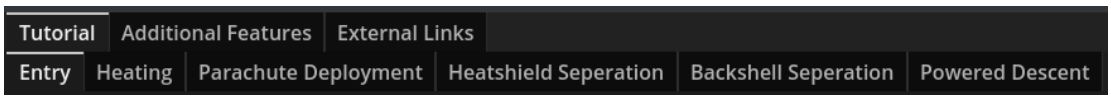
Listing 3.7: The changes associated to the player choosing the thick heatshield

```
2:
 $UI/Lander/HeatShield/Sprite2D.modulate =
 Color(0.27889686822891,
 0.25785693526268,
 0.27853071689606)
 heatshield_health = 100
 $UI/Lander/HeatShield.mass = 50
```

When nodes are invisible, processing on that node is paused. When a different part is selected, this sends a signal to the Main game scene. This signal passes a state variable which is handled by a match statement[reference to match statements]. Each part returns a unique value in the state variable which when matched upon, parameters of the respective part are changed. For example, when the thick heathshield is chosen the colour, the mass, and the health of the heatshield is changed in the main code. This behaviour can be seen in Listing 3.7, with similar behavior extended to the parachute object.

3.7.2 Encyclopaedia

Figure 3.15: How the TabContainer node was used to create an in-game encyclopaedia



As can be seen in the use case diagram in Figure 3.11, an encyclopaedia feature is mentioned. This feature is intended to hold all the information the player needs about the systems and features in the game. It also includes a tutorial that should guide a player into making the correct choices and successfully land the lander. This encyclopaedia was implemented through the use of a TabContainer node [39]. This allowed for each step of the tutorial to be split into different tabs. An overview of how this functionality looked in the game is shown in Figure 3.15.

This feature has a lot of future potential, as in theory, any number of tabs and subtabs could be added to extend the amount of information presented in the encyclopedia. This is something that will most likely be reviewed in the next section.

3.8 Finishing Touches

This task consisted of making final changes to systems which have been previously covered as part of this report. This section will focus on identifying the most notable changes and additions made to the project at this stage.

3.8.1 Heat Overlay

To give the feeling that parts on the lander are being heated, an identical copy of the current Sprite2D was added to the Heatshield and Backshell scenes. This Sprite2D was named HeatOverlay, with the opacity of this sprite being set to 0.

Listing 3.8: Code responsible for changing the opacity of the heat overlay.

```
state.get_node("HeatOverlay").modulate =
Color(0.84313726425171, 0, 0.00784313771874, heat_rate)
```

The `modulate` statement shown above in Listing 3.8 sets the colour of the HeatOverlay node to be red, with the opacity being dependant on the `heat_rate` variable. This resulted in the heatshield looking 'hot' when being heated. This effect is demonstrated in Figure 3.16.

Figure 3.16: The heating effect in Mars Lander

3.8.2 Atmospheric Sounds

To make the game more entertaining and engaging, the addition of atmospheric sound/wind was added. When reaching the `upper_atmo_body` as shown in Figure 3.8, an AudioStreamPlayer2D node [27] attached to the Lander scene would begin playing an audio sample of wind taken from the BBC Sounds website [42]. This audio was set to loop for the duration of the mission. Upon restarting of the game, the sound would be paused, until it was set to play again upon the lander reaching the `upper_atmo_body`.

3.8.3 External Links

In the final discussion with the project supervisor an interesting feature was discussed. This feature was the implementation of external links to the game and allowed for further investigation of the concepts presented in the game. By utilising the `OS` singleton

allowed for the opening of links through the `OS.shell_open()` method. Entering a URL into this method allowed the opening of any link through the default selected browser on the users computer. Currently in the game, there exists two external links. These links include; a link to all the work Aberystwyth University has carried out on the PanCam and all the past and current missions to mars being conducted by NASA.

Unfortunately, if a version of the game were distributed with these links being replaced with malicious links, then this could result in massive security issue. As the application is intended for distribution only within the university, the possibility of this attack happening is low. However, it is still important to consider.

3.8.4 Play Tests

This point in the development marked the first releasable version of the game. This version was called the alpha version and was used to conduct play tests with a few individuals. Players were asked to play the game, selecting the various parts, and then attempting to land on the martian surface. Every player remarked that they were able to land on the surface, with every player saying the difficulty offered by the game was just right. They gave the difficulty a three, with five being considered too difficult, and one being considered too easy.

An interesting result from the play tests was that every player gave the game a four for educational value, with five being ”lots of educational value”. The most important area of feedback gained from these play tests was how the players felt about the length of the game. Two out of three of the players felt the game’s length was a four, with a score of five being considered ”too long”. The results of these play tests can be located in the ’Testing’ folder of the technical submission.

3.8.5 Time Control

In response to the feedback given on the length of the game, a ”Time Control” mechanic was devised as a solution to this issue. Games such as Kerbal Space Program (See Section 2.1.3.1) have a time warp function, so the idea was not revolutionary. After some cursory investigation the solution was discovered. By controlling the speed of the engine, a speed-up/slow-down mechanic could be implemented. Using the same system from the part selection screen, but changing the parts to be time modifiers such as 1.5 and 2.5. Using a match statement again to control when the engine speed was altered allowed this solution to be fully realised.

There were unintended consequences from this system. For example, when the engine is accelerated the drag is not applied upon the lander correctly. This leads to the lander accelerating more than it should. When the engine speed is restored to realtime, the drag is applied correctly once again and no long lasting effects are felt. There were more effects which were noticed when time control was applied, such as the `heat_rate` variable not being subtracted away from the `heatshield_health` variable correctly. This led to the thin heatshield surviving entry, a behaviour which was undesired. More issues with the Time Control began to appear as testing was carried out on the game.

Due to this, it was discovered that the only stage that was unaffected by the increase in engine speed, was any time spent above the `middle_atmo_body` Area2D. Passing the same signal in as was received in the Lander script, when the Lander object reached the `upper_atmo_body` the second highest altitude zone, time control would be disabled and reset back to realtime. This method allowed the player to skip the 'boring' parts of the game that required little input, whilst leaving the physics simulation unaffected.

Chapter 4

Testing and Final Design

4.1 Testing

The nature of this game meant that trying to formulate unit tests for player input and the results of those inputs would be difficult. Instead, the author tested behaviors and interactions in the game itself. Testing throughout the project was carried out using a regression testing strategy.

This involved performing the aforementioned behaviour testing whenever a major change in the code occurred. Over the course of development four major changes happened. When testing was carried out after these changes, every test that had previously been carried out was rerun and any new tests were added where appropriate. The resultant testing table can be accessed through the 'Testing' folder in the technical submission.

4.2 Final Design

The final design can be seen below in Figures 4.1 and 4.2. This design was reached due to the iterative workflow undertaken throughout the project. As can be seen in Figure 4.1, the design is broken up into components, that when combined form the Main scene. The state machine also changed from the version shown in Figure 3.2 as a result of the iterative process.

With a final version settled on, the game was then exported to Windows and Linux, generating an .exe and .x86-64 release of the game. This was done through the Export menu built into Godot. It was observed that functionality existed to export the project to HTML5, this could not be investigated deeply, but the conclusion was that a hosting solution would be required to fully take advantage of this.

Figure 4.1: A top level view of the design of Mars Lander.

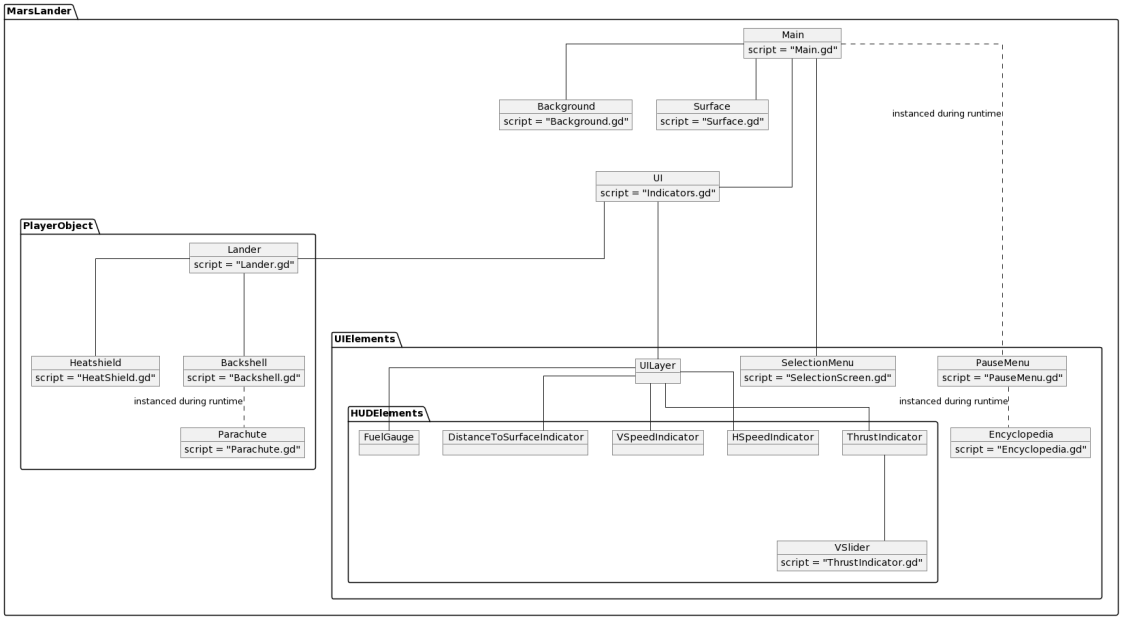
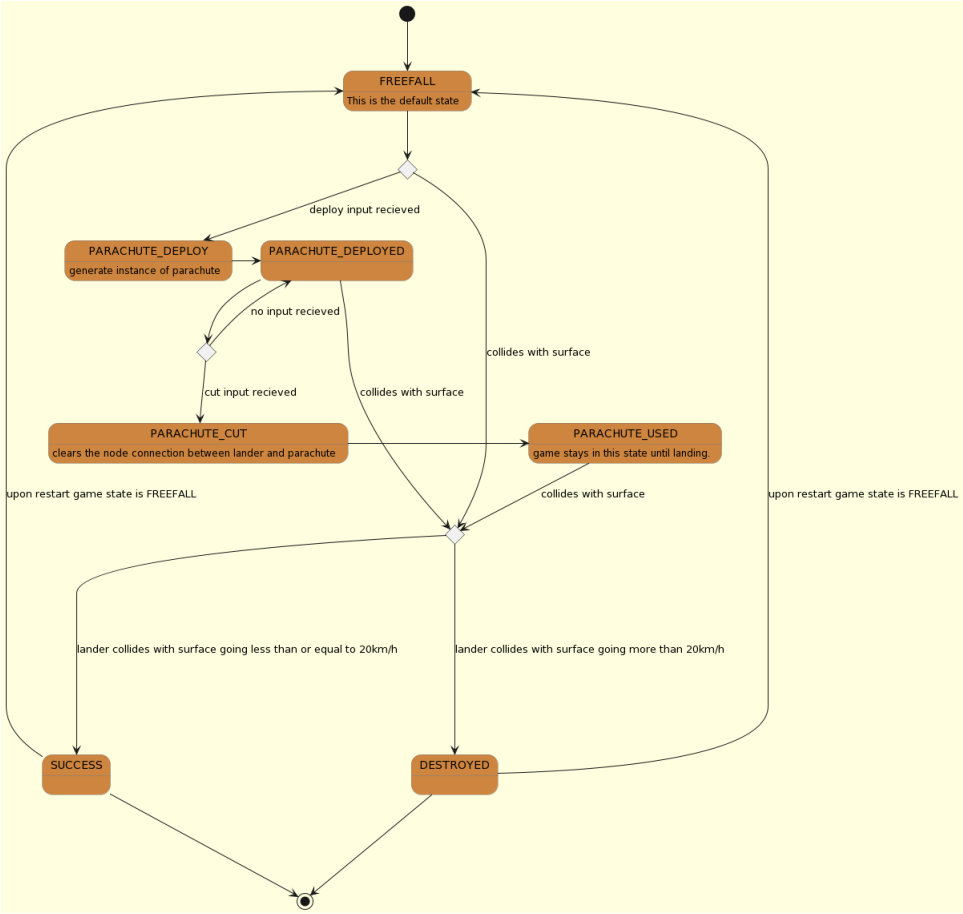


Figure 4.2: Final diagram of the state machine in the main script..



Chapter 5

Evaluation

5.1 Evaluation

In the first few weeks I feel the research I carried out could have been better. In the initial stages of development, a fair amount of time was spent attempting to establish the level of accuracy that the physics in the game would aim to replicate. This was not a waste of time, however, as whilst trying to accomplish this task I became well accustomed to the workflow of Godot. The background and analysis stage was more focused on the game side of the project and the educational implications rather than the martian physics that needed to be implemented. This generally hindered development when work had to be completed on the topic of physics. Over time this lead to physics systems being sidelined, until a breakthrough was had with the implemented drag function.

The decisions that went into the design were made weeks before any formal designs were created. The process that is outlined in chapter one of this report, helped to actually get code into the game and get this code working with other components. However, generally it did not allow time for the design to be considered at every step of the development cycle. I do not necessarily think this is a bad thing, despite it perpetuating the creation of technical debt in the project. This debt was dealt with every time the game reached a state in which it needed to be in a fully working order. The first instance of this happening was in preparation for the Mid Project Demonstration. Code was continually tested until it passed the test cases required for the Mid Project Demonstration. From this point, the same test cases were built upon and used to form a regression based testing strategy.

As the project came to a close more and more fixes to mechanics were done in a dirty and time-cheap way. The objective for future development would be to reduce the technical debt. This would be done by; exploring ways to enhance the game's current states, implementing 'setter' and 'getter' functions to access node properties, reorganizing the file structure, and refactoring all of the aforementioned changes.

One of the biggest downsides of Godot is its small user-base when compared to competitors such as Unreal and Unity. This was not fully apparent when background research was carried out and only became known later in the project. Godot is fully

documented, however, due to the comparatively small user-base, not every thing documented has been demonstrated in a practical way by users. This meant that for the most part investigation into concepts and systems had to be done through the Godot documentation, with questions sometimes asked on the Godot Forums and Discord Server. If I were to make a game similar to this project in the future, I would definitely spend time assessing whether or not Unity would better suit the needs of the project.

This does not mean that my time spent using Godot was not enjoyable. I found the concepts covered in Section 3.1.1 were highly intuitive. This, combined with GDScript's python-inspired indentation format, made the only challenging part of learning Godot was understanding the engine's limitations and reading the relevant literature associated with specific nodes.

As a game, the gameplay loop is intuitive enough that with a few attempts players can successfully land on the martian surface. There are contextual hints presented to the player when they destroy the lander. I feel these hints do a good job at leading the player towards the correct solution.

On a physics basis, the game simulates the physics outlined in the Section 2.2.1.4. The code behind these functions does not always mirror real life. Where the code deviates from real life, the reason behind the final version has been stated in this report.

For educators, ample information is given on how the player can successfully land a lander on the surface of Mars within the context of the game. As a proof of concept, external links were added to the encyclopedia. This external links section could be expanded to include links to more subjects revolving around Mars.

Furthermore, adding the ability for educators to create new tabs and fill these tabs with their own information could provide further value. This would extend the functionality of Mars Lander to be inline with other EDU Games as mentioned in Section 2.1.3.1. Another possible feature that could be added is the ability to change how the player lands the lander. Currently, in the game the only option that exists is powered descent. However, other methods such as gasbags and the Skycrane could be implemented to expand this functionality.

References

[1] " Godot Engine - QA," https://godotengine.org/qa/questions, accessed February 2023.

A link to the Godot Forums where questions are frequently asked.

[2] "3D Vs. 2D: The Eternal Battle to Develop Video Games," https://starloopstudios.com/3d-vs-2d-the-eternal-battle-to-develop-video-games/, accessed February 2023.

An article by Starloop discussing the differences between 2D and 3D game development

[3] "Entry, Descent, and Landing," https://mars.nasa.gov/mars2020/timeline/landing/entry-descent-landing/, accessed February 2023.

An article prepared by NASA before the Perseverance Mission landed on Mars.

[4] "Features - Unreal Engine," https://www.unrealengine.com/en-US/features, accessed February 2023.

A webpage from the Unreal Engine discussing its features

[5] "Godot Discord Server," https://discord.com/invite/zH7NUgz, accessed February 2023.

A link to the Godot Discord Server that the author browsed

[6] "Godot Subreddit," https://www.reddit.com/r/godot/comments/11or4ka/distance_between_two_objects_not_center_of_objects/, accessed February 2023.

A link to the Godot subreddit where the author posed a question relating to how to get an accurate altitude.

[7] "Home - Epic Games," https://www.epicgames.com/site/en-US/home, accessed March 2023.

Epic Games homepage, displayed on this homepage is Unreal Engine. Epic Games develop the Unreal Engine.

[8] “How game engines create great 2D games ,” https://unity.com/how-to/beginner/unity-good-2d-development, accessed March 2023.

A webpage from Unity showing all the things you can do with 2D game development in their software.

[9] “Mars Atmospheric Model - Metric Units,” https://www.grc.nasa.gov/www/k-12/airplane/atmosmrm.html, accessed February 2023.

An article from the Glenn Research Center discussing the atmospheric model for Mars.

[10] “Missions,” https://mars.nasa.gov/mars-exploration/missions/?page=0&per_page=99&order=date+desc&search=, accessed May 2023.

A webpage containing every mission to mars.

[11] “The Drag Equation,” https://www.grc.nasa.gov/www/k-12/rocket/drageq.html, accessed February 2023.

An article from the Glenn Research Center discussing the drag equation.

[12] “Timeline of Major Mission Events During Curiosity’s Landing ,” https://www1.grc.nasa.gov/beginners-guide-to-aeronautics/shape-effects-on-drag/, Aug. 2012, accessed February 2023.

An image showing the EDL of the Curiosity rover.

[13] “Kerbal Space Program - Duna,” https://wiki.kerbalspaceprogram.com/wiki/Duna, Oct. 2016, accessed May 2023.

An image showing Duna as it appears in Kerbal Space Program

[14] “The state of educational games in the Minecraft era,” https://www.gamedeveloper.com/business/the-state-of-educational-games-in-the-i-minecraft-i-era, Oct. 2016, accessed March 2023.

An article from GameDeveloper discussing how much education through gaming has changed since the introduction of games like Minecraft Edu edition

[15] “NASA’s MAVEN Reveals Most of Mars’ Atmosphere Was Lost to Space,” https://www.nasa.gov/press-release/nasas-maven-reveals-most-of-mars-atmosphere-was-lost-to-space/, Mar. 2017, accessed April 2023.

An article discussing the findings of the MAVEN mission.

[16] "ESA and Auroch Digital launch Mars Horizon game ," https://www.esa.int/About_Us/Branding_and_Partnerships/ESA_and_Auroch_Digital_launch_Mars_Horizon_game#:~:text=Mars%20Horizon%20was%20developed%20with,assistance%2C%20gameplay%20advice%20and%20testing., Nov. 2020, accessed March 2023.

An article from ESA which discusses the partnership between themselves and Auroch Digital.

[17] "Kerbal's 'Shared Horizons' launched with real ESA missions," https://www.esa.int/About_Us/Branding_and_Partnerships/Kerbal_s_Shared_Horizons_launched_with_real_ESA_missions, 2020, accessed March 2023.

An article from ESA which discusses the free dlc "Shared Horizons" developed for the game Kerbal Space Program. This DLC allows players to take control of the Ariene 5 Launch vehicle and carry out real world ESA missions.

[18] "Is Unity Good for Beginners?" https://inspirationtuts.com/is-unity-good-for-beginners/, Apr. 2021, accessed February 2023.

An article produced by a youtuber who discusses why unity is a good game engine for beginners.

[19] "Unreal for 2D Top Down RPG?" https://forums.unrealengine.com/t/unreal-for-2d-top-down-rpg/245123, Aug. 2021, accessed February 2023.

A forum post from a user asking if they should use Unreal Engine for their game. The replies mention using Godot.

[20] Atari, "Lunar Lander - Operation, Maintenance, and Service Manual," https://arcarc.xmission.com/PDF_Arcade_Atari_Kee/Lunar_Lander/Lunar_Lander_TM-136_1st_Printing.pdf, 1979, accessed April 2023.

A manual for the Lunar Lander arcade game developed by Atari.

[21] I. Caponetto, J. Earp, and M. Ott, "Gamification and Education: A Literature Review," accessed February 2023.

An article discussing the recent developments in the area of gamification.

[22] B. Edwards, "Forty Years of Lunar Lander," https://www.technologizer.com/2009/07/19/lunar-lander/, July 2009, accessed April 2023.

An article discussing the history and legacy of 'Lunar Lander'.

[23] N. Hall, "Beginners Guide to Aeronautics - Shape Effects on Drag," https://www1.grc.nasa.gov/beginners-guide-to-aeronautics/shape-effects-on-drag/, July 2014, accessed May 2023.

An article covering how shape effects the drag coefficient

[24] M. Hossain, “Ropes 2D in Godot [in 3 Minutes] — with Physics,” https://www.youtube.com/watch?v=0ABq6wL016g, Aug. 2021, accessed March 2023.

A video showing how to make a rope in Godot

[25] E. Howell, “’Mars Horizon’ review: Design rockets, run missions and compete for glory in this addictive space adventure,” https://www.space.com/mars-horizon-space-game-review, May 2021, accessed February 2023.

An article from which an image showing the different decisions posed to the player can be seen.

[26] J. P. Laboratory, “Gamification and Education: A Literature Review,” https://www.jpl.nasa.gov/news/press_kits/insight/launch/download/mars_insight_launch_presskit.pdf/, May 2018, accessed March 2023.

A presskit released ahead of the InSight mission.

[27] J. Linietsky, A. Manzur, and the Godot community, “Godot Documentation - AudioStream,” https://docs.godotengine.org/en/stable/classes/class_audiostream.html, 2014, accessed April 2023.

Godot documentation on how to use the AudioStream node

[28] ——, “Godot Documentation - CharacterBody2D,” https://docs.godotengine.org/en/stable/classes/class_characterbody2d.html, 2014, accessed March 2023.

Godot documentation for the CharacterBody2D node

[29] ——, “Godot Documentation - FastNoiseLite,” https://docs.godotengine.org/en/stable/classes/class_fastnoiselite.html, 2014, accessed March 2023.

Godot documentation on FastNoiseLite

[30] ——, “Godot Documentation - GDScript documentation comments,” https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_documentation_comments.html, 2014, accessed March 2023.

Godot documentation on how to generate Godot super comments.

[31] ——, “Godot Documentation - Keyboard/Controller Navigation and Focus,” https://docs.godotengine.org/en/stable/tutorials/ui/gui_navigation.html, 2014, accessed March 2023.

Godot documentation for focus

[32] ——, “Godot Documentation - Match,” https://docs.godotengine.org/en/latest/tutorials/scripting/gdscript/gdscript_basics.html#match, 2014, accessed March 2023.

Godot documentation for the match statement

[33] ——, “Godot Documentation - Node2D,” https://docs.godotengine.org/en/stable/classes/class_node2d.html#class-node2d-property-global-rotation-degrees, 2014, accessed April 2023.

Godot documentation for the global_rotation_degrees method

[34] ——, “Godot Documentation - Nodes and Scenes,” https://docs.godotengine.org/en/stable/getting_started/step_by_step/nodes_and_scenes.html, 2014, accessed March 2023.

Godot documentation for Nodes and Scenes

[35] ——, “Godot Documentation - Overview of Godot’s key concepts,” https://docs.godotengine.org/en/stable/getting_started/introduction/key_concepts_overview.html#scenes, 2014, accessed March 2023.

Godot documentation that provides an overview of Godot’s key concepts

[36] ——, “Godot Documentation - Range,” https://docs.godotengine.org/en/stable/classes/class_range.html#class-range-signal-value-changed, 2014, accessed March 2023.

Godot documentation for the value_changed signal

[37] ——, “Godot Documentation - Ray-casting,” https://docs.godotengine.org/en/stable/tutorials/physics/ray-casting.html, 2014, accessed March 2023.

Godot documentation on how to utilise the RayCast2D node.

[38] ——, “Godot Documentation - RigidBody2D,” https://docs.godotengine.org/en/stable/classes/class_rigidbody2d.html, 2014, accessed March 2023.

Godot documentation for the RigidBody2D node

[39] ——, “Godot Documentation - TabContainer,” https://docs.godotengine.org/en/stable/classes/class_tabcontainer.html, 2014, accessed April 2023.

Godot documentation on how to use the TabContainer node

[40] N. Lovato, “Finite State Machine in Godot,” https://www.gdquest.com/tutorial/godot/design-patterns/finite-state-machine/, Jan. 2021, accessed February 2023.

An article from Godot tutorial website GDQuest, on how to utilise a finite state machine in games.

[41] J. Peel, “Introducing KerbalEdu: a space program for schools,” https://www.pcgamesn.com/introducing-kerbaledu-space-program-schools/, July 2014, accessed February 2023.

An article discussing the presence of Kerbal Edu in the classroom.

[42] G. B. Sonnex, “BBC Sounds - Wind Atmosphere - open air strong wind atmosphere,” https://sound-effects.bbcrewind.co.uk/search?q=NHU9322987, 2022, accessed March 2023.

Atmospheric sound used in the Mars Lander game.

[43] B. Tristem, “Unity vs. Unreal: Which Game Engine is Best For You?” https://blog.udemy.com/unity-vs-unreal-which-game-engine-is-best-for-you/#:~:text=Unreal%20can%20be%20used%20for,set%20as%20much%20as%20Unity.&text=If%20you%27re%20creating%20a,then%20opt%20for%20Unreal%20Engine., May 2022, accessed February 2023.

An article from Udemy discussing the benefits and drawbacks of Unity and Unreal when compared against eachother.

[44] K. Willaert, “Moonlander: One Giant Leap For Game Design,” https://www.acriticalhit.com/moonlander-one-giant-leap-for-game-design/, Apr. 2021, accessed April 2023.

An article discussing the influence of ’Moonlander’.

Appendix A

Appendix

1.1 Repository README file

Mars Lander Simulation Game

Installation In order to run this game, you must execute the .exe for Windows systems, or the .x86_64 file for Linux.

To edit or open this code in Godot, you must first have the latest version of Godot installed. At the time of writing this version is Godot 4.

You must pull this repsitory, and then set the directory Godot scans for projects to either Release or Prototyping. From here the Godot launcher should detect the projects that can be opened.

It is worth following this tutorial to get accustomed to the Godot Editor.
https://docs.godotengine.org/en/3.0/getting_started/step_by_step/intro_to_the_editor_interface.html