Scarabs Core: The Research and Implementation PhysX and AI Behavior

Aaron McCormack

N00212142

Report submitted in partial fulfilment of the requirements for the BSc (Hons) in Creative Computing at the Institute of Art, Design and Technology (IADT).

Declaration of Authorship

The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Programme Chair.

WARNING: Take care when discarding program listings lest they be copied by someone else, which may well bring you under suspicion. Do not to leave copies of your own files on a hard disk where they can be accessed by others. Be aware that removable media, used to transfer work, may also be removed and/or copied by others if left unattended.

Plagiarism is considered to be an act of fraudulence and an offence against Institute discipline.

Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

The following is an extract from the B.Sc. in Computing (Hons) course handbook. Please read carefully and sign the declaration below

Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their own work.

Declaration

I am aware of the Institute's policy on plagiarism and certify that this thesis is my own work.

Signed: ________Aaron McCormack_________________

Date: __________________30th April 2025________________________________

Failure to complete and submit this form may lead to an investigation into your work.

Abstract

PhysX is the built-in physics engine found within the Unity game engine developed by Nvidia. It is a powerful and open-source SDK that supports scalable simulation and many other capabilities and the primary focus of this thesis project. On top PhysX, there is also an in-dept look at path-finding algorithms used within games, use cases and the evolution of complexity when it comes to AI behaviors. The purpose of my application is to explore how a physics-based game is developed and implemented with the current built-in engine as well as the current possibilities that AI behavior can be explored.

Acknowledgements

I would like to thank and give a heavy appreciation towards my family, friends and my supervisor Cyril Connoly for not only helping give continuous support, feedback and encouragement during times then project weighed heavy on my mind. For that, I heavily thank you all and hope my project has lived up to expectations.

Contents

Introduction

In today’s world, the impact of gaming has made huge strides in terms of technology and understanding of human psychology. They have evolved from simple 8-bit graphics, kilobyte memory and limited storage into an ever-expanding field of study with titles that explore human emotion, act as training simulations in specialized fields or help those connect with a niche community with similar interests. I have been involved within the gaming community for most of my life, from casual to competitive gaming, and developed a keen interest in the processes and logic that happens behind the scenes of final product.

For over 2 years I have been working within Unity, a free game engine, for both personal and professional use in college. Learning basics of multiple areas such as character functionality, shader graphs, 3d modeling, animation, etc. However, the field I lacked most was physics engines and AI behavior and pathfinding. I understood their meaning but not the process and decided that this thesis became the perfect steppingstone for investigating and creating my own application to show what I have learned during this process.

During this thesis you will follow the same steps that I took into learning the algorithms, history and implementation of physics and AI systems within the gaming sphere. You will learn the struggles, problems and solutions I had come across during the investigation. I will describe detailed comparisons to alternative solutions, weighing the pros and cons to explain the reason why one may be chosen over the other. This all done with the intent of bettering the knowledge on such systems and exploring their use cases.

Research

The research conducted during this study was to learn more about the history, evolution and potential development of AI solutions within gaming. Looking at examples from the past, techniques that were developed and expanded, and the potential impact of machine learning that could come to be in the future.

Background

In general, the video game industry has experienced multiple booms in popularity as mainstream consoles became commercially available to the public. With video games history starting back in the 1950's with an automated tic-tac-toe machine, we can also see the development of AI or NPC (non-playable characters) behaviours. From text-based adventure games to modern titles, the development of AI within gaming has been exponential.

Research Questions

With this research report I hope to answer the following questions:

1. How did NPC behaviour change from 1950 to Today?
2. What were the impacts of these developments?
3. What new methods were developed to take advantage of modern technology?
4. What could the future of AI and NPC behaviours look like?

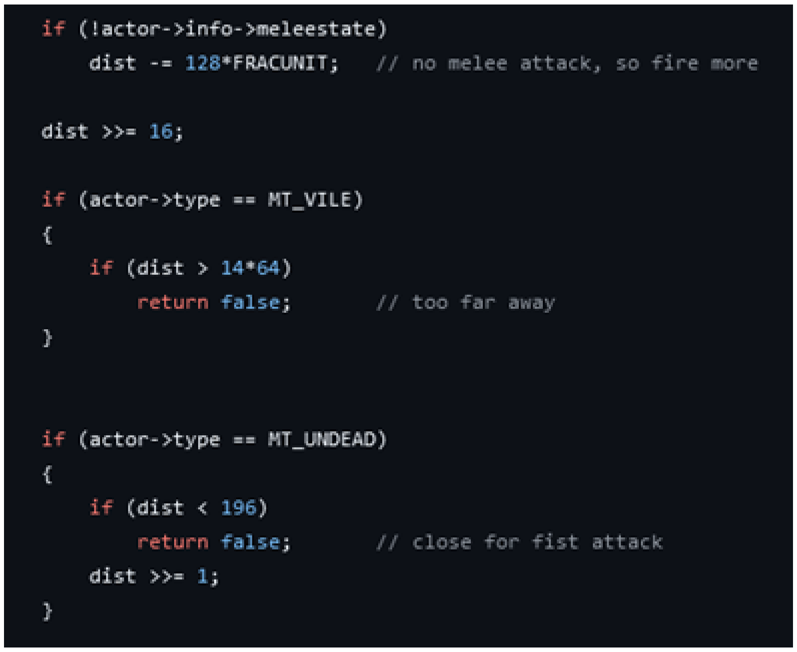
Objectives

The objectives of this research paper are to trace the evolution of NPC and AI techniques within this field, analyse the impacts of the technology in regard to the development of NPC AI, identify key milestones within the history of development and give a personal but grounded opinion on what the future of NPC AI could look like within the next 5 – 10 years.

Evolution of NPC AI Techniques

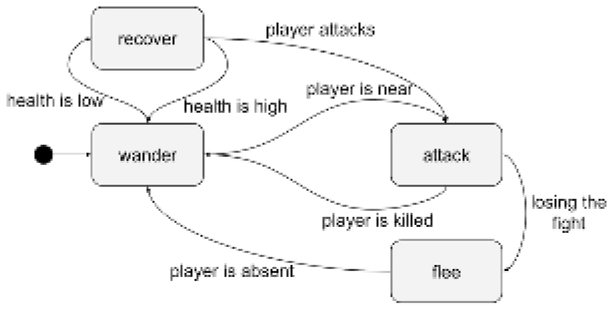
From Simple to Complex

When talking about AI Techniques, it is important to understand how it started and the steps that were taken to evolve the technologies. Starting off with scripted behaviours. These are pre-defined actions and sequences set to perform when a specific criterion was met, be it distance to a player, a button pressed or the level progresses beyond a certain point. These started off as the building blocks for modern NPC AI techniques and can be seen as early as Doom using distances to determine if an enemy should use melee or ranged attacks.



```
if (!actor->info->meleestate)
    dist -= 128*FRACUNIT;   // no melee attack, so fire more

dist >>= 16;

if (actor->type == MT_VILE)
{
    if (dist > 14*64)
        return false;       // too far away
}


if (actor->type == MT_UNDEAD)
{
    if (dist < 196)
        return false;       // close for fist attack
    dist >>= 1;
}
```

As technologies grew so did the methods used, eventually moving on to use what was known as “Finite State Machines” (FSMs). This was the skeleton that would be built upon to create the now popular and complex system known as “Behaviour Trees”. FSMs were predefined states based on a condition that when met or not met, would determine how the AI would react. An example can be seen from the diagram below.



Behaviour Trees build on FSMs by developing a hierarchical system of nodes that represent actions said NPC can perform. A common fault if FSMs is they do not consider the possibility of the NPC being interrupted during an action before it is completed or are required to perform multiple complex tasks in a certain sequence that is not guaranteed to be possible. FSMs need to be programmed and designed for all transitions. Working with Behaviour Trees, these sequences can be made up of relatively easy actions and tasks via the use of nodes and a priority variable, commonly called “weight”.

Pros and Cons

While both FSMs and Behaviour Trees are built with the same methodology and thought process, Behaviour Trees being more advanced by developing a hierarchical system, they both come with advantages and disadvantages. Three main points I wish to compare are the Complexity, Debugging and Performance. We can learn a lot by comparing these three points as they determine what will most likely cause future development and performance issues if not programmed with thought.

Starting with Complexity, FSMs are simple states that are easy to understand by a quick glance and good when needed to implement simple behaviours, however lacking in complexity when requiring the need of more complex behaviours. Behaviour Trees are good at adapting to complex environments by allowing the execution of simultaneous actions however this can lead to more complex and complicated designs when needed for more simpler tasks.

When talking about debugging, it's important to understand which method allows the user to find, understand and solve the issue in the quickest and most efficient manner. FSMs, since each state is its own predetermined action, can be hard to debug due to a behaviour system that has many complex state transitions. However, each action being in its own block of code allows for debugging to be immediately identified to a single block. Behaviour Trees suffer in debugging due to the complexity of the hierarchical system. While it may be visually helpful to determine where an action has failed, it will require further investigation to determine solutions without potentially compromising other states.

Finally, comparing the performance. FSMs suffer in the sense of limited flexibility. With more complex actions requiring a greater number of states and transitions, it may be difficult to understand however its simple layout allows for, generally, better performance during runtime. Behaviour Trees, due to the reliance on being hierarchical and modular, performance can suffer during runtime when certain environments require multiple complex actions, states and transitions.

Rise of Emergent Behaviour

Emergent Behaviour is the referring to the unexpected outcomes that arise from interactions that don't depend on individual parts but rather on their relationships to one another. In relation to video games this can refer to a game that is simple in gameplay and rules but open-ended in the execution. This can result in situations where the player can control the outcome in unexpected ways.

Two popular examples that show this are "Minecraft" and "The Sims". Minecraft focuses on the use of simple blocks to construct anything they wish. With the addition of redstone, the games electrical system, players can create complex machines and computers within the game. With the simplicity of the game, players can edit and create systems that can force AI behaviour. From what areas they traverse to what items the NPC may be able to trade with the player.

The Sims tackles this concept in a unique and complex way. Being built upon a system of maintaining a character's basic need and desires, the AI within the game can interact with each other to create unexpected results. These can range from a rivalry between the player and an AI NPC, a romantic relationship that the players character automatically haves with other NPCs or even accidental death from a stovetop fire.

Within the video game industry, emergent behaviours between players and AI are typically a complex and, sometimes, accidental. Becoming somewhat of an inhouse struggle as developers must decide if these unexpected outcomes are evident enough to require the game to be updated to remove said outcome or if the outcome is beneficial and can be built upon to give the players who perform said outcome a sense of accomplishment or responsibility.

Technical Difficulties

Due to emergent behaviours naturally unpredictable nature, there are multiple difficulties encountered. These difficulties may be addressed in multiple manners that could lead to advantageous or disadvantageous outcomes when published to the player. I will be lightly exploring the reproducibility and the unpredictability of emergent behaviours using the game "Mount & Blade" as my primary reference.

Mount & Blade, a free-roam action role-playing game, is a game focused on a medieval sandbox world where the player can be engaged in battles, wars and conflicts to raise to power in their own ways. This by nature lead to very unpredictable encounters and situations. Be it the unexpected raid from a wandering band of thieves, getting caught in the crossfire of a war or an unexpected interaction between nobles. As developers try to focus on controlling certain situations, Mount & Blade understood that the chaos of randomness allows the player to build their world in unique and interesting ways only they can explain.

However, with unpredictability comes the issue of being able to reproduce said encounter. In the case of Mount & Blade, each battle will be handled differently, each recruited soldier different proficiencies and each nation viewing the player in different lights. This could be potentially frustrating for players who may have lost in a previous world and wished to recreate the beginnings only to encounter very different situations due to the randomness. This could lead to the necessity of a seeding system, allowing the player to create a world seed where the randomness is no longer random. However, this could be against the developer's vision creating an internal struggle between vision and outcome.

The impact of Technology

Hardware Advancements

Over the course of the video game industry’s history, there has been several technological advancements that have contributed to both the evolution and the accessibility of NPC AI behaviours. The two primary technologies that I wish to talk about are faster processors and the now availability of cloud computing.

With hardware becoming increasingly more powerful, modern processing power allows for significantly more complex AI interactions. This can be seen in the real-time calculations that are required to run for NPCs behaviours. From pathfinding to decision making, modern processing power is at an all-time high and allows for what was once was locked behind technological wall to be achieved on the most common of devices such as mobile phones.

Cloud computing has gained a popularity within modern times. With the rise of AI generated content, it was only time before AI Computing was implemented within the gaming industry. With the use of cloud computing becoming more available to the public from 2006, this allows developers to offload the complex computations and high-end hardware requirement from the players platform to the cloud. Allowing players to enjoy the game regardless of their hardware limitations.

Today, cloud computing systems can be seen primarily in the multiplayer scene within gaming. Allowing for seamless interactions between players and the server. Allowing for players to simultaneously interact with the world and NPCs within it and creating a unique gaming experience for each player who may play the game alone or with friends.

Rise of Machine Learning

Being a relatively new concept within modern video games, there has been a recent popularity in the implementation of machine learning. This has been met with both praise and backlash as the technology is still developing and has resulted in undesirable results for the players of said games.

Machine learning can be seen in games such as “Hello Neighbour 2” and “Suck up!”. Hello Neighbour taking advantage of player data to evolve its basic AI pathfinding and logic to create a more challenging game. While a good concept was poorly executed, being met with the player base arguing the AI was not developed enough to warrant the release. Suck Up however uses common AI such as OpenAI to respond to the player when prompted. The player being able to equip multiple outfits, talk to different characters and try to convince them to invite them into their house based of their conversation and outfit. Resulting in unexpected and comical responses from the NPCs has resulted in an enjoyable gameplay experience.

During the rise of machine learning in the general audience space, Unreal Engine 5 (UE5) developed a tech demo to show of the power of an advanced AI system when implemented into NPC behaviours. “The Matrix Awakens: An Unreal Engine 5 Experience” was the demo that revealed said use of AI and machine learning. With the use of AI to populate the city with massive amounts of NPCs that were each driven by an advanced AI that allowed them to react to the player and engage in complex behaviours, they also had the ability to create dynamic conversations with the player.

Examples that were demonstrated included the AI reaction to gunfire or car crashes that the player may have caused, resulting in the AI to flee from the scene. This simultaneously gave the world a sense of life by allowing the players actions to create a ripple effect that directly affected the environment and the NPCs within it. Another example was the real-time conversations. The player could observe and listen into an existing conversation between NPCs and even participate in said conversations. NPCs have been seen to react to the player differently depending on if they are supposedly in a rush to get to work or as they get more agitated and begin to tell the player to leave them alone.

Modern Trends in NPC AI

Procedural Content

Procedural content generation has become more and more popular as the underlying understanding of the logic becomes available. This can create a diverse and dynamic environment that challenges both the player and NPC AI system to traverse, interact and reach the respective goals.

Two primary examples of procedural content I would like to focus on are from the games “No Mans Sky” and “Minecraft”. Both rely on the use of procedurally generated worlds for the players to explore and engage with in different ways. Minecraft focusing on the creativity and advancements by building more material rich structures, levelling up and upgrading your equipment. No Mans Sky focuses on freedom of exploration by allowing the player to explore an entire universe built up of multiple galaxies populated with unique solar systems. Each planet rich in different resources while inhabiting unique flora and fauna.

The reception of Minecraft upon release and up to today has been overwhelmingly positive. Being on of the most purchased games across multiple platforms, it is a well-respected entity within the gaming scene. This said, its world generation being reliant on a seed has allowed players to explore and find unique worlds that others can explore by entering the same seed. This has created a whole community of people who hunt for interesting seeds that people can use to start their world with a pretty site, a slight challenge or a quick start to supplies in attempt to beat the game as quickly as possible.

No Mans Sky was met with a largely negative reaction upon release. This is due to the feeling of emptiness players were met with when playing the game. This came due to a natural challenge when creating a large explorable space such as the universe which is unique models and interactions. Initially the game felt empty, lacking in any interactions beyond gathering materials and lacking in any ability to find other players easily. This led to the game receiving backlash from the gaming community as they felt betrayed, or promises had not been met. However, over the years with large updates, No Mans Sky has gradually built up its reputation once again, quickly becoming a gaming giant when talking about open world exploration due to the sheer size of the universe the game is set in.

Emotional AI

In games such as Telltales “Walking Dead” or the popular “Fallout” series, there has been systems implemented to create dynamic interactions between the player and NPCs. Such as Fallouts “Karma System”, which can determine how an NPC may converse with the player. Another example of relationships between NPC and the player being explored is games such as “Anno 1503” or “Civilization 6” as the player is set to build and govern over a body of land while trying to maintain or take over from opposing AI. Resulting in said NPC, or a collection of NPCs, to respond negatively to the player while an opposing NPC may view the player as an ally due to their actions.

Such systems have led to unique views from multiple game communities. An example of this could be story driven games such as “Detroit: Become Human”. Detroit, focusing on the concept of freedom and consciousness through the eyes of an artificial intelligence android within a human-dominated world. By having such complex emotions mixed in with interactions, players who dive into one path will result in different outcomes with NPCs within the game and may potentially lead to an alternative ending.

These complex stories that have been mixed with emotionally driven NPC relationships have begun to shape the intricacies of interpersonal relationships between the player and NPCs. While the player may recognise that what they are interacting with is nothing more than a 3d model with a script of predetermined responses, players may feel emotionally attached to characters and react with genuine grief upon negative experiences with said character. This has also allowed people to explore environments and situations that could be alien to them and explore how others may react to their characters actions through a different person’s eyes.

Research Conclusion

Key Findings

During the exploration of AI, NPC behaviours and the development of the relationship between the player and NPCs I have come to a few conclusions. In particular, the use of AI generation within gaming, the relationship between NPCs and players and the issues with current implementations.

AI Generation withing Gaming

I have talked about AI generation in multiple forms throughout this paper. Be it content generation, procedural world/interaction generation and the use of user input for machine learning. Throughout this research I believe that AI generation will only become more prevalent within the gaming industry as it becomes more cost effective. While players may not agree with how some developers may use the generated content and petition for changes of said content or even management responsibilities when regarding said content, it has already been seen in popular titles such as “Black Ops 6”.

Relationships Between NPCs and Players

Relationships have become more complex as the use of systems punish or reward players in certain ways depending on the individual interactions within a world. Be it indirectly by the Fallout karma system or more directly by the immediate consequences of interactions within Telltales “Walking Dead” Series. Games can be seen to be used to explore complex topics such as politics or moral dilemmas that could result in unexpected and emotional outcomes for the player.

Current Issues with Implementations

Current implementation of AI has been met with both positive and negative feedback. As mentioned before, large negative feedback can be found regarding generated content such as art and icons that may be used in games for UI, loading screens or box art. However, the opposite can be felt when implemented in a neutral position such as NPC interactions with the player or crowd simulations using AI. Being a still developing field and controversial topic within almost all fields of media, the development and implementation of AI will be heavily monitored by those who are invested in gaming.

Future Directions

After writing this research paper, I believe I can see a few exciting developments within the making as well as some worrisome patterns and issues that may become problematic in the future.

Developments I believe to be exciting, and which should be observed is the use of cloud computing and AI interactions for players. Cloud computing has been experimented with for the last decade and met with mixed results however with more stable systems becoming available and communication speeds increasing, I believe it is only time before cloud computing gaming becomes a profitable market. Aswell, AI interactions has begun to be implemented in indie developed games and have been seen to produce mixed results which may be both entertaining and unique. With the implementation becoming easier as packages and programs become more available, a quick spike in AI development within the gaming industry regarding indie developers is expected to happen with the coming years.

However, I do believe to have observed some worrying patterns that if not met with backlash or appropriate feedback could lead to future regulations to be put in place limiting both development and implementation. This is mainly seen regarding how certain gaming industries have been seen using existing generated content in final products sold to the public. However, a more worrying topic is the use of AI voice acting withing video games. While not powerful enough to convey convincing complex emotions for a long period of time, with development being explored and advanced, I believe its only time before we see this issue expand and be seen as an acceptable replacement for voice actors.

Requirements

In this chapter I will explore and discuss what requirements I deemed necessary to complete the project. This includes both functional and nonfunctional requirements, particularly the player/script functions that are used most often and have high impact on the player experience.

Functional Requirements

This refers to functions such as movement, interactions, machines, etc. This is the were most of the project is focused but not limited to. The requirements that were deemed necessary were:

1. **Physics based movement.** This refers to the primary method of how the player will move. Not using transformative movement but instead movement that is based on physical forces.

2. **Interactions.** This lets the players actions have impact on the gameplay, whether its small like pushing a button to revealing a whole new section of the level.

3. **Controllable Machines.** In Scarabs Core there are multiple machines that were developed that the player can control to progress throughout the level. These will be explored in deep detail, including the problems and solutions that were developed.

4. **Danger.** Danger refers to elements of the game that the player must avoid, in this case being spikes. Adding slight risk to the gameplay helps encourage the player to master the controls.

5. **AI Enemies.** Using the built-in pathfinding algorithm, NavMesh Agents that use the A* algorithm, to create some basic enemies and scripts showing the knowledge learned from my research into AI behaviours.

Non-Functional Requirements

This will show the important systems that I believed would have a big impact on the workflow of the project as well as the gameplay if not well thought out. These, while mostly in the background of the gameplay experience, include the following:

1. **Scalability.** The scripts should complement one another while being efficient to allow easy interactions between the game and the player.

2. **Performance.** The application should be performance friendly. Playable on most modern devices but still graphically impressive. Graphics were done using the URP (Universal Render Pipeline) that is included with Unity and can be imported.

3. **Usability.** The game should be accessible for those who wish to play. With support for both Keyboard and Controller, and a ramp in gameplay difficulty that will aid the player in learning new concepts as they progress through the levels.

As mentioned previously, most of the requirements are heavily functional based but most would not be able to run as successfully as they have without the aid from the non-functional side. Throughout the next few sections, I will show the scripts and solutions to accomplish the requirements needed to create the application.

Design

Scrabs Core was developed using the Unity game engine, programmed in C# via Visual Studio Code (VS Code), UI designed in Figma, and all models, animations and levels were modelled and created via Blender and Adobe 3D Substance Painter. I will go over each of the technologies, design steps, final looks and my reflection on each.

Technologies

Unity: VS Code and C#

Unity works via nonbehavioral scripts using the programming language C#. This is a language that I had to learn outside of college to create my application. While there are some compatibility issues initially, VS Code was my primary code editor throughout my college career, and I decided to continue using it as I have most experience with it.

All the scripts were written using the C# and aid from Unity API documentation for built in methods and parameters. These allow me to import necessary functions to use certain components that are attached to game objects. A good example of this is the NavMesh import as without it I could not calculate paths, set desired destination or access baked NavMesh data.

Assets and Textures: Blender and Substance Painter

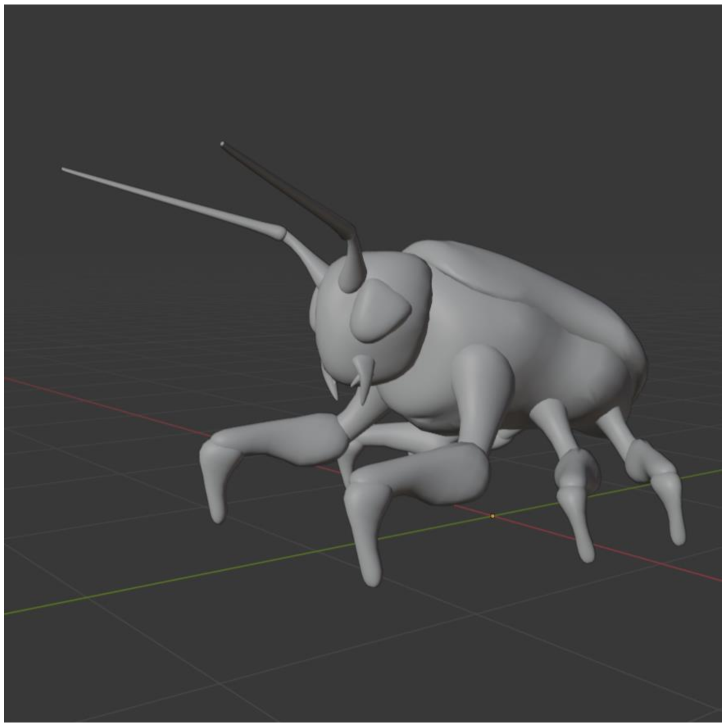
While developing this project, I decided to use my previous experience in 3D Modeling and my understanding of how game ready assets are created to model my own assets. For this Blender was used. Blender is a free open-source multifunctional studio tool. My main use for Blender is 3D sculpting, retopology and animation. For texturing my assets, I would use Substance Painter for the insects and machines. However, other textures such as environmental were sourced from Joao Paulo who offers free high-quality textures.

In later sections I will briefly go over the design process of how the assets were made and the techniques used. I will also go over some of the animations that were made but those will be further expanded on in later sections that handle animations within Unity rather than Blender.

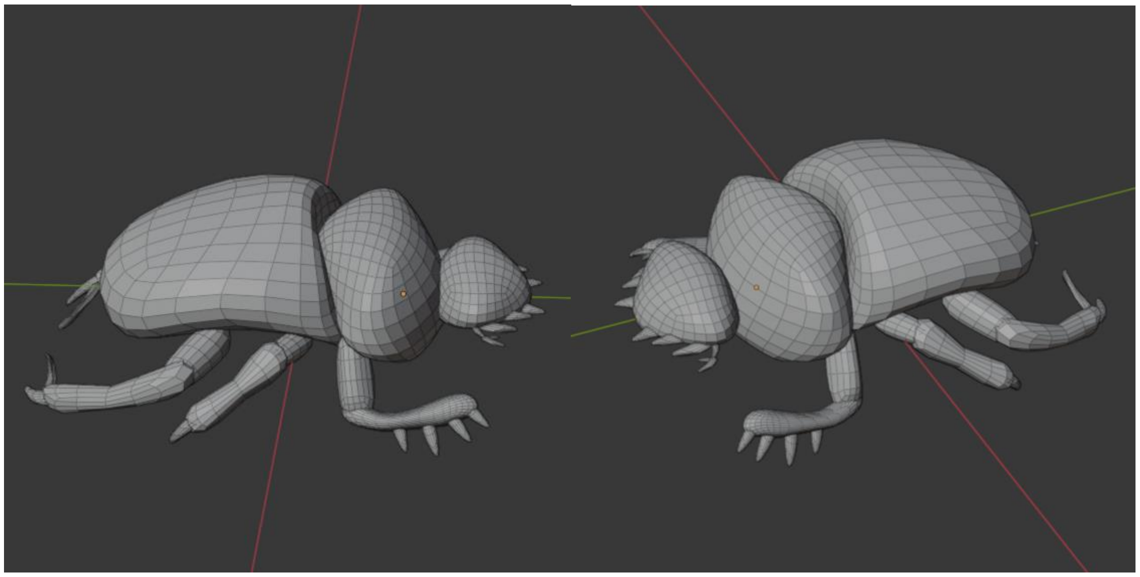
Assets and Models

In this section I will go over 3 models that were created, rigged and animated for the purpose of this project. I will explain the first in detail the process, inspiration and impact these models had on the game as they all followed the same steps to become the final product.

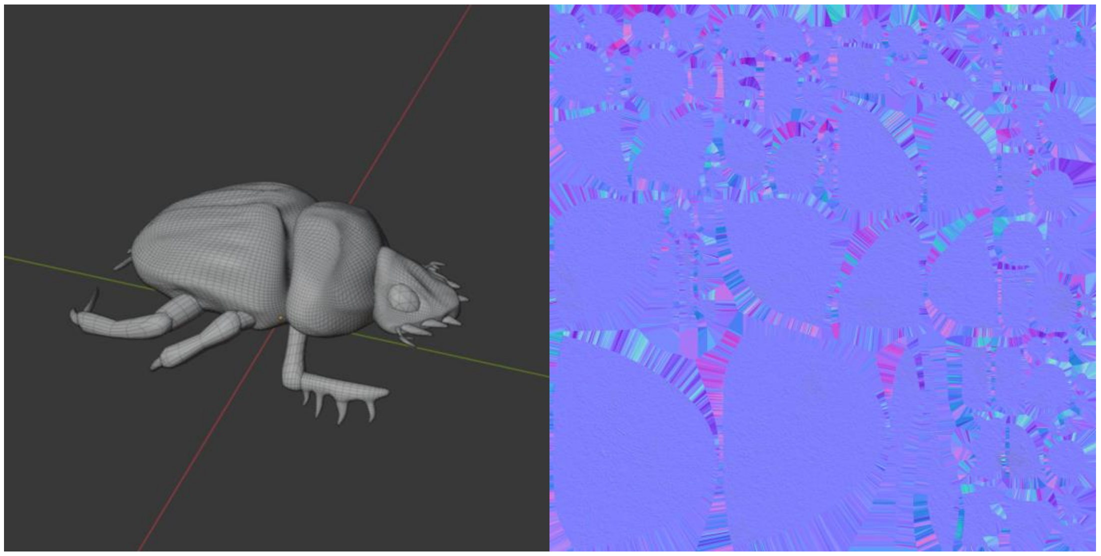
Placeholder Model



Being the insect the player will see primarily, I decided to model this first. Initially creating a simple model as a placeholder before investing time into creating a better and higher fidelity model. The model as such looked like the image above. A simple, rigged but not animated insect shaped model. This would allow me to get a sense of scale and how the character would feel during gameplay. However once ready, I started work on an official insect model.

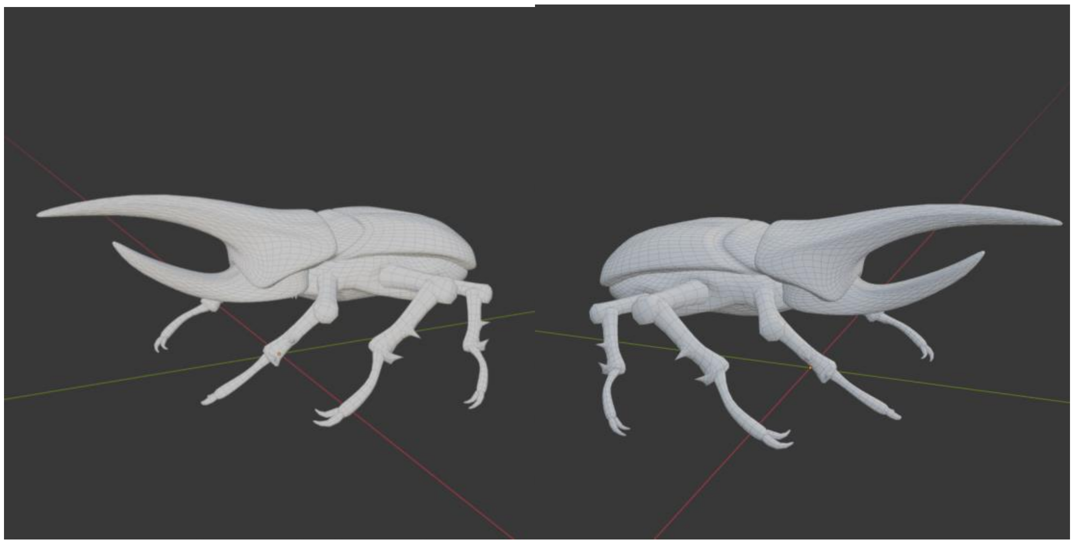


Above is shown the blocked outlook of the model. Made using a simple grab brush within Blender and spheres. Disregarding detail for an accurate shape and silhouette of what would soon become the final model. The next step was refining the shape by subdividing the mesh. This would allow me to create a higher detailed model but would cost more performance to run.

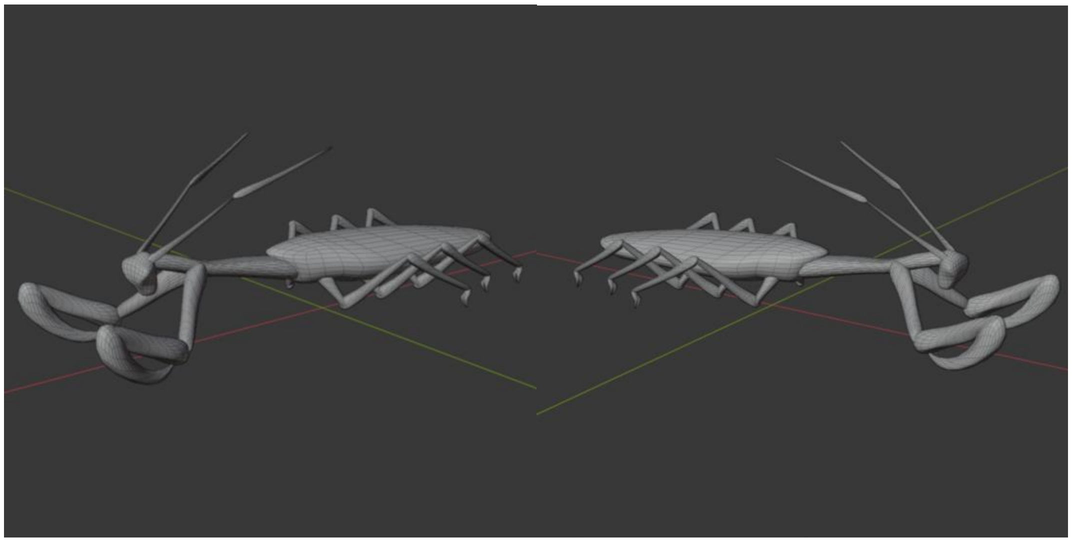


On the left you can see the higher density mesh model. If we were to use this and animate, it would cause huge performance issues. However, we can bake details into images that the GPU can render to give lower mesh dense models the illusion of being higher quality while preserving performance. The image on the right is the baked detail, this case called a normal map, of the model on the left.

Final Models



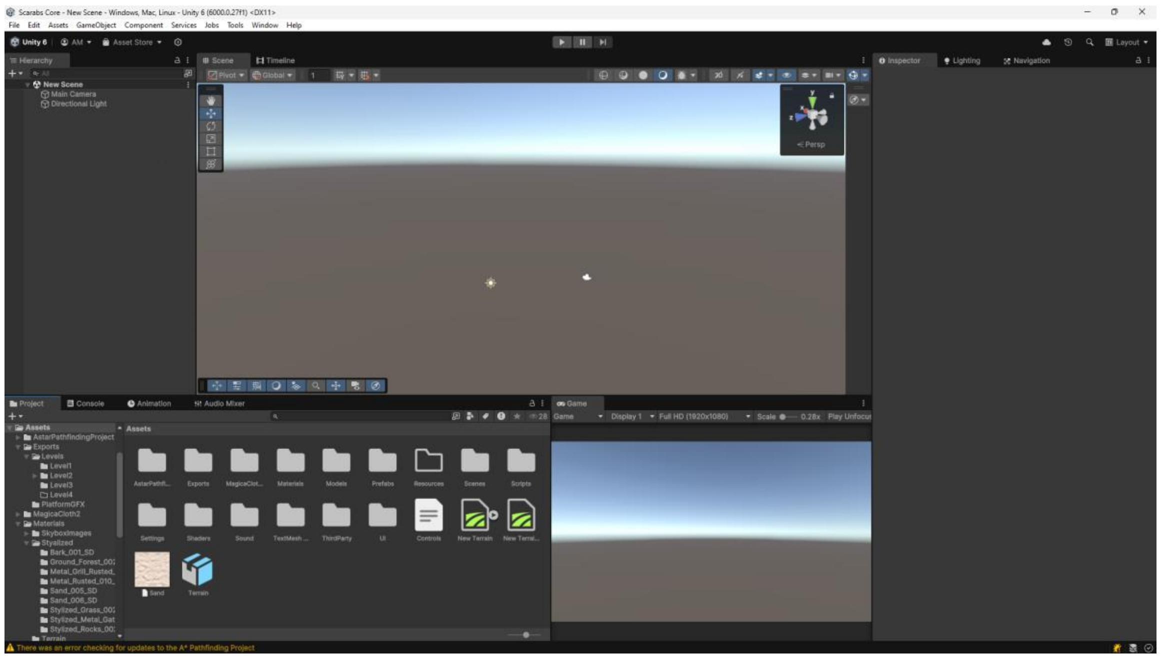
This is the Hercules Beetle. Resonsible for charging at the player and knocking them backwards. This bug will be explored in greater detail when discussing AI and the behaviour system used.



This is the Mantis Machine. A controllable machine within the world that can leap and cut through thick vines to help progress the level. This will be talked in more detail when discussing controllable machines.

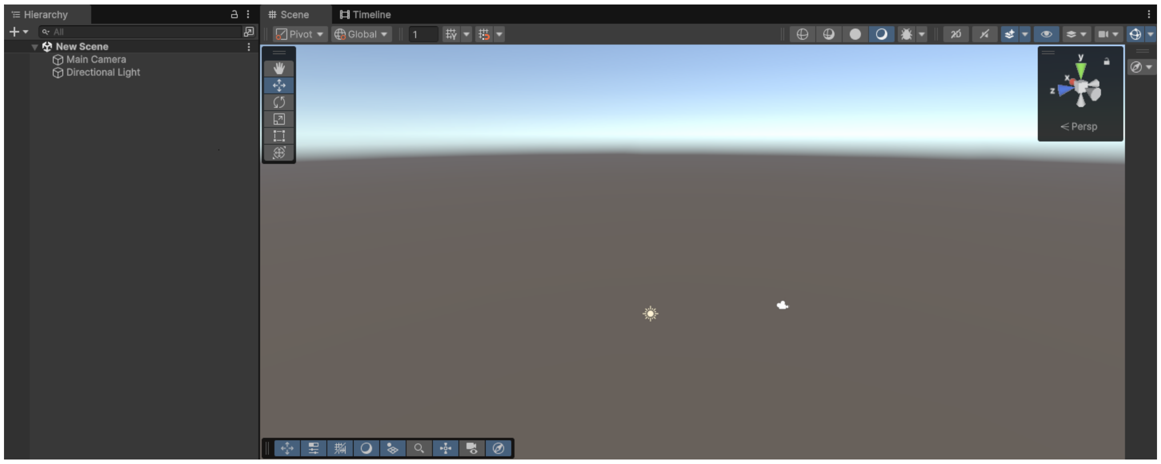
Unity Hierarchy and Inspector

I will briefly go over the GUI that I had to work with during the entirety of this project. Explaining certain parts of this GUI will aid in showing the file system, script management and general technical use of Unity in its most basic form.


Scarabs Core - New Scene - Windows, Mac, Linux - Unity 6 (6000.0.27f1) <DX11>
File Edit Assets GameObject Component Services Jobs Tools Window Help
Hierarchy
New Scene
Main Camera
Directional Light
Scene
Timeline
Inspector
Lighting
Navigation
Project
Console
Animation
Audio Mixer
Game
Assets
There was an error checking for updates to the A* Pathfinding Project

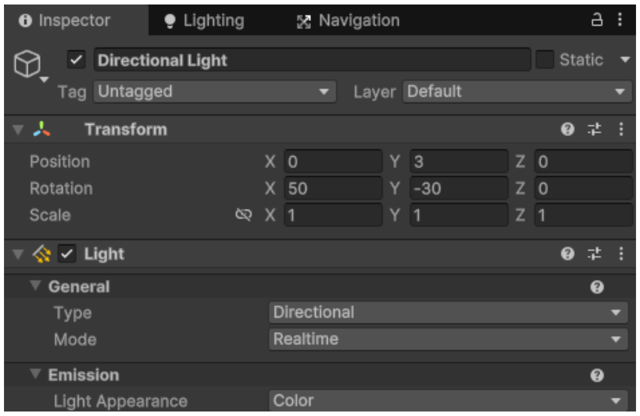
This is the Unity project window. As you can see, without and prior experience, this view can be confusing, intimidating and unwelcoming to people who wish to learn Unity. However, I will go over important sections of the window in detail to aid in my future references to the Unity software itself.

Scene view and Hierarcy



This is the one of the most important areas of the Unity software. The scene view is responsible for allowing me, the developer, to create objects, place models and view the level. To the left of the scene view is the hierarchy, responsible for displaying all the game objects located within the scene. This is also used to organise game objects by assigning them into parent and child relationships. Mastery over these will greatly assist in creating a the completed application.

Inspector



Directional Light Static
Tag Untagged Layer Default

Transform

Position	X	0	Y	3	Z	0
Rotation	X	50	Y	-30	Z	0
Scale	X	1	Y	1	Z	1

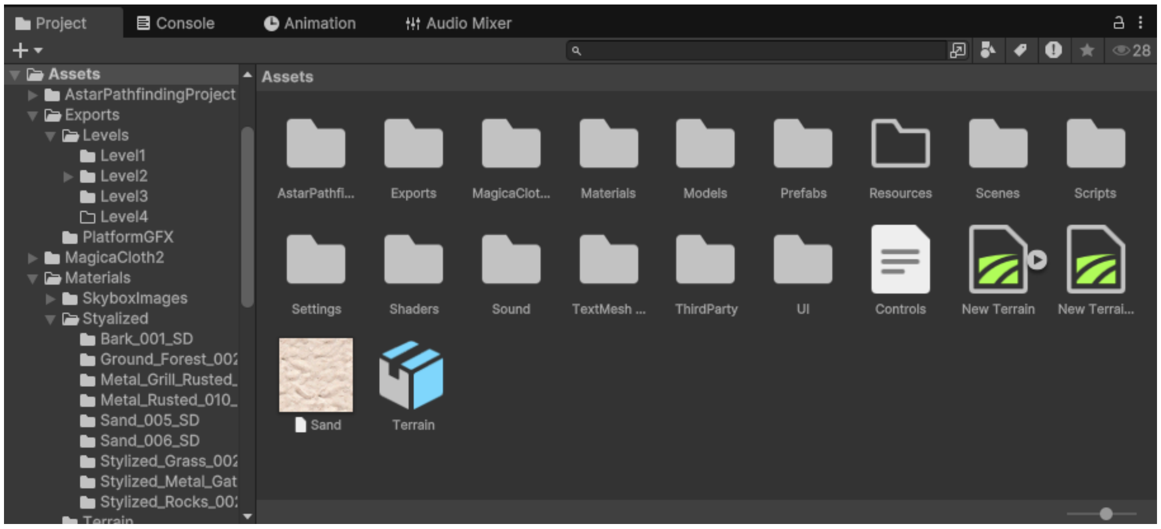
Light

General
Type Directional
Mode Realtime

Emission
Light Appearance Color

The inspector is where developers can see all the components assosiated with each game object and is arguably the most important window. In the image above we are selecting the "Direction Light" object which has 2 components, Transofrm and Light. Transform refers to position, rotation and scale of the object and can be edited in real-time. Light is a component that emits light within the scene. The three main light types are Directional (Sun/Moon), Point (Lamp) and Spot Light (Flashlight). In this case it is a Directional light and acts as the sun to light up the scene.

Project / Asset Window



This is the project window, where we can view all the folders, scripts, files, models and animations that the project includes. This is directly linked to the actual file system within the operating system and can be viewed and edited directly from there if necessary. On the left of the project window, you can see the file hierarchy which updates in real time as you create folders, images, etc.

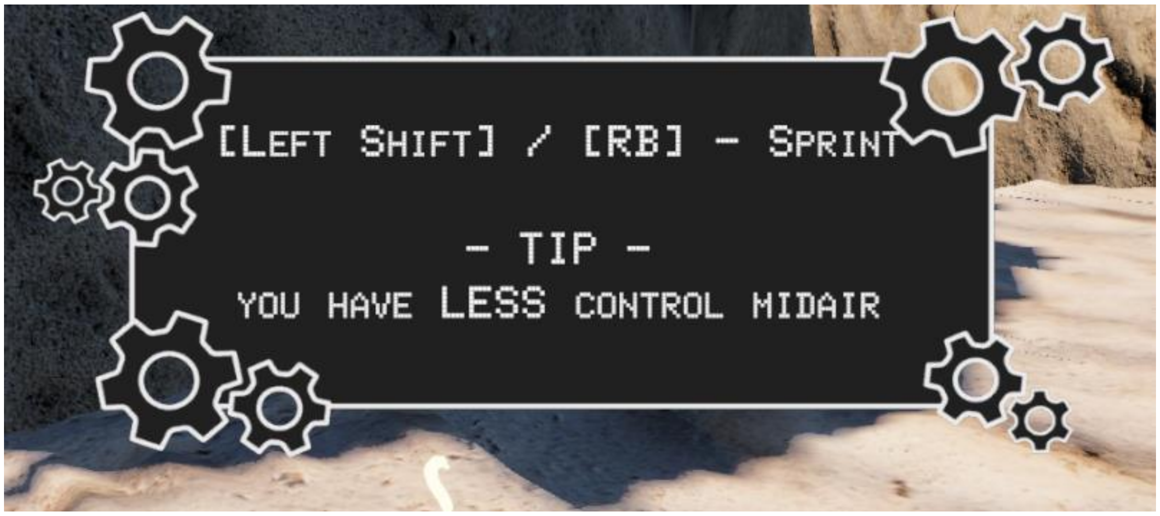
Console

This is the console. While running the game, if there are any errors, warning or debug messages sent from scripts, they will be displayed here for debugging purposes.

UI

Due to the project being focused on more technical sides, the UI is limited but no less important. The UI is a simple, clean and effective in its purpose. Guiding the player through levels by giving hints and useful tips while also purposing as the menu system.

This is the pop-up asset. When giving hint to the player or information, this is what will be displayed. Made in Figma and exported as a PNG, I will go over how I created a simple script that creates this asset with the desired text.



This is an example of how it would look in-game. Showing a control, in this case the sprint button for PC (Left) and Controller (Xbox controls on the right), and also a giving a helpful Tip.

Design Conclusion

While not the focus of the project, it is important to show the design processes that go into creating a game application. While there are a lot more that goes into creating larger projects, I decided to keep it limited to what was said above in order to not overwhelm with information while also help inform and explain the decisions that went into the design process.

Going forward, I will discuss more of functionality, scripts that were created, bug and challenged and the solutions that were developed to overcome said problems. I will use screenshots of scripts and explain key lines in detail with the assumption that the reader has little to know prior experience in programming languages in order to help the readability of the language used.

Implementation

In this chapter I will go over the process of implementing all the key features and requirements in my project. From writing scripts and creating systems that allow for easy expansion on existing concepts if this project is further developed in the future.

Player Functionality

During this section, I will go over the player functionality that was developed, the process that went into creating the scripts and methods used, and issues that were encountered and solved. These will be reinforced with screenshots of code that will be fully explained to demonstrate the knowledge and methodology behind the final product. Player functionality will be broken into multiple sections. These sections include:

1. **Key Movement Functionality**. Exploring the methods used to move the player within the game.

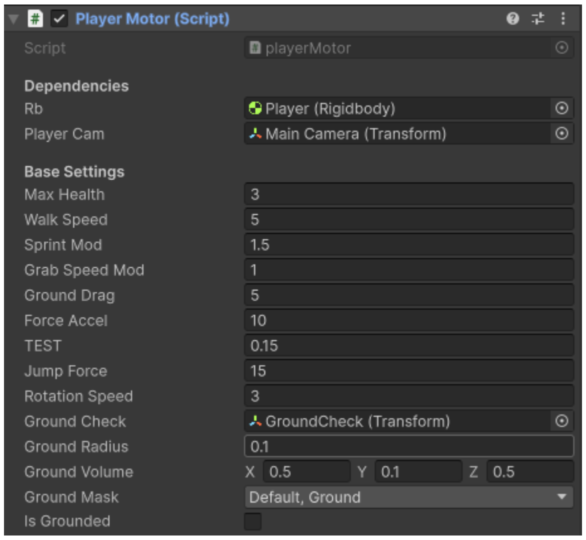
2. **Knockback Implementation**. How the physics system played into creating the movement scripts and associated struggles or problems.

3. **Interactions**. The functions and techniques used to ensure interactions between the player and the game world.

4. **Controllable Machines**. Techniques used to create machines that were controlled via the player inputs.

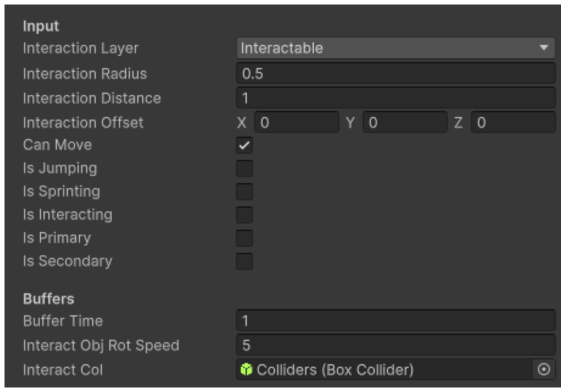
Key Movement Functionality

To understand this section, we will need to explore the components used by the player, assigned variables, colliders and how they were used within the player movement script. Please note that each reference to code will be marked in **Bold** in order to distinguish between normal text and code text.



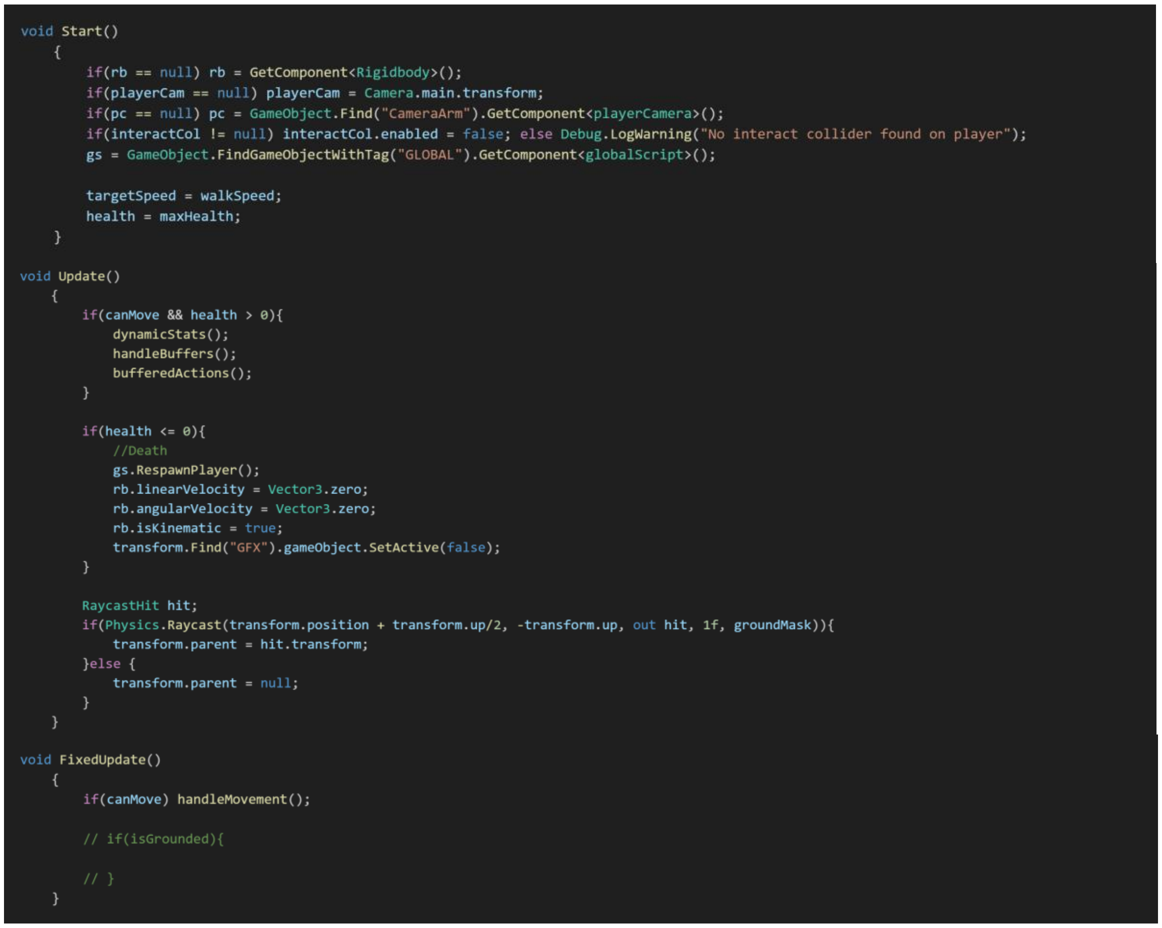
Player Motor (Script)

Script	playerMotor
Dependencies	
Rb	Player (Rigidbody)
Player Cam	Main Camera (Transform)
Base Settings	
Max Health	3
Walk Speed	5
Sprint Mod	1.5
Grab Speed Mod	1
Ground Drag	5
Force Accel	10
TEST	0.15
Jump Force	15
Rotation Speed	3
Ground Check	GroundCheck (Transform)
Ground Radius	0.1
Ground Volume	X 0.5 Y 0.1 Z 0.5
Ground Mask	Default, Ground
Is Grounded	☐



Input	
Interaction Layer	Interactable
Interaction Radius	0.5
Interaction Distance	1
Interaction Offset	X 0 Y 0 Z 0
Can Move	✓
Is Jumping	☐
Is Sprinting	☐
Is Interacting	☐
Is Primary	☐
Is Secondary	☐
Buffers	
Buffer Time	1
Interact Obj Rot Speed	5
Interact Col	Colliders (Box Collider)

Above is the "Player Motor" script which is attached to the player. During this section I will explain in great detail how each this is used to control the players movement using physical forces. From this point onwards, the way I will explain code will be full screenshots of the code, this may take up to one or two full pages, followed by the explination of key blocks and terms.



```
void Start()
    {
        if(rb == null) rb = GetComponent<Rigidbody>();
        if(playerCam == null) playerCam = Camera.main.transform;
        if(pc == null) pc = GameObject.Find("CameraArm").GetComponent<playerCamera>();
        if(interactCol != null) interactCol.enabled = false; else Debug.LogWarning("No interact collider found on player");
        gs = GameObject.FindGameObjectWithTag("GLOBAL").GetComponent<globalScript>();

        targetSpeed = walkSpeed;
        health = maxHealth;
    }

void Update()
    {
        if(canMove && health > 0){
            dynamicStats();
            handleBuffers();
            bufferedActions();
        }

        if(health <= 0){
            //Death
            gs.RespawnPlayer();
            rb.linearVelocity = Vector3.zero;
            rb.angularVelocity = Vector3.zero;
            rb.isKinematic = true;
            transform.Find("GFX").gameObject.SetActive(false);
        }

        RaycastHit hit;
        if(Physics.Raycast(transform.position + transform.up/2, -transform.up, out hit, 1f, groundMask)){
            transform.parent = hit.transform;
        }else {
            transform.parent = null;
        }
    }

void FixedUpdate()
    {
        if(canMove) handleMovement();

        // if(isGrounded){

        // }
    }
```

Before explaining the code shown above, let me explain certain key terms you will see multiple times throughout this and sections to come.

- **Rigidbody**: A way of controlling an objects position through physics simulation.
- **Raycast:** An intersecting ray with the plane.
- **RaycastHit:** A structure used to get information back from the raycast.
- **Transform:** Referring to the tranform component located on the object to get information and apply positional and rotational data if necessary.
- **LayerMask:** A list of layers that can be used as a filter.
- **Vector3:** A 3 dimensional representation of data (X,Y,Z).

These are the functions that untimatly run this script. **Start()** function runs when the application is started, **Update()** runs every frame while **FixedUpdate()** runs at a constant time interval. These two update functions are important and **Update()** is good for handling input and logic while **FixedUpdate()** handles physics.

Within the **Start()** function I am assigning variables to the relivant components. For example, “**if(rb == null) rb = GetCompoent<Rigidbody>();**” get the Rigidbody component from the object this script is attached to, in this case the player, and assignes it to a variable **rb**, which is defined at the top of the script outside of these screenshots.

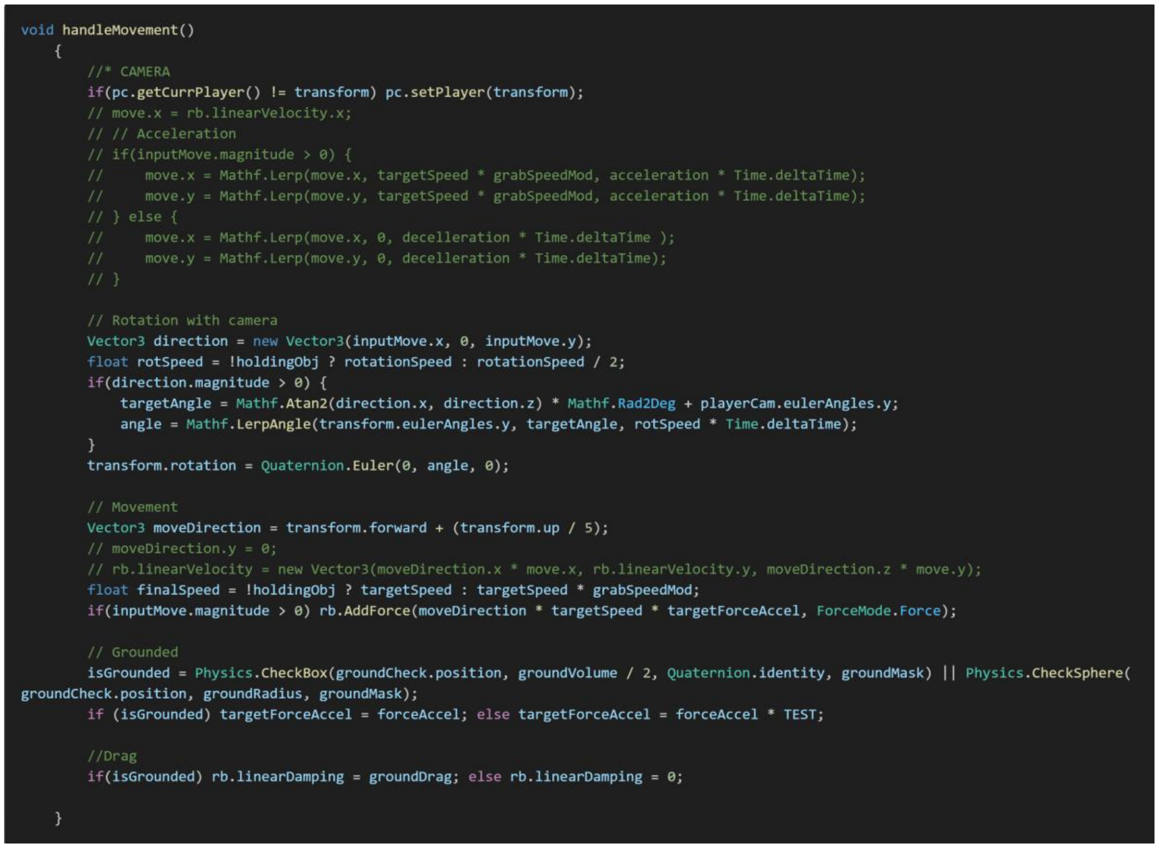
Within the **Update()** function you can see logic being handled. This includes what to do if the player dies and how to handle dynamic moving platforms by assigning the player as a child to the object below. I would like to go into detail about how this is performed.

First we create a **RaycastHit** to store the vairables, in this case called “**hit”**. We then define 5 parameters to assign to **Physics.Raycast()**. Its origin, direction, out (where the information is stored), length and layermask. In this case:

- **Origin** is “**tranform.position + transform.up/2**”. Transform up is refering to a Vector3 that equals (0, 1, 0). Therefore, the origin is half a unit up from the current position of the player.
- **Direction** is “**-tranform.up**”. This equals a Vector3 (0, -1, 0), or a downwards direction.
- **Out** is the previously defined variable of **RaycastHit hit**.
- **Length** is 1f. The F is how we degine numbers as either integers or float point numbers.
- **LayerMask** is equal to a varriable called “**groundMask**” which is assigned the layers of which the player object considers ground.

Physics.Raycast() returns as a boolean, true or false statement, that we can use to create conditional statements. In this case if returned true we set the players parent, seen as **tranform.parent**, to the object that the ray collided with, seen as **hit.transform**. Otherwise, the player is considered an independent game object and we set is parent to **null**.

With this, we can ensure that if the player stands on a moving platform, its position will be directly linked to the platform. If this was not performed, any moving object that the player is standing on would not influence the players position, leaving the player in a position disconnected from the platform.



```
void handleMovement()
    {
        //* CAMERA
        if(pc.getCurrPlayer() != transform) pc.setPlayer(transform);
        // move.x = rb.linearVelocity.x;
        // // Acceleration
        // if(inputMove.magnitude > 0) {
        //     move.x = Mathf.Lerp(move.x, targetSpeed * grabSpeedMod, acceleration * Time.deltaTime);
        //     move.y = Mathf.Lerp(move.y, targetSpeed * grabSpeedMod, acceleration * Time.deltaTime);
        // } else {
        //     move.x = Mathf.Lerp(move.x, 0, decelleration * Time.deltaTime );
        //     move.y = Mathf.Lerp(move.y, 0, decelleration * Time.deltaTime);
        // }

        // Rotation with camera
        Vector3 direction = new Vector3(inputMove.x, 0, inputMove.y);
        float rotSpeed = !holdingObj ? rotationSpeed : rotationSpeed / 2;
        if(direction.magnitude > 0) {
            targetAngle = Mathf.Atan2(direction.x, direction.z) * Mathf.Rad2Deg + playerCam.eulerAngles.y;
            angle = Mathf.LerpAngle(transform.eulerAngles.y, targetAngle, rotSpeed * Time.deltaTime);
        }
        transform.rotation = Quaternion.Euler(0, angle, 0);

        // Movement
        Vector3 moveDirection = transform.forward + (transform.up / 5);
        // moveDirection.y = 0;
        // rb.linearVelocity = new Vector3(moveDirection.x * move.x, rb.linearVelocity.y, moveDirection.z * move.y);
        float finalSpeed = !holdingObj ? targetSpeed : targetSpeed * grabSpeedMod;
        if(inputMove.magnitude > 0) rb.AddForce(moveDirection * targetSpeed * targetForceAccel, ForceMode.Force);

        // Grounded
        isGrounded = Physics.CheckBox(groundCheck.position, groundVolume / 2, Quaternion.identity, groundMask) || Physics.CheckSphere(
groundCheck.position, groundRadius, groundMask);
        if (isGrounded) targetForceAccel = forceAccel; else targetForceAccel = forceAccel * TEST;

        //Drag
        if(isGrounded) rb.linearDamping = groundDrag; else rb.linearDamping = 0;

    }
```

Once again, allow me to explain certain key terms that will be necessary to explore this section of code which is responsible for handling the movement logic. These terms include:

- **Mathf:** A collection of common math functions.
- **Quaternion:** Representation of rotations using a 4D vector (x,y,z,w).
- **Euler:** Angles represented in degrees of rotation.
- **Vector2:** A 2D vector (x,y).
- **Lerp:** A term that refers to the process of linearly interpolating between two values.
- **CheckSphere:** A physics methods that returns true if collider is detected in the given radius.
- **CheckBox:** A physics method that returns true if collider is detected in the given box dimensions.
- **ForceMode:** To specify how to apply a force.

Above you can see the function **handleMovement()** which is continuously called from within the **FixedUpdate()** function show previously. This function is responsible for rotation the player model, moving the player using rigid body physics and handling the logic to determine if the player is grounded. I will briefly go over the rotation, grounded and go into a in-dept look at how the movement is handled.

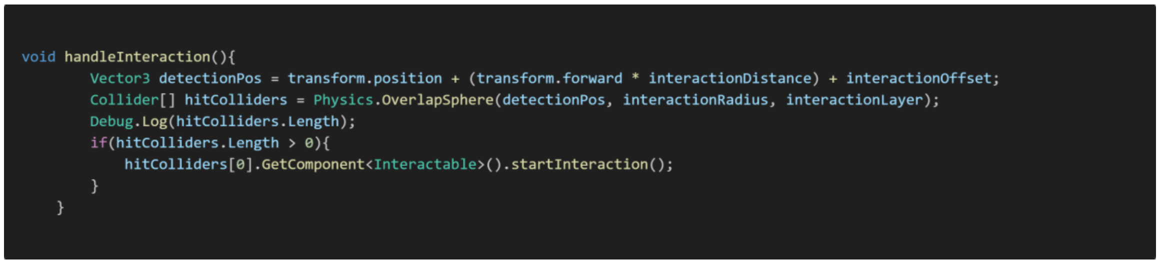
Rotation of the player model is determined by the current movement direction. As you can see, "**Vector3 direction = new Vector3(inputMove.x, 0, inputMove.y);**" is responsible for getting the direction. The variable **inputMove** refers to a Vector2 that handles the X and Y movement of the player. We then calculate the **targetAngle** and **angle**, variable set at the top of the script, by performing trigonometry, adding the current rotation of the main camera to ensure the model rotates relative to its y-rotation and linearly interpolating between the two angles to smoothly turn the player model.

To determine if the player is considered grounded, we perform a simple and popular method of checking. Commonly it is done by using the **CheckSphere()** method but in my case I decided to use **CheckBox()** alongside as I found some detection problems with only **CheckShphere()**. By having the Boolean **isGrounded** determined by **"Physics.CheckShpere(…) || Physics.CheckBox(…)"** (|| => conditional operator referring to **OR**) we can use that to apply certain logic when the player is grounded.

When we want to move the player, we use rigid body physics to apply forces in the desired directions. In this case, since we rotate the player model based on inputs and relative to the camera, we only want to move the player in the direction they are facing when a movement input is being held. To do this we must define the direction which in this case is **transform.forward + (transform.up / 5)**, we add an upwards direction to aid in movement when encountering slopes.

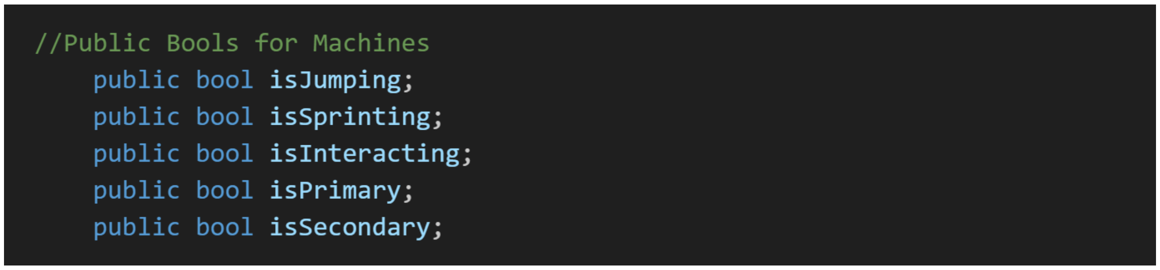
We then determine the speed the player moves. For this we have the line "**float finalSpeed = !holdingObj ? targetSpeed : targetSpeed * grabSpeedMod**". This is known as a ternary operator. Instead of using if statements we can use this to define a variable depending on the result of a conditional statement. In this case, if **holdingObj** is true, the player moves slower by multiplying a **grabSpeedMod** variable into the **targetSpeed**, otherwise, the player moves at **targetSpeed** without any modifications.

With direction and speed handles, we can now apply the force to the player. This is handled by "**if(inputMove.magnitude > 0) rb.AddForce(…)**". If the player is holding a movement direction, for example left, the **inputMove.magnitude** would equal 1, running the next code **rb.AddForce(…)**. The method **AddForce()** requires two parameters. A Vector3 and a **ForceMode**. In this case we can multiply all of our calculations to get the Vector3 required and set the **ForceMode** to "**Force**". This applies a continuous force to the rigid body relative to the rigid body mass.



```
void handleInteraction(){
        Vector3 detectionPos = transform.position + (transform.forward * interactionDistance) + interactionOffset;
        Collider[] hitColliders = Physics.OverlapSphere(detectionPos, interactionRadius, interactionLayer);
        Debug.Log(hitColliders.Length);
        if(hitColliders.Length > 0){
            hitColliders[0].GetComponent<Interactable>().startInteraction();
        }
    }
```

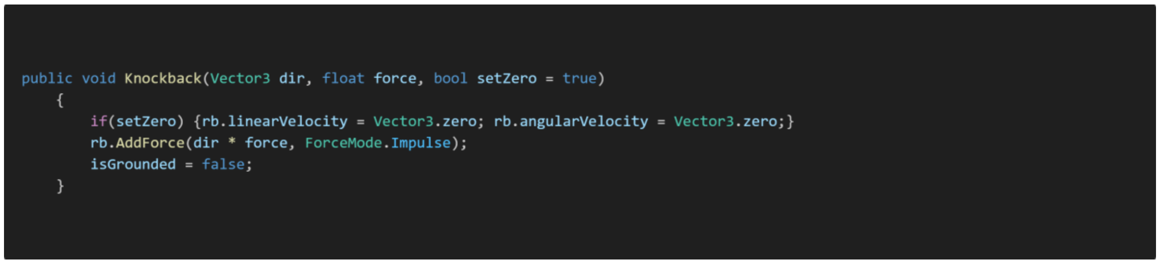
Above is the **handleInteraction()** function, responsible for allowing the player to interact. It is important to explain this here before going in-dept on how interactions were written. Here we are defining a **detectionPos** and **hitColliders**. We use to **detectionPos** with a method **OverlapSphere(…)** to collect all the colliders that overlap in each radius at the **detectionPos**. We can then get the first collider in that array of colliders, determine if that object has an the **Interactable** script and then perform the interaction between the object and the player.



```
//Public Bools for Machines
    public bool isJumping;
    public bool isSprinting;
    public bool isInteracting;
    public bool isPrimary;
    public bool isSecondary;
```

Above is a list of bools. While not relevant right now, it will be important to keep in mind in the future when discussing machines and how the player inputs were translated from the player script to machine scripts. The main point to remember is that this is a list of bools that keep track of input states. With this in mind, I will discuss the next section.

Knockback Implementation



```
public void Knockback(Vector3 dir, float force, bool setZero = true)
    {
        if(setZero) {rb.linearVelocity = Vector3.zero; rb.angularVelocity = Vector3.zero;}
        rb.AddForce(dir * force, ForceMode.Impulse);
        isGrounded = false;
    }
```

Above is a knockback function from the playerMotor.cs script. This is the function responsible for applying knockback forces from other objects within the scene. This function is accessible from other scripts and plays a key part in the physics-based gameplay of Scarabs Core. Therefore, I believe it important to break this down and explain the reasoning behind it.

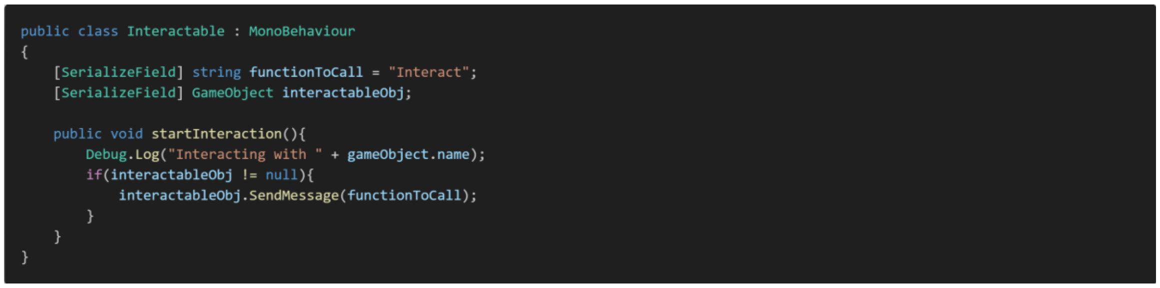
Knockback(…) has two required parameters with one optional. These are:

- **Vector3 dir:** A Vector3 representation of the desired direction to knockback the player.
- **Float force:** The amount of force to apply to the player.
- **Bool setZero = true:** This allows us to decide if we should or should not apply forces relative to the current velocity of the player.

If you look within the function you will notice that we are again using **AddForce** but this time with a different **ForceMode**. This time we are using **Impulse**, which instead of applying a continuous force relative to the objects mass, it adds an instant force relative tit he objects mass. While they may seem similar it is important to note the distinction between “continuous” and “instant”. **Force** applies the force over the duration of time in each **FixedUpdate()** while **Impulse** applies the force instantly over a single function call.

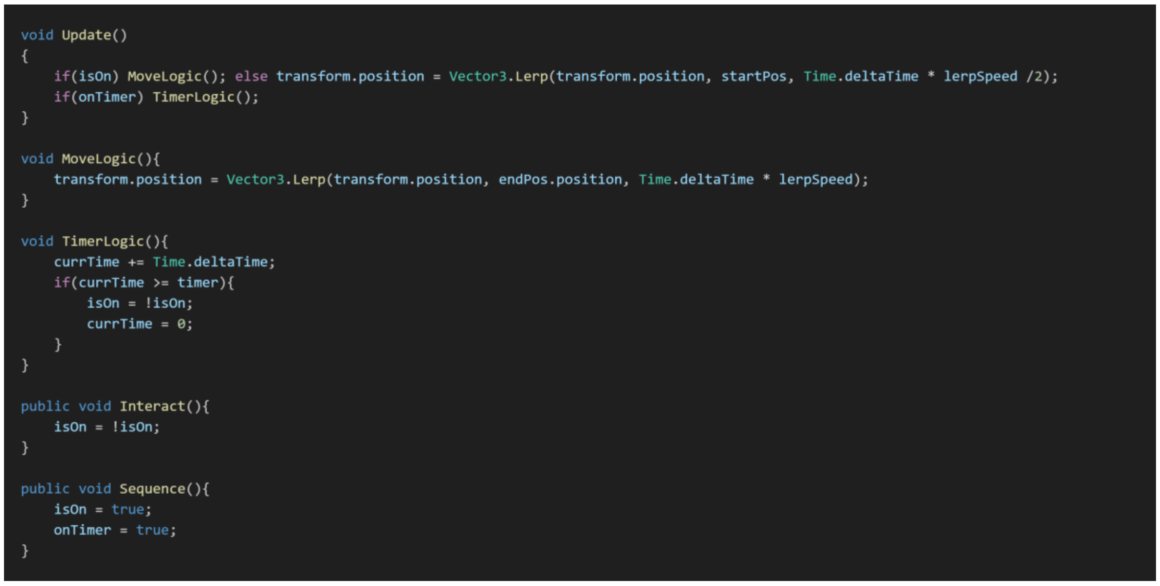
With this function explained, we can move on to the interactions between the player and the world and how they may apply certain forces to the player.

Interactions



```
public class Interactable : MonoBehaviour
{
    [SerializeField] string functionToCall = "Interact";
    [SerializeField] GameObject interactableObj;

    public void startInteraction(){
        Debug.Log("Interacting with " + gameObject.name);
        if(interactableObj != null){
            interactableObj.SendMessage(functionToCall);
        }
    }
}
```

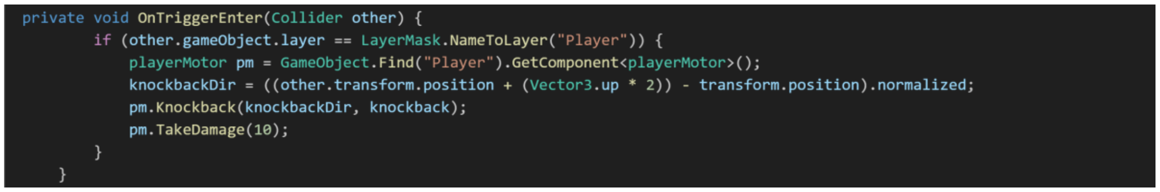
Above is the interactable script, responsible for handling interactions between the player and game objects. As you can see it is short but powerful as this can handle simple interactions from opening doors to complex interactions such as handling machines. The most important line is "**interactableObj.SendMessage(fucntionToCall);**". The method **SendMessage(…)** lets me call a function inside another object. This is different than calling a public function from another script as the **SendMessage** goes through each script attached to the object and, if they have the correct function, calls the specified function on each script allowing for multiple interactions to occur from a single line of code.



```
void Update()
{
    if(isOn) MoveLogic(); else transform.position = Vector3.Lerp(transform.position, startPos, Time.deltaTime * lerpSpeed /2);
    if(onTimer) TimerLogic();
}

void MoveLogic(){
    transform.position = Vector3.Lerp(transform.position, endPos.position, Time.deltaTime * lerpSpeed);
}

void TimerLogic(){
    currTime += Time.deltaTime;
    if(currTime >= timer){
        isOn = !isOn;
        currTime = 0;
    }
}

public void Interact(){
    isOn = !isOn;
}

public void Sequence(){
    isOn = true;
    onTimer = true;
}
```

The above code is taken from the **lethalSpikes.cs** script. Responsible for interacting with the player by damaging and applying knockback. Here you can see multiple functions which determine the behaviour of the spikes. These being:

- **MoveLogic:** Responsible for moving the spikes to their ON position.
- **TimerLogic:** Responsible for enabling and disabling the spikes based on a timer.
- **Interact:** Called by a separate script to enable or disable the spikes
- **Sequence:** Called by a sequence script to enable in a specific order.



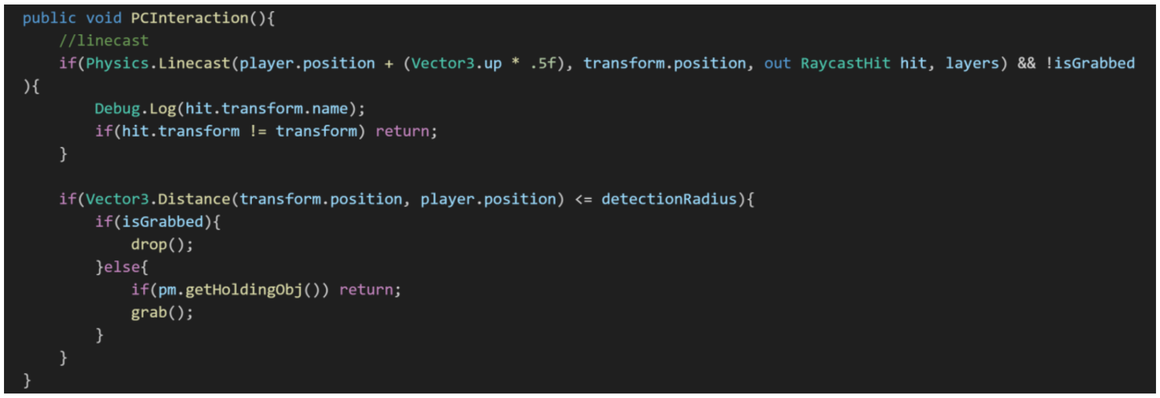
```
private void OnTriggerEnter(Collider other) {
        if (other.gameObject.layer == LayerMask.NameToLayer("Player")) {
            playerMotor pm = GameObject.Find("Player").GetComponent<playerMotor>();
            knockbackDir = ((other.transform.position + (Vector3.up * 2)) - transform.position).normalized;
            pm.Knockback(knockbackDir, knockback);
            pm.TakeDamage(10);
        }
    }
```

Before exploring the logic within this block, it is important to define certain terms. Those being:

- **OnTriggerEnter:** This runs when the trigger area attached to the object detects another collider.
- **NameToLayer:** A method to convert a string representation of the a layer into the accurate layer datatype.
- **GameObject.Find:** A method to find an object within the scene given the name as a string.
- **Normalized:** This scales the values of the given vector so the magnitude is equal to 1.

This block of code is responsible for applying the damage and knockback to the player. Using the **OnTriggerEnter** function that is built into Unity, we can detect if a collider enters the trigger zone. Then by checking if "**other.gameObject.layer == LayerMask.NameToLayer("Player")**" we can ensure the object that was detected was the player. We can then create a reference to the players **playerMotor** script by finding the player object and using the **GetComponent<>()** method to pull its **playerMotor** script.

With that we calculate the normalized direction by getting the position of **other**, which is the player in this case, and subtracting the position of the spikes. This is based of the geometry method of calculating the resultant vector between 2 points. With the direction, we can call the **knockback** method within **playerMotor** to apply a direction and force. This is the primary method of getting direction of force used throughout this project and will be seen in later examples.

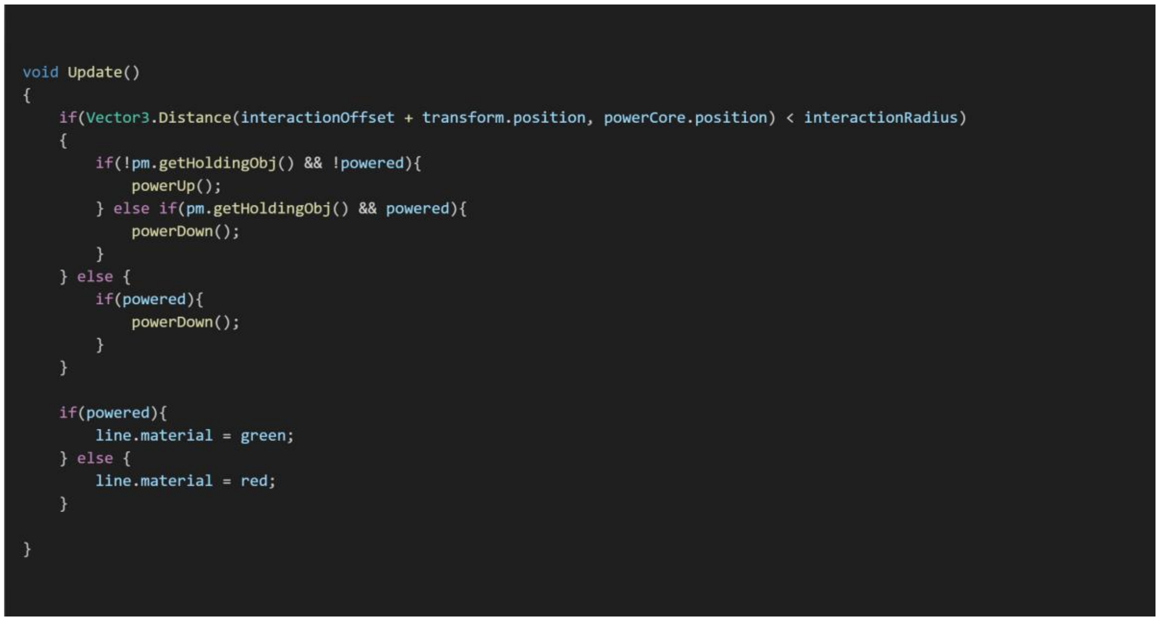


```
public void PCInteraction(){
    //linecast
    if(Physics.Linecast(player.position + (Vector3.up * .5f), transform.position, out RaycastHit hit, layers) && !isGrabbed
){
        Debug.Log(hit.transform.name);
        if(hit.transform != transform) return;
    }

    if(Vector3.Distance(transform.position, player.position) <= detectionRadius){
        if(isGrabbed){
            drop();
        }else{
            if(pm.getHoldingObj()) return;
            grab();
        }
    }
}
```

The above block was taken from the **powerCore.cs** script. The power core is a key part of the game that is responsible for powering up certain areas to allow progression. We will go over the **powerCore** script before moving onto the **powerSocket** script, which is responsible for handling the interactions between the power core and interactable objects. First, lets define new methods:

- **Linecast:** A line drawn from start to end that returns true if a collider on a given layer blocks the line.
- **Distance:** A built in Vector method to calculate the distance between two Vectors.

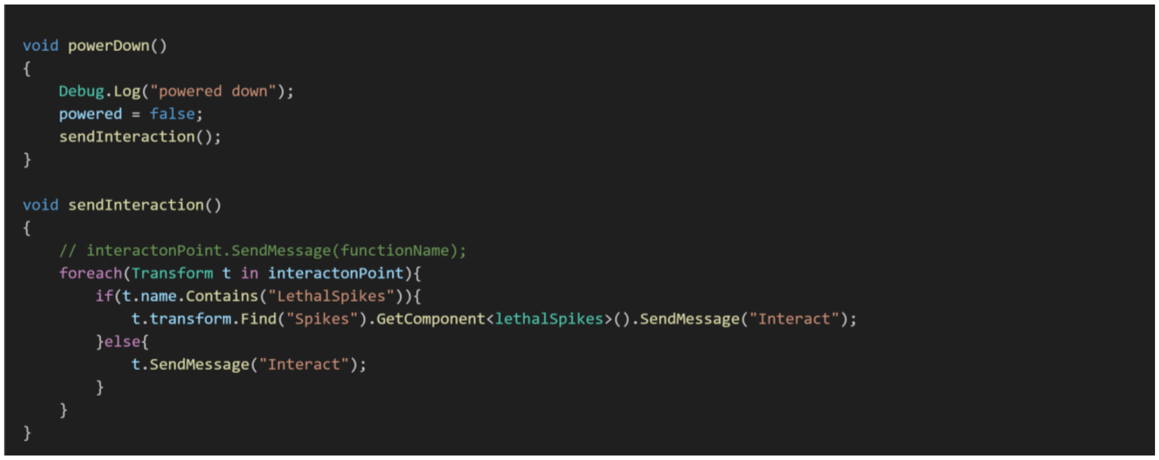
In the function above **PCInteraction()** we are handling whether the player can pick up or drop the power core. First by running a **Linecast** to check if the line of sight (LOS) is broken between the power core and player. If the LOS is not broken, it runs a second check to ensure that the distance between the power core and player isn't too far to logically be grabbed. Then it is simply checking if the power core **isGrabbed**. If true, the core is dropped, else, it is grabbed.



```
void Update()
{
    if(Vector3.Distance(interactionOffset + transform.position, powerCore.position) < interactionRadius)
    {
        if(!pm.getHoldingObj() && !powered){
            powerUp();
        } else if(pm.getHoldingObj() && powered){
            powerDown();
        }
    } else {
        if(powered){
            powerDown();
        }
    }

    if(powered){
        line.material = green;
    } else {
        line.material = red;
    }

}
```

Above is code from the **powerSocket.cs** script. Responsible for allowing interactions between the power core and interactable objects, such as spikes and doors. As we have discussed most terms previously, I will move immediately into explaining the logic.

Within the **Update()** function, which as described previously runs every frame, we are checking if the distance between the power core and the socket is less than a defined **interactionRadius**. If within said radius, we check if the player is **getHoldingObj()**, since the only grabbable object within the game is the power core, and if the socket is already powered. If the power core is within the radius, the player isn't holding onto the power core and the socket isn't currently powered, it becomes powered, otherwise it will power down if the power core is being held or if the distance between the power core and socket is outside of the **interactionRadius.**

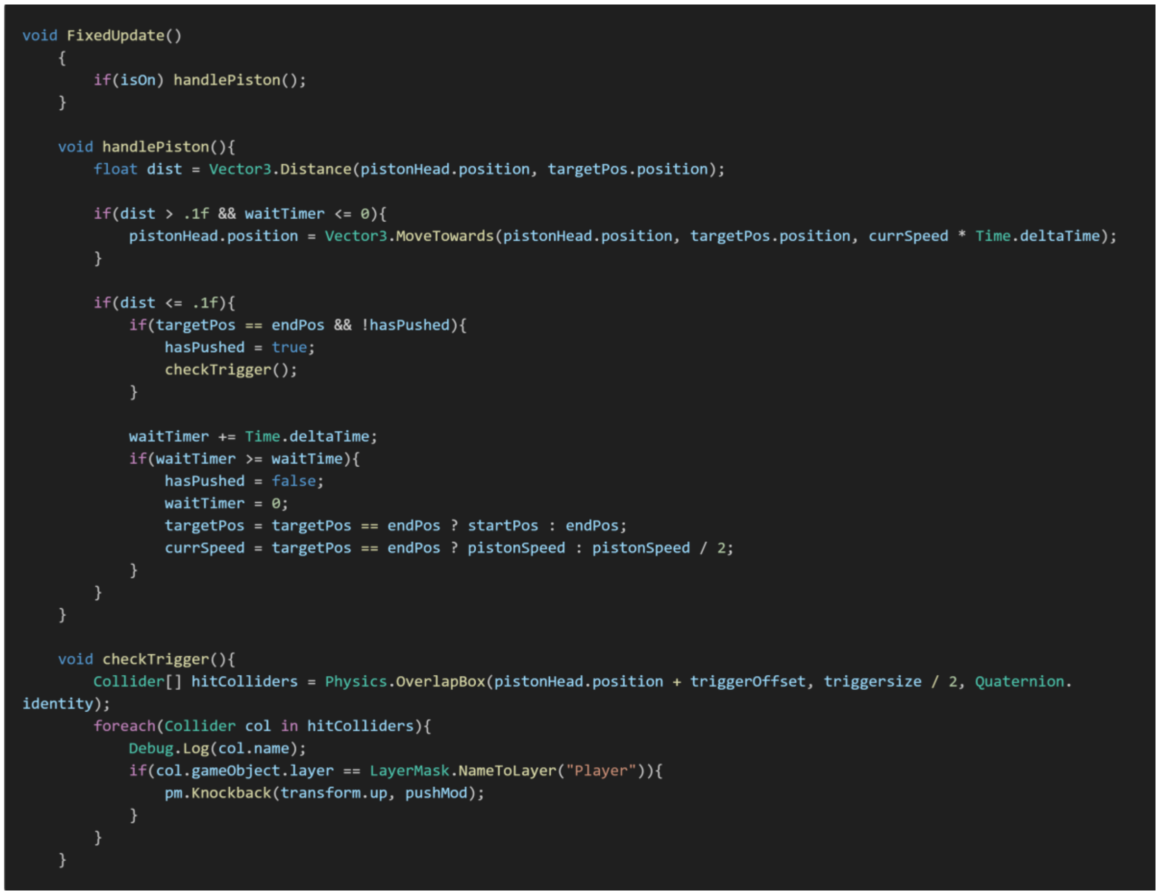


```
void powerDown()
{
    Debug.Log("powered down");
    powered = false;
    sendInteraction();
}

void sendInteraction()
{
    // interactonPoint.SendMessage(functionName);
    foreach(Transform t in interactonPoint){
        if(t.name.Contains("LethalSpikes")){
            t.transform.Find("Spikes").GetComponent<lethalSpikes>().SendMessage("Interact");
        }else{
            t.SendMessage("Interact");
        }
    }
}
```

Here we can see the **powerDown** and **sendInteraction** functions. Please note that both **powerDown** and **powerup** functions call **sendInteraction()** and for ease, only **powerDown** will be shown but not discussed. However, there are a two new terms I would like to discuss:

- **Foreach:** A method to loop through elements in a given array
- **Contains:** A method for strings that returns true if the given string is found at any point within the method string.

For this section we will focus on the **sendInteraction()** function. This function is responsible for sending out the interact message to objects within a list of objects, in this case called **interactionPoint**. To do this, when the **sendInteraction()** function is called, we run a **foreach** to send the interact message. However, as you can see, we have a conditional statement that check if the name of **t**, the current interactionPoint in the loop, **Contains("LethalSpikes")**. This is done as the lethal spike objects do not directly have the script **lethalSpikes.cs**. Instead, being in a child object called **spikes**. Hence why we must check and go to the relevant child object to correctly send the interact message. Otherwise, the message is sent directly into the object. This is block is primarily seen within the game to open hatches or enable spikes.



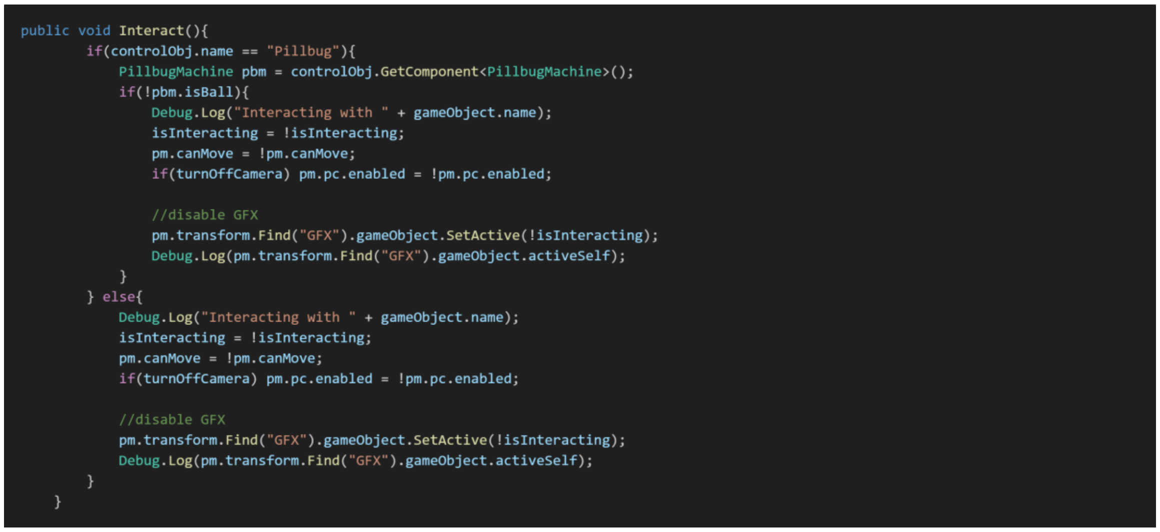
```
void FixedUpdate()
    {
        if(isOn) handlePiston();
    }

    void handlePiston(){
        float dist = Vector3.Distance(pistonHead.position, targetPos.position);

        if(dist > .1f && waitTimer <= 0){
            pistonHead.position = Vector3.MoveTowards(pistonHead.position, targetPos.position, currSpeed * Time.deltaTime);
        }

        if(dist <= .1f){
            if(targetPos == endPos && !hasPushed){
                hasPushed = true;
                checkTrigger();
            }

            waitTimer += Time.deltaTime;
            if(waitTimer >= waitTime){
                hasPushed = false;
                waitTimer = 0;
                targetPos = targetPos == endPos ? startPos : endPos;
                currSpeed = targetPos == endPos ? pistonSpeed : pistonSpeed / 2;
            }
        }
    }

    void checkTrigger(){
        Collider[] hitColliders = Physics.OverlapBox(pistonHead.position + triggerOffset, triggersize / 2, Quaternion.
identity);
        foreach(Collider col in hitColliders){
            Debug.Log(col.name);
            if(col.gameObject.layer == LayerMask.NameToLayer("Player")){
                pm.Knockback(transform.up, pushMod);
            }
        }
    }
```

The above block is taken from the **pistonLogic.cs** script. This is responsible for a piston object that can push players in the direction of the piston. For this, we only must define two new terms:

- **MoveTowards:** A built-in vector method that moves an object towards the end point. Requires three parameters (start, end, step).
- **Time.deltaTime:** Interval in seconds from last frame to the current one.

The logic behind the piston can be broken into two parts. The piston position and the application of force. The position is handled by a timer, called **waitTimer**, and its distance between the current position and the **targetPos**. The timer is adding **Time.deltaTime** to count upwards in seconds. Here you can see that **targetPos** and the **currSpeed** are set by the timer, simply switching between two options depending on if target is equal to the desired end position.

checkTrigger() is the function that handles the knockback. First by getting all the colliders that are within a given **overlapBox(…)**, the box located on the platform of the piston, then running a **foreach** loop to check if the object in the current iteration of the array is the player. If it is, then it pushes the player **transform.up**, which is relative to its global rotation, with a given force of **pushMod**. This allows players to be propelled into new areas to progress further.



```
public void Interact(){
        if(controlObj.name == "Pillbug"){
            PillbugMachine pbm = controlObj.GetComponent<PillbugMachine>();
            if(!pbm.isBall){
                Debug.Log("Interacting with " + gameObject.name);
                isInteracting = !isInteracting;
                pm.canMove = !pm.canMove;
                if(turnOffCamera) pm.pc.enabled = !pm.pc.enabled;

                //disable GFX
                pm.transform.Find("GFX").gameObject.SetActive(!isInteracting);
                Debug.Log(pm.transform.Find("GFX").gameObject.activeSelf);
            }
        } else{
            Debug.Log("Interacting with " + gameObject.name);
            isInteracting = !isInteracting;
            pm.canMove = !pm.canMove;
            if(turnOffCamera) pm.pc.enabled = !pm.pc.enabled;

            //disable GFX
            pm.transform.Find("GFX").gameObject.SetActive(!isInteracting);
            Debug.Log(pm.transform.Find("GFX").gameObject.activeSelf);
        }
    }
```

The following code is from **controlPanel.cs** script and the final script we will look at for interactions. This script is responsible for interactions between the player and machines and is import to look at before moving onto how the machines were programmed. With this, we must look at one new term:

- **SetActive:** Enable or disable an object from a script.

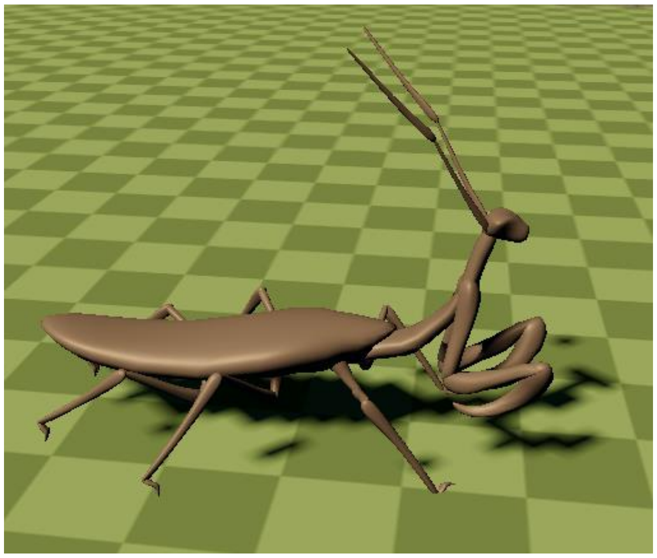
Both sections of the code are similar with one difference so I will focus on the similarities first. Both the if and else portion enable, disable and define bools. The line "**isInteractin = !isInteracting**" refers to swapping the current state of **isInteracting**. The same logic is performed for **pm.canMove**, which if we look back the **playerMotor** script on page 32, which determines most of the functionality of the player movement. We also disable/enable the camera depending on if the control panels variable **turnOffCamera** is true or false.

The main difference between the if and else results is checking if the control panels target object is named "Pillbug". This is because the controllability of the pill bug is greatly determined by whether it is in a state called **isBall**. If the pillbug is a ball, we don't want the player to be able to eject in an uncontrollable situation, therefore we make the restriction that the pill bug must be out of ball form in order to enter or exit. Pill bug will be discussed in the following chapter on controllable machines.

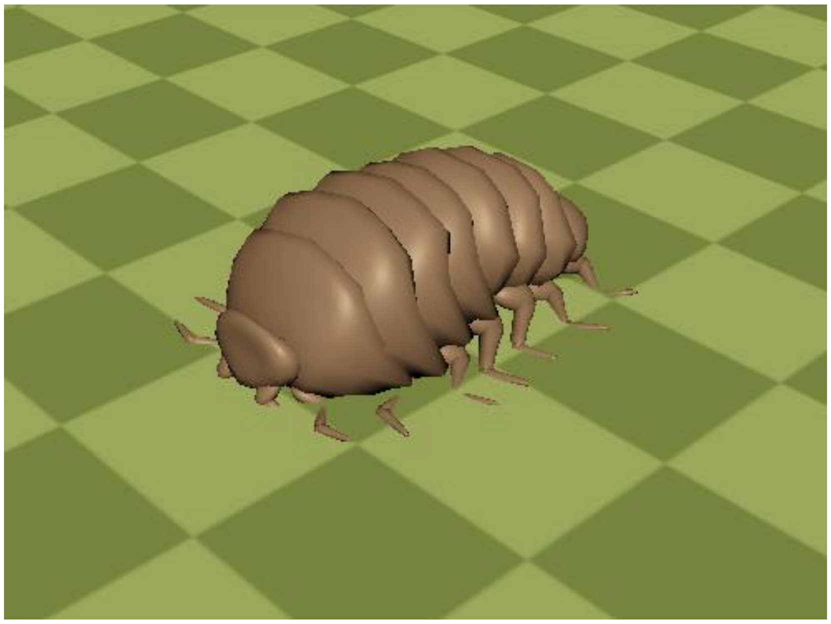
Controllable Machines

In this section I will discuss how the controllable machines were made, their use cases and why they were or weren't used in the final application. There are 3 main machines I will talk about, these being the Pill Bug, Mantis and Claw Machine. Please see the images below to understand how each machine looked and their potential uses.

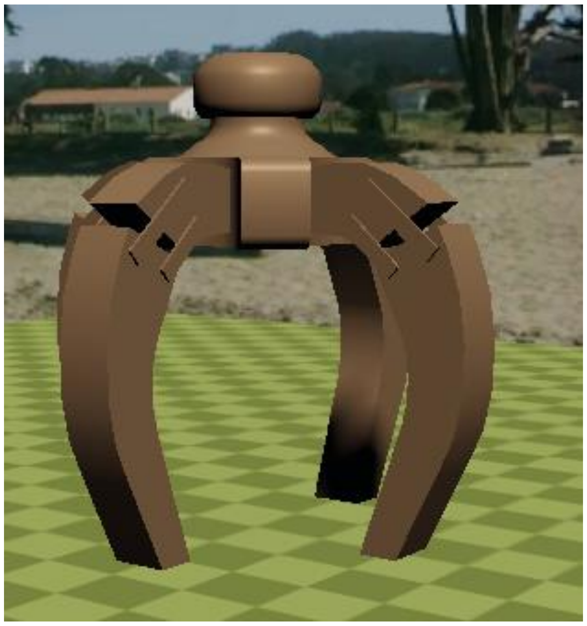
Mantis

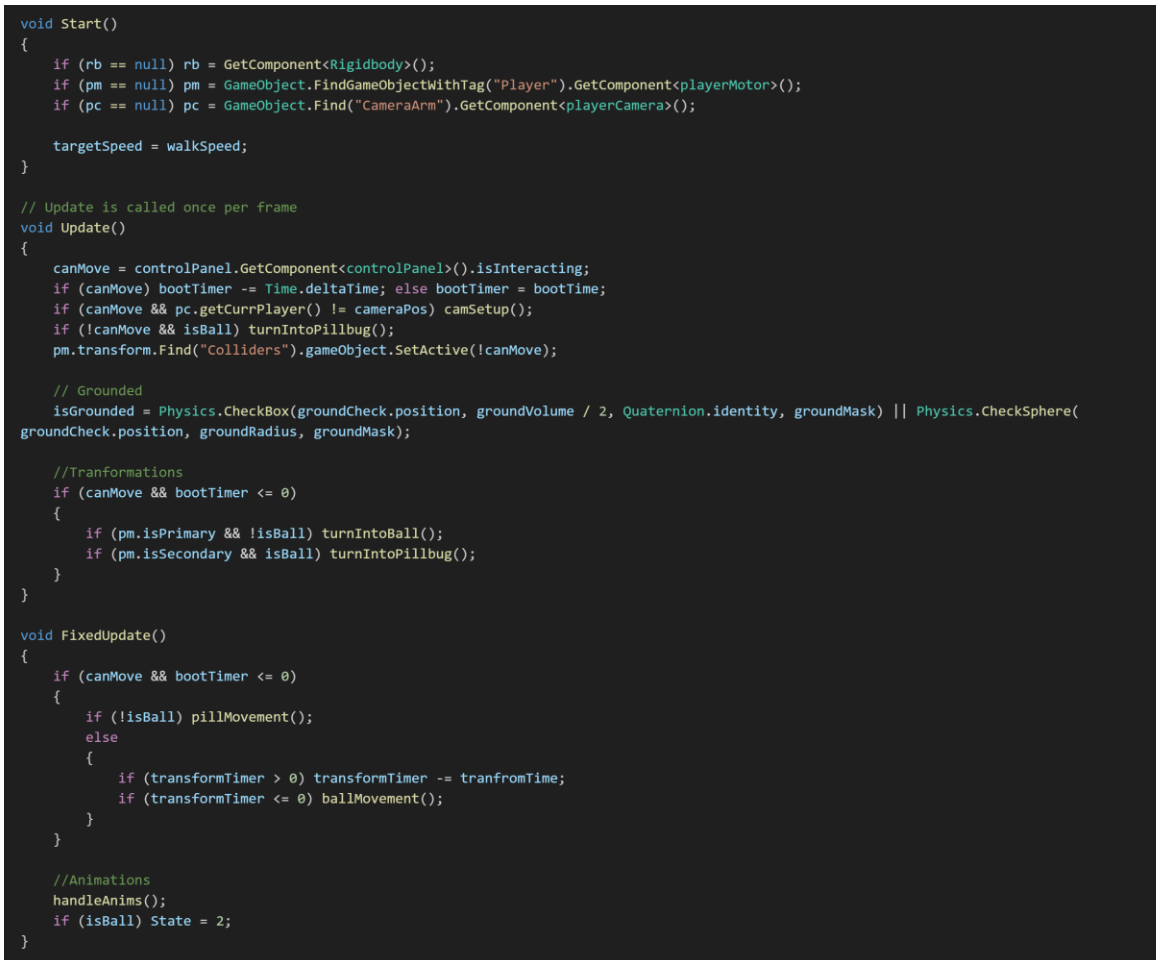


Pill bug



Claw Machine



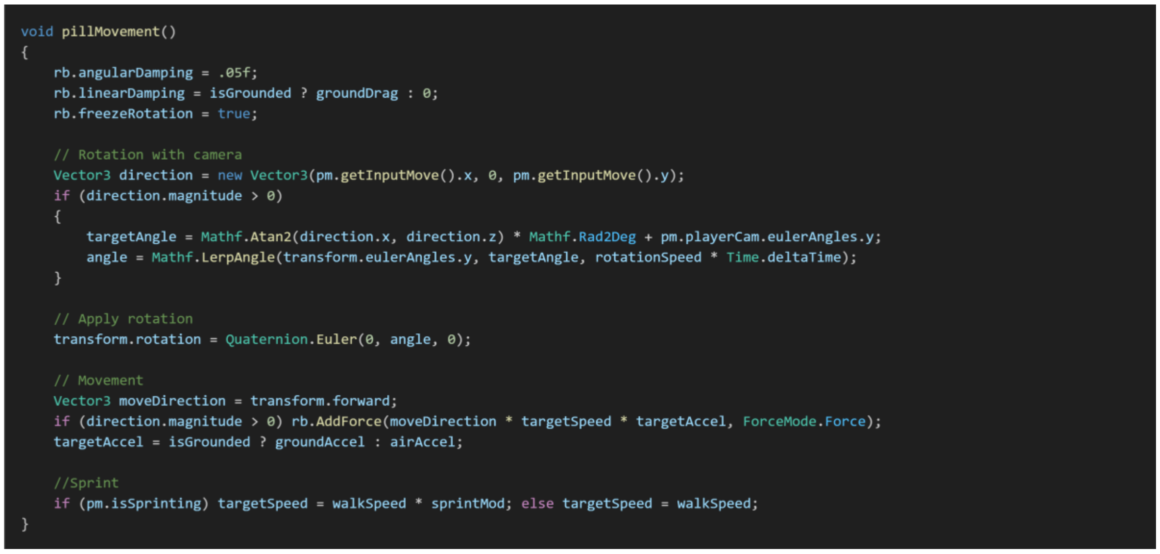


```
void Start()
{
    if (rb == null) rb = GetComponent<Rigidbody>();
    if (pm == null) pm = GameObject.FindGameObjectWithTag("Player").GetComponent<playerMotor>();
    if (pc == null) pc = GameObject.Find("CameraArm").GetComponent<playerCamera>();

    targetSpeed = walkSpeed;
}

// Update is called once per frame
void Update()
{
    canMove = controlPanel.GetComponent<controlPanel>().isInteracting;
    if (canMove) bootTimer -= Time.deltaTime; else bootTimer = bootTime;
    if (canMove && pc.getCurrPlayer() != cameraPos) camSetup();
    if (!canMove && isBall) turnIntoPillbug();
    pm.transform.Find("Colliders").gameObject.SetActive(!canMove);

    // Grounded
    isGrounded = Physics.CheckBox(groundCheck.position, groundVolume / 2, Quaternion.identity, groundMask) || Physics.CheckSphere(
groundCheck.position, groundRadius, groundMask);

    //Tranformations
    if (canMove && bootTimer <= 0)
    {
        if (pm.isPrimary && !isBall) turnIntoBall();
        if (pm.isSecondary && isBall) turnIntoPillbug();
    }
}

void FixedUpdate()
{
    if (canMove && bootTimer <= 0)
    {
        if (!isBall) pillMovement();
        else
        {
            if (transformTimer > 0) transformTimer -= tranfromTime;
            if (transformTimer <= 0) ballMovement();
        }
    }

    //Animations
    handleAnims();
    if (isBall) State = 2;
}
```

The above code is taken from the **PillbugMachine.cs** script which is responsible for controlling said pillbug machine. The pillbug machine has two main states, ball mode and normal mode. Normal mode uses physical forces in similar fashion of the player movement script. Ball mode uses physical forces to imply a torque that rolls the machine around allowing for unique solutions to levels such as ramps. We will breifly go over the purpose for the **Start()**, **Update()** and **FixedUpdate()** functions before going indept in the more important movement functions.

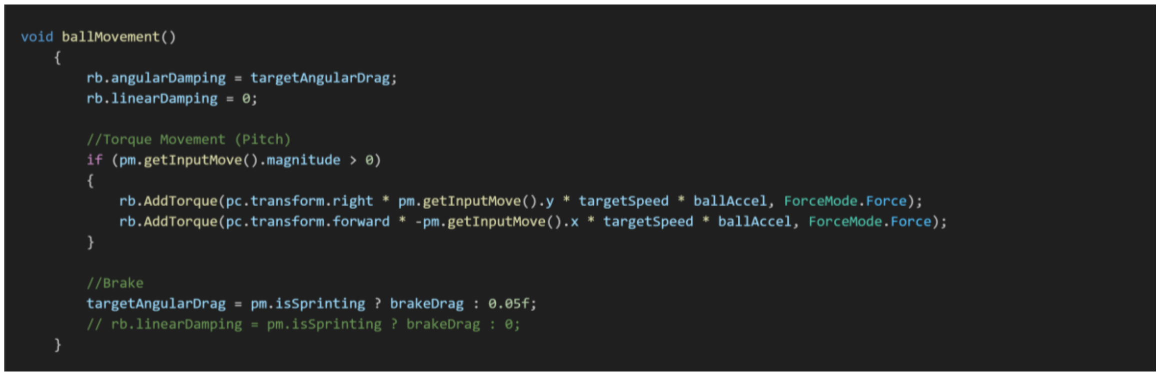
Start(), similarly to the **playerMotor** script defines needed components such as rigid bodies, the player movement scripts and camera. **FixedUpdate()** is used to handle the physics engine that allows the machine to move. The only required check is if the machine **isBall** to determine whether it’s a linear force or an angular force. These two force types will be explored deeper when looking at the movement.

Update() again is used for the main logic that isnt related to the physics engine. This includes checking if the machine is grounded, same method as the player, and controlling if the machine transforms between ball and normal mode. However, **canMove** for the machine is determined by the linked **controlPanel** and if the player is interacting.



```
void pillMovement()
{
    rb.angularDamping = .05f;
    rb.linearDamping = isGrounded ? groundDrag : 0;
    rb.freezeRotation = true;

    // Rotation with camera
    Vector3 direction = new Vector3(pm.getInputMove().x, 0, pm.getInputMove().y);
    if (direction.magnitude > 0)
    {
        targetAngle = Mathf.Atan2(direction.x, direction.z) * Mathf.Rad2Deg + pm.playerCam.eulerAngles.y;
        angle = Mathf.LerpAngle(transform.eulerAngles.y, targetAngle, rotationSpeed * Time.deltaTime);
    }

    // Apply rotation
    transform.rotation = Quaternion.Euler(0, angle, 0);

    // Movement
    Vector3 moveDirection = transform.forward;
    if (direction.magnitude > 0) rb.AddForce(moveDirection * targetSpeed * targetAccel, ForceMode.Force);
    targetAccel = isGrounded ? groundAccel : airAccel;

    //Sprint
    if (pm.isSprinting) targetSpeed = walkSpeed * sprintMod; else targetSpeed = walkSpeed;
}
```

Above is the code responsible for handling the movment of the pill bug while not in ball mode. As you can see, this is quite similar to the player movement, almost exactly. The main difference can be seen at the top. The first three lines are important in order to make sure the movement between the ball and normal mode don’t interfere with one another. In order to do this we set the damping, or drag, for the linear and angular velocities. We do this by referencing the rigid body, **rb** in this script, and setting the relevent property. For example, **rb.angularDamping = .05f;** sets the angular drag and **rb.freezeRotation = true;** makes it so the machine can not rotate.

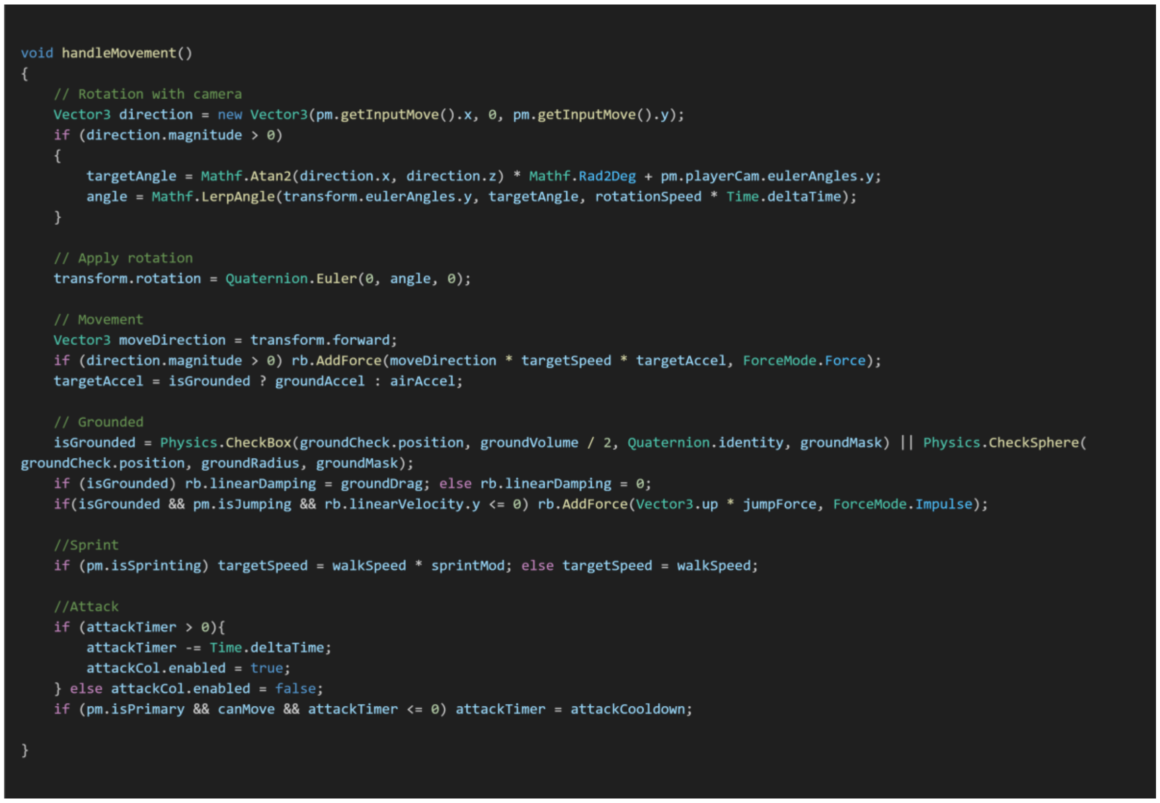


```
void ballMovement()
    {
        rb.angularDamping = targetAngularDrag;
        rb.linearDamping = 0;

        //Torque Movement (Pitch)
        if (pm.getInputMove().magnitude > 0)
        {
            rb.AddTorque(pc.transform.right * pm.getInputMove().y * targetSpeed * ballAccel, ForceMode.Force);
            rb.AddTorque(pc.transform.forward * -pm.getInputMove().x * targetSpeed * ballAccel, ForceMode.Force);
        }

        //Brake
        targetAngularDrag = pm.isSprinting ? brakeDrag : 0.05f;
        // rb.linearDamping = pm.isSprinting ? brakeDrag : 0;
    }
```

The above code is responsible for the ball movement. As you can see, it is entirely different from any movement code we have previously explored. Introducing a new method:

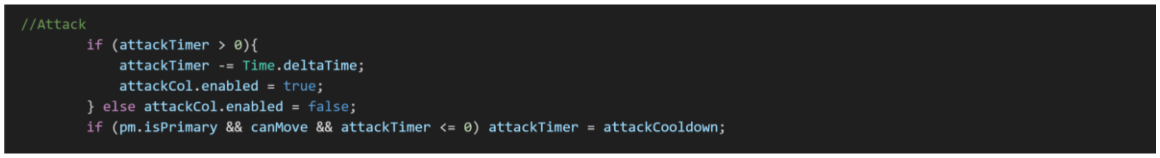
- **AddTorque:** Adds a torque to the rigidbody.

This is how we allow the player to add angular velocity to the machine, resulting in the machine rolling around. This means that by giving a direction, torque speed and the **ForceMode**, this case **Force** because we want mass to play a role into the speed, we can roll the machine around and progress the level.



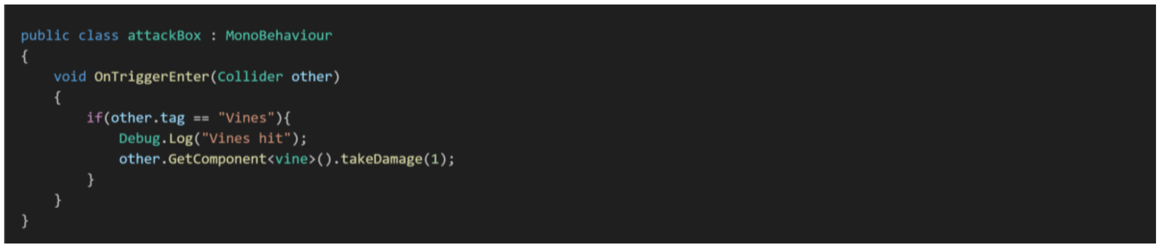
```
void handleMovement()
{
    // Rotation with camera
    Vector3 direction = new Vector3(pm.getInputMove().x, 0, pm.getInputMove().y);
    if (direction.magnitude > 0)
    {
        targetAngle = Mathf.Atan2(direction.x, direction.z) * Mathf.Rad2Deg + pm.playerCam.eulerAngles.y;
        angle = Mathf.LerpAngle(transform.eulerAngles.y, targetAngle, rotationSpeed * Time.deltaTime);
    }

    // Apply rotation
    transform.rotation = Quaternion.Euler(0, angle, 0);

    // Movement
    Vector3 moveDirection = transform.forward;
    if (direction.magnitude > 0) rb.AddForce(moveDirection * targetSpeed * targetAccel, ForceMode.Force);
    targetAccel = isGrounded ? groundAccel : airAccel;

    // Grounded
    isGrounded = Physics.CheckBox(groundCheck.position, groundVolume / 2, Quaternion.identity, groundMask) || Physics.CheckSphere(
groundCheck.position, groundRadius, groundMask);
    if (isGrounded) rb.linearDamping = groundDrag; else rb.linearDamping = 0;
    if(isGrounded && pm.isJumping && rb.linearVelocity.y <= 0) rb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);

    //Sprint
    if (pm.isSprinting) targetSpeed = walkSpeed * sprintMod; else targetSpeed = walkSpeed;

    //Attack
    if (attackTimer > 0){
        attackTimer -= Time.deltaTime;
        attackCol.enabled = true;
    } else attackCol.enabled = false;
    if (pm.isPrimary && canMove && attackTimer <= 0) attackTimer = attackCooldown;

}
```

Above is the code block from the **MantisMachine.cs** script is responsible for handling the movement of the machine. Simlar to **PillbugMachine.cs**, the movement is almost identical to the default **playerMotor.cs**. However, we can see a new section with the comment **//Attack**. This handles the attack function that is unique to the mantis, allowing the player to cut through vines and progress the level. In order to understand this further, lets have a deeper look into how attacks are handles.



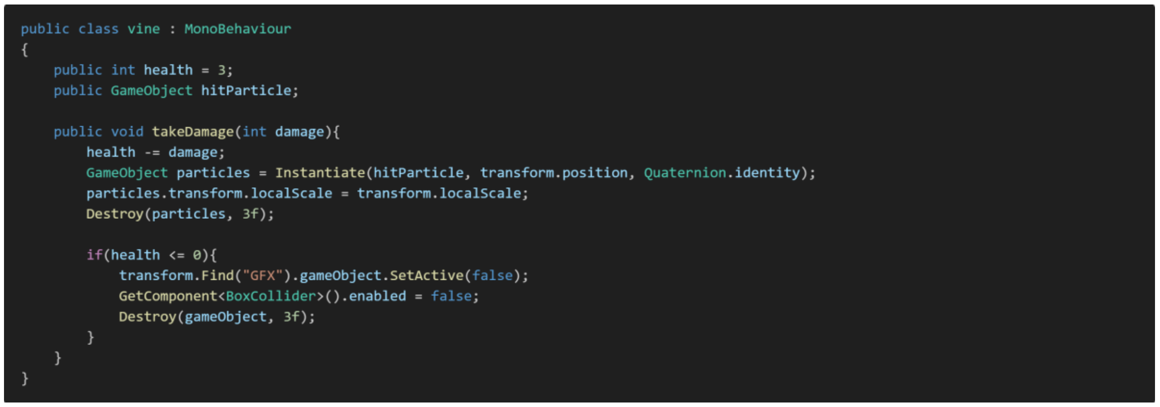
```
//Attack
        if (attackTimer > 0){
            attackTimer -= Time.deltaTime;
            attackCol.enabled = true;
        } else attackCol.enabled = false;
        if (pm.isPrimary && canMove && attackTimer <= 0) attackTimer = attackCooldown;
```

Here we see the attacks are handle by two if statements. The variable **attackTimer** is what determins if the player is attacking. As you can see, if greater than 0 a collider, called **attackCol** is enabled. In order to understand what happens we must look at that collider and the unique script called **attackBox.cs** to understand how it works.



```
public class attackBox : MonoBehaviour
{
    void OnTriggerEnter(Collider other)
    {
        if(other.tag == "Vines"){
            Debug.Log("Vines hit");
            other.GetComponent<vine>().takeDamage(1);
        }
    }
}
```

Here you can see the **attackBox.cs** script attached to the collider **attackCol**. We also notice a familiar function, **OnTriggerEnter** which we understand handles colliders that intersect with the trigger zone on an object. The logic within the function requires that the other object has the tag of “Vines” in order for it to do damage to the object. We can see it gets a component by the name of **vine** and calls a method **takeDamage()** to apply damage.

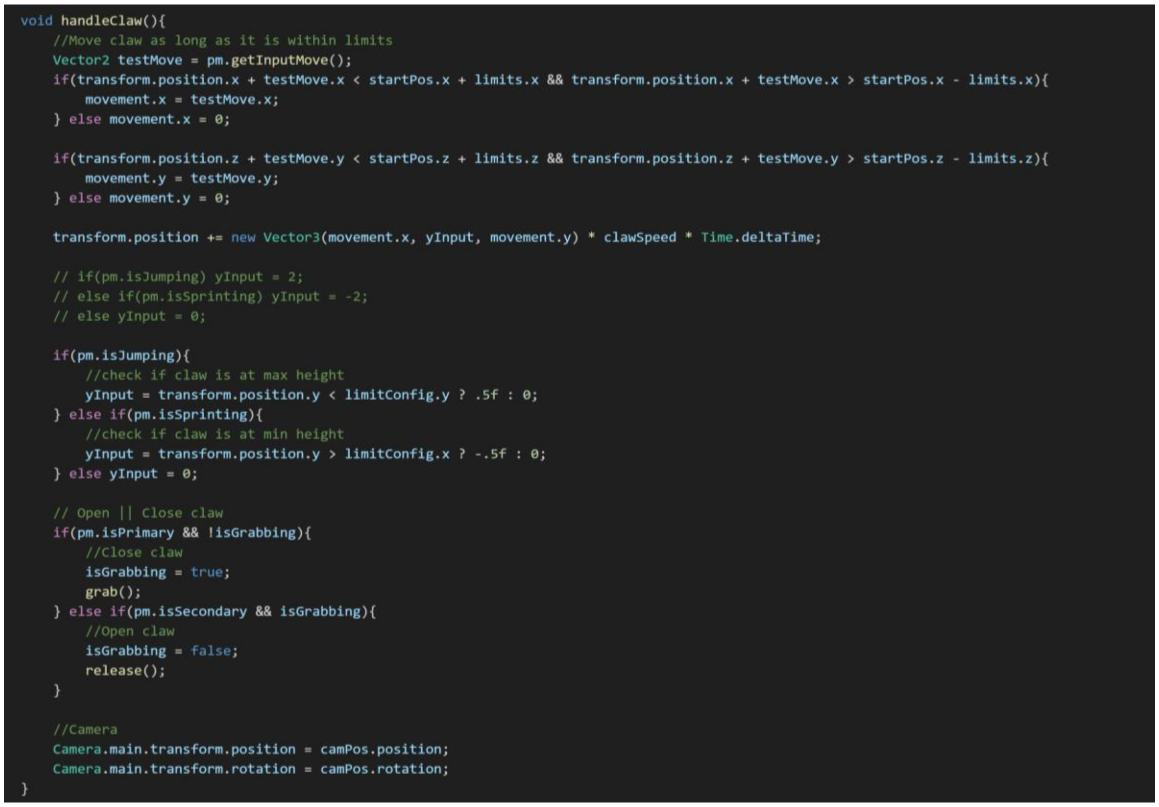


```
public class vine : MonoBehaviour
{
    public int health = 3;
    public GameObject hitParticle;

    public void takeDamage(int damage){
        health -= damage;
        GameObject particles = Instantiate(hitParticle, transform.position, Quaternion.identity);
        particles.transform.localScale = transform.localScale;
        Destroy(particles, 3f);

        if(health <= 0){
            transform.Find("GFX").gameObject.SetActive(false);
            GetComponent<BoxCollider>().enabled = false;
            Destroy(gameObject, 3f);
        }
    }
}
```

If we look at the **vine.cs** script, we can see the exact method called from the **attackBox.cs** script. Lets define the two new terms:

- **Instantiate:** Clones the given object and returns the clone.
- **Destroy:** Destroys the referenced object in N seconds if a time is given, otherwise instantly.

Going line by line we see that the vine loses health equal to the damage we pass in, instanties a new game object called **hitParticle** and destroys the object in 3 seconds. It then checks if the health is less than or equal to 0. If it is, it disables the child object “GFX”, disables the collider and destroys the vine object in 3 seconds.



```
void handleClaw(){
    //Move claw as long as it is within limits
    Vector2 testMove = pm.getInputMove();
    if(transform.position.x + testMove.x < startPos.x + limits.x && transform.position.x + testMove.x > startPos.x - limits.x){
        movement.x = testMove.x;
    } else movement.x = 0;

    if(transform.position.z + testMove.y < startPos.z + limits.z && transform.position.z + testMove.y > startPos.z - limits.z){
        movement.y = testMove.y;
    } else movement.y = 0;

    transform.position += new Vector3(movement.x, yInput, movement.y) * clawSpeed * Time.deltaTime;

    // if(pm.isJumping) yInput = 2;
    // else if(pm.isSprinting) yInput = -2;
    // else yInput = 0;

    if(pm.isJumping){
        //check if claw is at max height
        yInput = transform.position.y < limitConfig.y ? .5f : 0;
    } else if(pm.isSprinting){
        //check if claw is at min height
        yInput = transform.position.y > limitConfig.x ? -.5f : 0;
    } else yInput = 0;

    // Open || Close claw
    if(pm.isPrimary && !isGrabbing){
        //Close claw
        isGrabbing = true;
        grab();
    } else if(pm.isSecondary && isGrabbing){
        //Open claw
        isGrabbing = false;
        release();
    }

    //Camera
    Camera.main.transform.position = camPos.position;
    Camera.main.transform.rotation = camPos.rotation;
}
```

The above code is taken from **clawMachine.cs** script and is responsible for handling all the logic. The top half is how the movement is performed. Creating a **Vector2 testMove** in order to determin if the next frame of movement would result in the claw exiting predetermined limits, in this case a **Vector3** called **limits**. Depending on the results, we players X and Z can be updated, otherwise it dosnt go any further.

In order to determine the Y position, its important to look at the if statements that tetermin the **yInput**. If we look we can we see two bools being used from **pm** which in this case refers to **playerMotor.** If we look back, we can see that this corolates with the list of bools I noted previously. These bools update true or false based on player inputs. The bools and the linked inputs are:

- **isJumping:** Linked to the jump button (PC: Space. Controller: A)
- **isSprinting:** Linked to sprint button (PC: Left Shift. Controller: Left Sholder)
- **isInteracting:** Linked to interact button (PC: F. Controller, Controller: X)
- **isPrimary:** Linked to primary action button (PC: Left Click. Controller: Right Trigger)
- **isSecondary:** Linked to secondary action button (PC: Right Click. Controller: Left Trigger)

If we look at all the machine scripts, we will notice that there are references to these bools in order to control key functions like tranforming into a ball or attacking. In this case, our **isJumping** and **isSprinting** are responsible for raising or lowering the claw. With this in mind we can see that **isPrimary** and **isSecondary** are responsible for grabbing and dropping objects.

Conclusion: Player Functionality

During the process of implementing player functionality I had run into multiple issues that had to be resolved. I previously mentioned that player direction was **transform.forward + (transform.up/2)** because of slopes. Previously this was absent and resulted in the player unable to traverse sloped surfaces. However, this method allowed the player to walk on slopes that were not too steep.

You may notice that within the game there is an absence of the claw machine in the final product. This was due to the feeling that it didn’t belong in the current world. While the functionality was interesting and, in the future, I would like to use the experience I gained while working on the claw machine script, I believe I made the right decision not to include it within this project.

Overall, I am happy with the result of player functionality. The interaction scripts allow for scalability with simple calling of functions while also having room for complex interactions. The movement feels fluid and makes logical sense once the understanding of physics-based movement is realised. If given a chance I would like to expand on this type of movement further, potentially working on a physics-based car movement or space adventure game that focuses on zero-gravity while being physics-based.

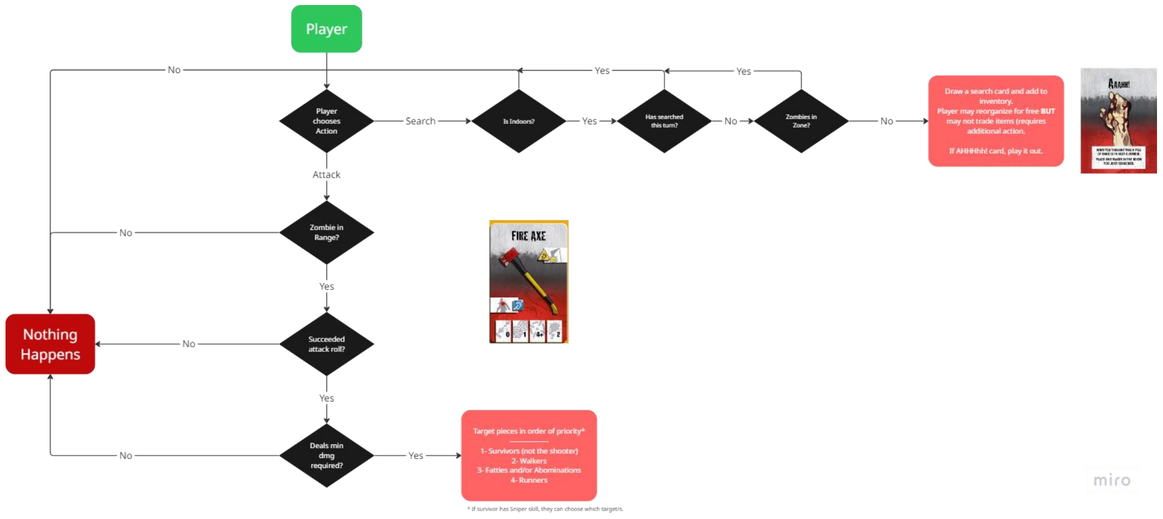
AI Behaviour and Pathfinding

In this section I will go over the scripts, logic and steps taken to create the AI within Scarabs Core. There is only one AI agent within my project but the experience I gained from working with it was extremely valuable non the less. To correctly inform the steps and reasoning behind the process I took, this section will be cut into three important parts:

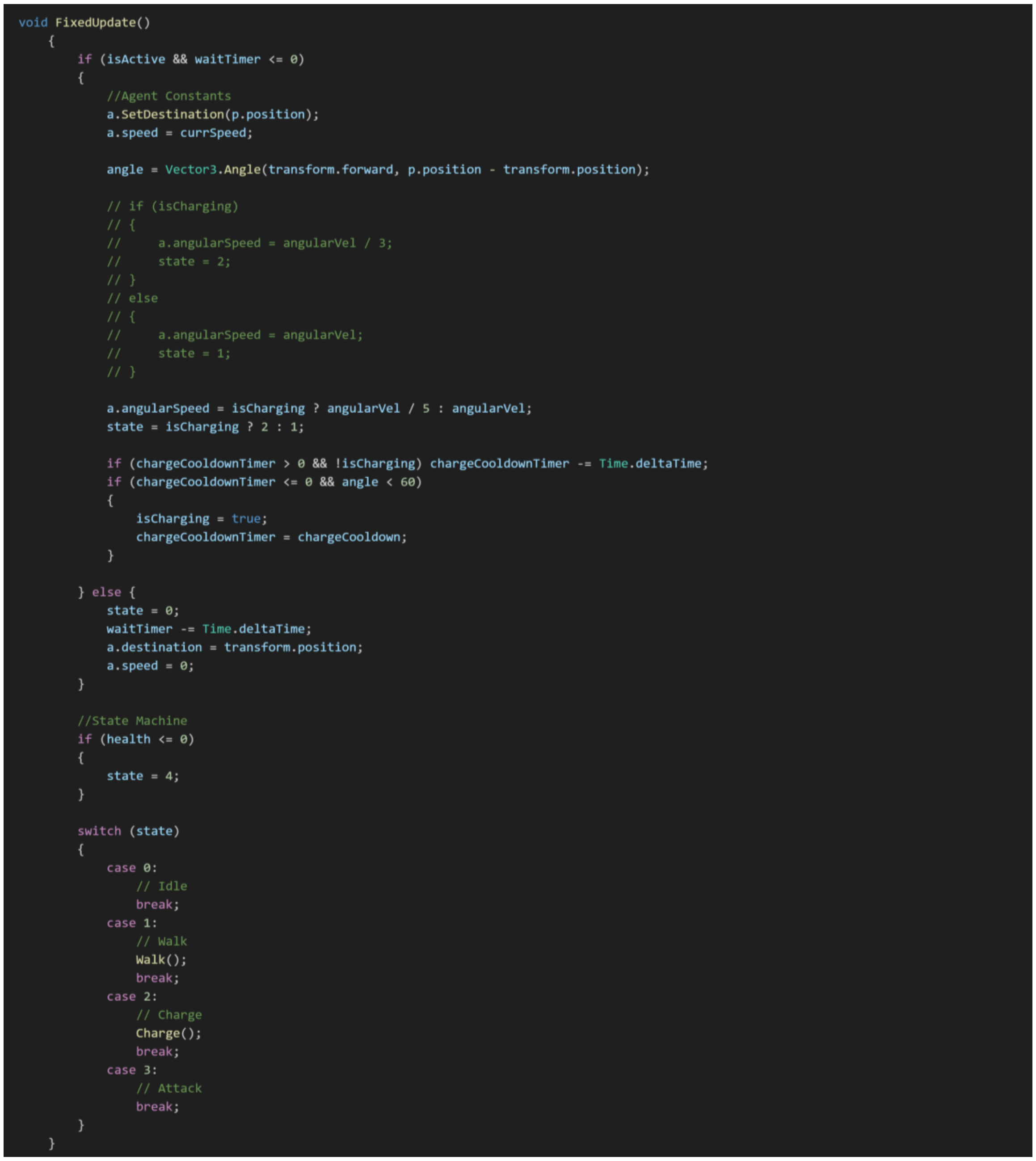
1. **Finite State Machine:** This is the behaviour type I decided to use for my AI and will explain what this means, how it was achieved, pros and cons.

2. **Pathfinding:** With the use of NavMesh within Unity I will talk about the process of creating the NavMesh surface, the basics of understanding the A* algorithm and the process of calculating paths and using them within scripts.Finite State Machine

A Finite State Machine (FSM) is a model to show the steps by which the logic will follow. An important thing to keep in mind is that FSMs can only ever be in one state of a finite amount of pre-defined states. As a good example of this, think of this as the logical steps it takes for an enemy to attack. Below is an example of a FSM that I made working on a previous project


Player
Player chooses Action
Search
Is Indoors?
Yes
Has searched this turn?
No
Zombies in Zone?
No
Yes
Yes
Draw a search card and add to inventory.
Player may reorganize for free BUT may not trade items (requires additional action.
If AHHHhh! card, play it out.
AAAHH!
Attack
Zombie in Range?
No
Yes
FIRE AXE
Succeeded attack roll?
No
Nothing Happens
Yes
Deals min dmg required?
No
Yes
Target pieces in order of priority*
1- Survivors (not the shooter)
2- Walkers
3- Fatties and/or Abominations
4- Runners
* If survivor has Sniper skill, they can choose which target/s.
miro

As you can see, it is a logical step by step process of true/false statements to determine the outcome of an action. These steps must lead towards an outcome, be it what the player was intending or nothing.



```
void FixedUpdate()
    {
        if (isActive && waitTimer <= 0)
        {
            //Agent Constants
            a.SetDestination(p.position);
            a.speed = currSpeed;

            angle = Vector3.Angle(transform.forward, p.position - transform.position);

            // if (isCharging)
            // {
            //     a.angularSpeed = angularVel / 3;
            //     state = 2;
            // }
            // else
            // {
            //     a.angularSpeed = angularVel;
            //     state = 1;
            // }

            a.angularSpeed = isCharging ? angularVel / 5 : angularVel;
            state = isCharging ? 2 : 1;

            if (chargeCooldownTimer > 0 && !isCharging) chargeCooldownTimer -= Time.deltaTime;
            if (chargeCooldownTimer <= 0 && angle < 60)
            {
                isCharging = true;
                chargeCooldownTimer = chargeCooldown;
            }

        } else {
            state = 0;
            waitTimer -= Time.deltaTime;
            a.destination = transform.position;
            a.speed = 0;
        }

        //State Machine
        if (health <= 0)
        {
            state = 4;
        }

        switch (state)
        {
            case 0:
                // Idle
                break;
            case 1:
                // Walk
                Walk();
                break;
            case 2:
                // Charge
                Charge();
                break;
            case 3:
                // Attack
                break;
        }
    }
```

The above code is directly taken from the **enemyRhino.cs** script. This script is responsible for handling both the FSM and NavMesh agents' pathfinding. However, in this section we will focus on the FSM. Once again, I will define important terms that will aid readers in greater understanding the scripts. These being:

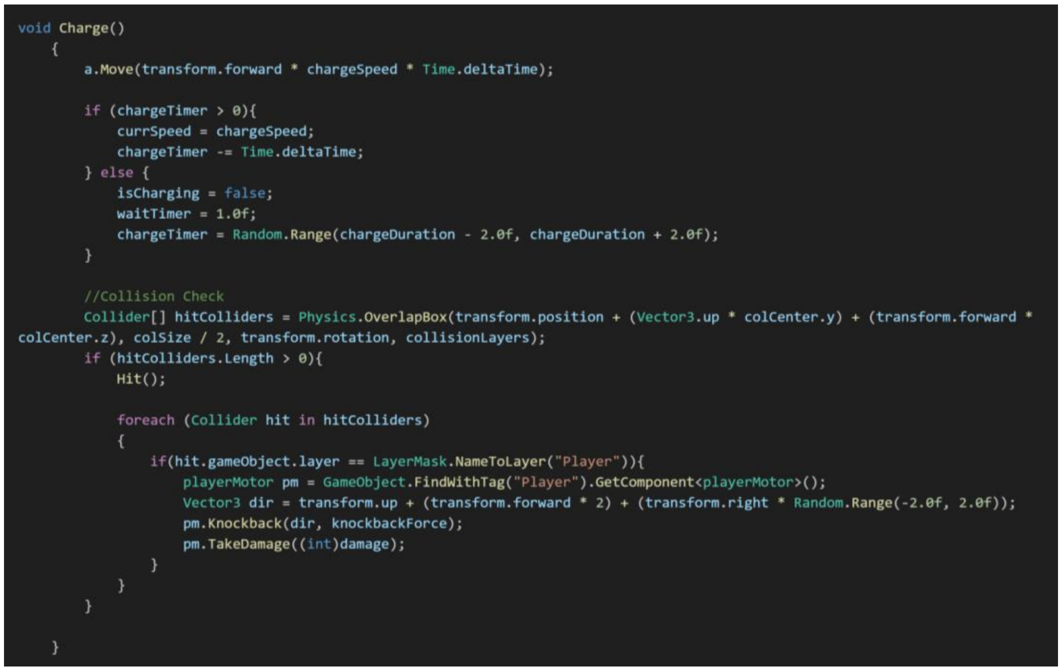
- **SetDestination:** This is the target location of the NavMeshAgent.
- **NavMeshAgent:** This is the name given to a character to allow the use of NavMesh to navigate the scene.
- **NavMesh:** Used to perform spatial queries such as pathfinding and walkability tests.
- **Angle:** Method of Vectors used to return the angle between two direction.

In this case we must note that **a** is a variable assigned to the **NavMeshAgent**. The beetle is constantly looking for the players position, represented by **p.position**, therefore is constantly in an active search state for the path to the player. However, we will focus on how the variable **state** controls the behaviour of this beetle.

State has four possible states:

1. **Idle:** Inactive, doing nothing.
2. **Walk:** Basic movement towards a goal.
3. **Charge:** The main attack of the beetle, charge towards the goal.
4. **Attack:** A state that happens if the charge attack hits.

If we look at the line "**state = isCharging ? 2 : 1;**" we can see that the bool **isCharging** determines the state of this agent. However, we can see that the if statement above is making sure the agent isn't waiting and is active, other wise the **state** is set to 0, with is Idle. Looking at the switch statement, you see that depending on **state**, certain functions get called.



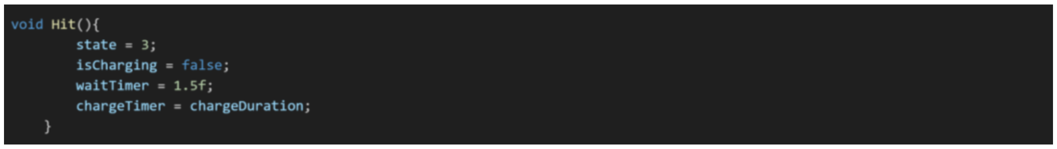
```
void Charge()
    {
        a.Move(transform.forward * chargeSpeed * Time.deltaTime);

        if (chargeTimer > 0){
            currSpeed = chargeSpeed;
            chargeTimer -= Time.deltaTime;
        } else {
            isCharging = false;
            waitTimer = 1.0f;
            chargeTimer = Random.Range(chargeDuration - 2.0f, chargeDuration + 2.0f);
        }

        //Collision Check
        Collider[] hitColliders = Physics.OverlapBox(transform.position + (Vector3.up * colCenter.y) + (transform.forward *
colCenter.z), colSize / 2, transform.rotation, collisionLayers);
        if (hitColliders.Length > 0){
            Hit();

            foreach (Collider hit in hitColliders)
            {
                if(hit.gameObject.layer == LayerMask.NameToLayer("Player")){
                    playerMotor pm = GameObject.FindWithTag("Player").GetComponent<playerMotor>();
                    Vector3 dir = transform.up + (transform.forward * 2) + (transform.right * Random.Range(-2.0f, 2.0f));
                    pm.Knockback(dir, knockbackForce);
                    pm.TakeDamage((int)damage);
                }
            }
        }

    }
```

Here we can see the **Charge()** function. Before looking further into this function we should investigate new terms. These being:

- **Move:** Move the NavMeshAgent in a given direction and speed.
- **Random.Range:** Returns a random float between two given numbers, min and max.
- **FindWithTag:** Similar to Find, this method is used to find the first object with the tag specified.

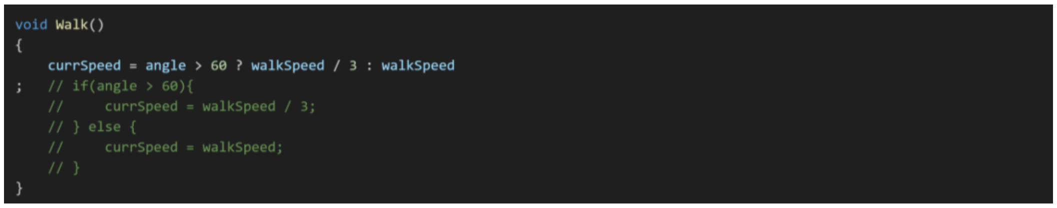
We can see that when the agent is set to **state** 2, it is set to charge. Running the **Charge()** function every frame. The line **a.Move(…)** to move the agent in the forward direction in a speed defined as **chargeSpeed** multiplied by **Time.deltaTime** to keep the speed consistent between frames. Below the **a.Move()** method we see that as long as **chargeTimer** is greater than zero, it will constantly be charging, **chargeTimer** going down every frame by **Time.deltaTime**. Once **chargeTimer** is less than or equal to zero, it is considered no longer charging, the **waitTimer** is set to 1, which in turn sets the **state** to 0 which was defined as Idle.

We can see collision checks happening on the lower half. Similar to the mantis machine, we are creating an array of colliders that enter a specific defined zone, in this case it is the collider box size. Checking that the array is not empty, it runs a separate function **Hit()**.



```
void Hit(){
        state = 3;
        isCharging = false;
        waitTimer = 1.5f;
        chargeTimer = chargeDuration;
    }
```

Above is said function **Hit().** As you can see if charge attack hits, the **state** is set to 3 and timers are reset. Under the **Hit()** function call, we can see a **foreach** loop that runs through the array **hitColliders**. Checking each collider, if the collider is found to be the player it applies a knockback via **pm.Knockback(…)** and damages the player via **pm.TakeDamage(…)**. Not unlike the spikes, this is the primary way that damage and knockback are always applied to the player.



```
void Walk()
{
    currSpeed = angle > 60 ? walkSpeed / 3 : walkSpeed
;   // if(angle > 60){
    //     currSpeed = walkSpeed / 3;
    // } else {
    //     currSpeed = walkSpeed;
    // }
}
```

Here you can see the **Walk()** function that is called given that **state** is set to 1. It comprises of a single line "**currSpeed = angle > 60 ? walkSpeed / 3 : walkSpeed"**. This line is responsible for making the **NavMeshAgent** walk speed depend on its angle to the player. This is calculated in the **FixedUpdate()** with the line "**angle = Vector3.Angle(transform.forward, p.position – transform.position)**". The direction **transform.forward** is relative to the **NavMeshAgent** rotation while **p.position – transform.position** is relative to the player and **NavMeshAgent**s position in the world. If said angle is greater than 60 the walk speed is slowed down in order to allow the agent to rotate before speeding up agai

Before moving onto Pathfinding and understanding the A* algorithm, lets talk about the pros and cons of the FSM used for the beetle.

Pros:

- FSMs are easy to read and follow the logic line by line, giving the developer an easy understanding of the functionality without running the code.
- Conditional based. Since the states are determined by angles and timers, they are easy to create.

Cons:

- While easy to read, adding additional logic can be difficult as it may interfere with previous defined states.
- Due to being conditional, code and become muddied by if and else statements leading to fatigue as the developer must reference multiple statements.

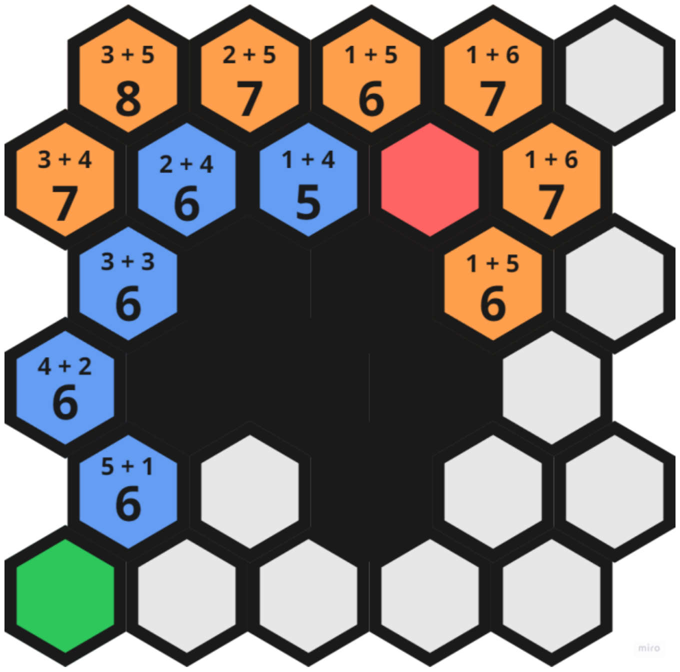
If given the chance to once again do this project and make an AI behavior system, I would like to investigate a behavior tree approach. As Unity upgraded over the past years, they introduce a build in behavior tree system that works similarly to a graph-node system with assignable variables and states that work on conditional logic and loops.

Pathfinding

The pathfinding algorithm you will find used most amongst the gaming industry is the A* algorithm. This is because of its versatility and optimization. A* is used in real-world scenarios such as robotics to aid in navigating obstacles and finding optimal paths. This is the same reason it is found within video games. Its low-cost computations make it one of the most optimized algorithms for pathfinding. But how does A* work?

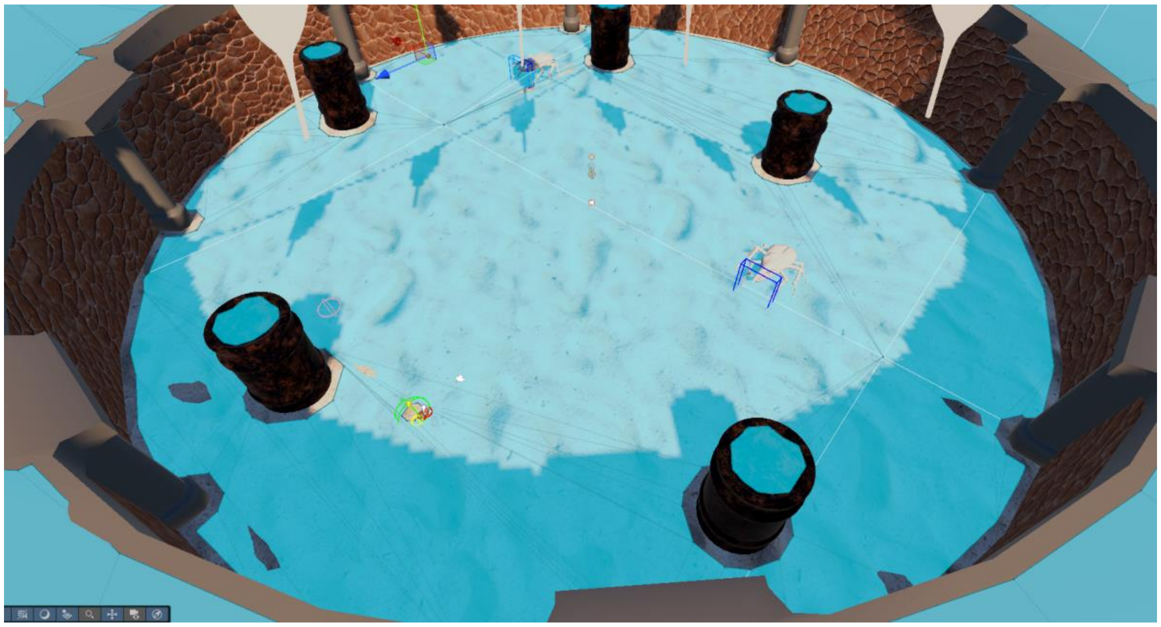
A* works by working with two specific values. Goal cost, or G-Cost, is the distance between the given node and start node. Heuristic cost, or H-Cost, is the cost of the cheapest path from the given node and the goal node. H-Cost is optimistic and will either be exact or underestimated as it disregards obstacles. The F-Cost is the G-Cost plus the H-Cost. This can be written down as F(n) = G(n) + H(n) where n is the current node, not the start node. The lower the F-Cost, the more desired that path is. This calculation is done over and over until the n is equal to the goal node.

While this is over simplified, this is an accurate and helpful way of visualizing how A* works. Looking at the image below you can see an example path made with each of the F values per tile. Red being the start, Green being the goal, Blue being the chosen paths and Orange being the not chose path.



This is the underlying logic behind the A* algorithm. While this knowledge isnt neccisarrily needed to work with the **NavMesh** system within unity, it is important to understand the basic principle behind how the algorithm works.

Within Unity, there is very little code needed to calculate the path and apply it to the agent. This has been done in the previous section with **a.Move(…)** and **a.SetDestination(…)**. They both work with the **NavMesh** and calculate the path for the agent to follow. **Move()** based on direction while **SetDestination()** set on a target position. Instead, let me show how the **NavMesh** was created, baked and what it looks like to the developer.



What the image above is showing is the baked **NavMesh** surface. The blue surfaced represent the areas that are considered "Walkable". When setting the agent path, it will get the closest point from the target on the **NavMesh** and attempt to walk to that point. Whether is it possible or not, the agent will move as close as possible before deciding it has reached its "destination". The **NavMesh** can be baked depending on needs. In my case it was baked to keep in mind the width of the beetle and collidered within the scene that were assigned to specific layers, such as ground, obsticle, etc. This is why you can see gaps between the walls and the blue surface as this is the width of the beetle and without this, the beetle would attempt to walk through the wall.

Conclusion: AI Behaviour and Pathfinding

I am more than happy with the experience and product of my AI behaviour and the pathfinding solutions used. I learned a lot about scripting the pathfinding, making my own Finite State Machine and the logic behind the A* algorithm. If I were to redo this project, as mentioned above, I would love to research and implement a more scalable behaviour tree method. Otherwise, I am proud of what I have achieved with my research and application of said knowledge.

Play Testing and Analysis

In this chapter I will touch on the player testing that happened throughout the development, what insights were learned and how problems that arose were handled.

Play Testing

During the development it was important to consistently play test to ensure that features added worked as intended and there were little to no oversights on my part as the developer. By rigorously testing the Application throughout the entirety of the project, I was able to create working features and have a game that played and flowed as intended. However, it’s important to understand that play testing my own application revealed little of the true problems as developers will mostly play in idealistic ways, following the immediate correct path, understanding puzzles and problems. This was solved by doing public play testing.

Public Testing: Spring Comic Con 2025

During my development I was lucky enough to attend Comic Con in Spring as a student endorsement for the new Game Design course. During which time I also got to show of my project, get people on the floor to test and receive feedback. During this day I gained valuable insight into how players would approach my game, levels and what issues became immediately apparent.

Discovered Issues

While the primary goal of my game was achievable, play 3 levels, end in the arena with 2 large beetles and get knocked around, there were some issues that became apparent after watching people play. The top 3 that effected my game would be:

1. **Difficulty / Misleading Instructions.**

2. **Bugs with player movement.**

3. **Issues with colliders.**

Difficulty / Misleading Instructions

While someone who has experience playing keyboard, mouse and controller might not find it difficult to, people who had little experience in one or the other found sections harder than most. In level one, the player is taught the controls and how to traverse the level. Unfortunately, players failed to understand the instructions to progress through a door. This came down to a writing error on my part. However, people who managed to open the door found the spikes in the floor to be difficult to traverse.

People had difficulty jumping through a section of spikes that rose in a sequence. If patient, the player could walk calmly though the spikes as they raised and lowered however since the first time the player encountered spikes was give the instructions to jump over, it was interpreted as the only possible way to avoid spikes in this section was to jump over. This made the section particularly difficult as the roof was low preventing higher jumps.

Bugs with player movement

During playtesting, someone managed to discover an issue relating to the player movement, specifically the **isGrounded** state. This would flag as true when unintended, allowing the player to effectively jump across certain walls if taken from the right angle. While funny and enjoyable, it was an unintended result of my player motor script and could allow players to exploit certain levels, potentially falling infinitely if they did not hit a death plane.

Issues with Colliders

An issue that arose with colliders came in the last two levels. Level 3 was focused on the mantis; the player was to open a vine covered tunnel to progress. However, the collider that was to prevent the player from jumping out of bounds was missing. Resulting in players who didn't follow the intended path, walking through the wall and falling infinitely. This again is not acceptable as it breaks the players immersion and results in them having to leave the level to retry.

Solutions

Clearer instructions

By adding more clear instructions, players will be able to progress through the levels without getting stuck on sections and being prevented from moving forward. Adding additional tips across the levels reinforces the correct mindset to have when approaching potential puzzles while allowing for experimentation on the players behalf.

Refined Ground Check

The issue with the ground check came about as an over-correction for previous ground check issues. By scaling down the detection zones, I was able to eliminate the issue with wall-jumping without compromising on any of the gameplay that was already present within the levels.

Added Colliders

Being a simple thing to miss, it was a simple thing to correct. Fixing the issue with being able to walk through certain walls was accomplished by adding the appropriate colliders and ensuring they were not set as trigger zones.

Conclusion: Testing

Testing was extremely exciting and effective in showing where attention to detail may have been missed, where bugs went unseen and how the game progressed through the eyes of the average player. I learned so much throughout testing by talking to people, their opinions and what they would like to see if this project gets expanded upon and I will value this experience dearly.

Conclusion and Final Thoughts

Although I have a long way to go when it comes to a completed game, there is no denying that this experience has helped me grow personally and professionally. Being able to demonstrate external skills such as my understanding of 3D Modeling, game development and Unity to reinforcing existing skills, this has been a valuable and unforgettable experience. I got the chance to focus on a single project, refine features and scope in on what could be possible with the correct time and mindset when it comes to game development.

This project was challenging in learning about pathfinding algorithms, researching the history of AI and NPC behaviours to applying physics-based movement rather than character controller. Nevertheless, I am extremely grateful for the chance to learn and experience this growth first hand and look forward to the possibilities it may lead me to.

References

Algorithms. (n.d.). Cs.stanford.edu.

https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/intelligent-search/breadth.html

Simpson, C. (2014, July 18). *Behavior Trees for AI: How They Work*. Game Developer. https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work

rubenwardy. (2022, July 17). *Creating worker NPCs using behavior trees*. Rubenwardy.com. https://blog.rubenwardy.com/2022/07/17/game-ai-for-colonists/

Bycer, J. (2017, November 7). *How to Design a Game Around Emergent Gameplay*. Medium; Medium. https://medium.com/@GWBycer/how-to-design-a-game-around-emergent-gameplay-a9348557570b

What are the most effective techniques for creating emergent AI behaviors? (2024). Linkedin.com. https://www.linkedin.com/advice/0/what-most-effective-techniques-creating-emergent-ai-vwiae

Joseph, M. (2023). Emergent Behaviour in Game AI: A Genetic Programming and CNN-based Approach to Intelligent Agent Design. *Brocku.ca*. http://hdl.handle.net/10464/18167 id-Software. (2024). *DOOM/linuxdoom-1.10/p_enemy.c at master · id-Software/DOOM*. GitHub. https://github.com/id-Software/DOOM/blob/master/linuxdoom-1.10/p_enemy.c

NodeCanvas. (2014). @UnityAssetStore; Unity Asset Store. https://assetstore.unity.com/packages/tools/visual-scripting/nodecanvas-14914

Webster, H. (2024, September 4). *How Big Is A Planet In No Man's Sky?* TheGamer. https://www.thegamer.com/how-big-world-planet-universe-no-mans-sky/

Barriga, N. A. (2019). A Short Introduction to Procedural Content Generation Algorithms for Videogames. *International Journal on Artificial Intelligence Tools*, *28*(02), 1930001. https://doi.org/10.1142/s0218213019300011

Gladyshev, P., & Patel, A. (2004). Finite state machine approach to digital event reconstruction. *Digital Investigation*, *1*(2), 130–149. https://doi.org/10.1016/j.diin.2004.03.001

OSF. (n.d.). Osf.io. https://osf.io/preprints/osf/wn5y3

Sekhavat, Y. A. (2017). Behavior Trees for Computer Games. *International Journal on Artificial Intelligence Tools*, *26*(02), 1730001. https://doi.org/10.1142/s0218213017300010

Hemenover, S. H., & Bowman, N. D. (2018). Video games, emotion, and emotion regulation: expanding the scope. *Annals of the International Communication Association*, *42*(2), 125–143. https://doi.org/10.1080/23808985.2018.1442239

Recognition and Use of Emotions in Games. (n.d.). Ieeexplore.ieee.org. https://ieeexplore.ieee.org/abstract/document/8602898