Honors College Thesis
Submitted in partial fulfillment of the requirements for
graduation from the Honors College

Magnet-based Puzzle-platform Video Game

Michael Scandiffio
Lee Stemkoski
Neda Naseri
Zachary Pournazari
May 12, 2025

Abstract

This paper exists as a companion piece to the work I completed with the goal of creating a fun and unique video game that teaches magnetism. This process first entailed a four month period of study, research, and deliberation on subjects such as what game engine to use, the targeted format and style of play, and conceiving mechanics related to magnetism that can exist within those frameworks. A four month period of active development followed that saw me through the creation of a working demo, built in the free, open-source Godot Engine, that has succinctly met my intentions. The demo tasks the player with completing four consecutive puzzles using a device that fires "polarity projectiles" that alter the magnetic polarity of special cubes. The interactions between these cubes, simulating magnetism, are the core focus of the gameplay. Through development, I became deeply familiar with Godot's capabilities and workflow, and learned how to utilize them to achieve my goals. The result of this project is a first-person, puzzle-platformer genre video game demonstrating the unique mechanic I developed to simulate and teach the attraction and repulsion properties of magnetism.

Table of Contents

Introduction

The culmination of my Honors College Thesis Project has been the creation of a playable proof-of-concept demo of a video game produced in response to the question, “How can we create a unique and fun video game that teaches magnetism?”. This report will detail the process of development from the project’s inception in August 2024 to its completion in May 2025. The process includes my initial brainstorming and research phase and the transformation of fruits of that labor into the final product that this written piece accompanies. The game is of the puzzle-platformer-genre and is played from a first-person camera perspective. The control scheme is purposefully reminiscent of first-person shooter-style video games. The product, “Magnet-based Puzzle-platform Video Game”, was produced by myself under the guidance of Professor Lee Stemkoski in the Godot game engine.

In August of 2024, I first met with Professor Stemkoski, who served as my thesis advisor. At that time, I had a loose concept of the game I wished to produce. However, I knew that I wanted to draw inspiration from *Portal;* a 2007 video game developed and published by Valve Corporation. A first-person puzzle-platform video game, *Portal* equips its player with a unique device that it calls the “portal gun” in place of an actual weapon. Turning traditional first-person shooter-styled games on their head, *Portal* asks the player to wield this device and utilize it to complete a gauntlet of puzzles. These puzzles often center around physics concepts, such the conservation of momentum, and asking the player to use the device to solve them. The “portal gun” is a unique gameplay mechanic in that the player wields it and controls it like they would an actual weapon in most other similarly styled games, with the caveat that it fires portals instead of bullets. What this means in practice is that players are asked to fire two portals at walls, which connects them and allows the player and any other physics object to pass through one and

exit from the other as if the spaces were truly connected. The momentum of whatever passes through a portal is retained, allowing one to manipulate the positions of the portals in order to execute complex maneuvers. An example of such could be: building up a large amount of speed, changing portal positions, and then launching oneself out of the new portal to cross a large gap (Valve Corporation, 2007).

The video game *Portal* gives the player a familiar control scheme with an unfamiliar gameplay mechanic and asks them to run wild and figure out how to utilize it, while along the way dispensing simple physics lessons that force the player to think critically about all of the tools at their disposal. The game has been praised for its parallels to real world physics and motion problems as well as its ability to model them. This was essentially a summary of my discussion with Stemkoski in our initial meeting–I knew that I wanted to make something with a similar feel to *Portal,* but I also wanted to achieve something new. Our discussions identified that its educational nature and its genre are the two most critical aspects to what makes *Portal* so effective. In an educational setting, having a game capable of communicating high-level concepts to a beginner would be of high value, especially in the increasingly digital age. Lieberman (2010) addresses this topic, highlighting the value of children's comfortability with technology as a, "fundamental rationale for teaching with content-aligned games". As learning becomes more computer-oriented at every level of education, it is only natural that aspects of computing, like gaming, will seek to fill the niches that crop up. The two of us agreed that, in theory, *Portal* could be an entirely different game so long as it retains its design philosophy. The actual portal concept could be substituted for anything so long as the rest of its components are effective at utilizing it. That being decided, Stemkoski asked me to brainstorm what exactly my unique concept would be, if not portals.

Figuring out what the actual “thing” that I needed to come up with strengthened my ability to brainstorm and quelled a lot of the initial anxieties I had going into this project. What the subject matter of the game would be was absolutely crucial to being able to truly begin. Our initial meeting and discussion took inventory of my ambitions and helped boil down what I was ultimately looking to accomplish with the result of my project. While I had gone into the meeting and the thesis process as a whole very unsure of what my project would be or look like, I had left it knowing I only had one real decision to make. By mid-September, I had decided that the game would revolve around magnetism. I had landed on this idea fairly early but liking the potential I saw with it, I decided to move forward following approval and positive reception from my advisor. Having determined the content that my thesis project will concern itself with and what I will ultimately produce, I was asked to seek out a faculty member from the Department of Mathematics and Computer Science and one from the Physics Department to serve as additional readers on my overall committee. In October, I added Professor Zachary Pournazari and Professor Neda Naseri to my thesis committee alongside Lee Stemkoski.

Planning and Research

Meeting biweekly on Friday mornings at 9:00 AM during the Fall 2024 semester, Professor Stemkoski led me through the natural planning process, which constituted the vast majority of the work I completed through December. With preliminary research, an actual idea to work off of, and my thesis committee constructed, the next area of focus was to decide upon a game engine. The two under my consideration were Unity and Godot. Drawing upon my own prior experience working with Unity, I knew that a major concern of mine was to not pick an engine that would take me too long to learn. A steep learning curve would limit the amount of time I could spend in active development, which I wanted to avoid given the hard deadlines

demanded by the thesis process. Previous endeavors with the Unity engine and an assessment of popular comparisons between Unity and Godot led me to decide upon the latter despite having zero experience developing in it.

The Godot game engine is free and open source. That fact alone boosted it above Unity as it enables frequent stable releases, vast community support, and zero licensing overhead, which would have been an issue had I been hard-set on that engine. The engine is also fully compatible with exporting to a diverse range of desktop, mobile, web, and VR platforms. It also fully supports both 2D and 3D development. This provides a great deal of flexibility before you've even written any code. Godot is built on a system of scenes, nodes, and scripts. A scene is a collection of nodes, and nodes consist of all of the assets and objects required for an individual level or game-scene (Godot Engine. (n.d.). *Nodes* section*)*. Nodes can consist of a diverse array of standard game development components, like lights, text, meshes, colliders, rays, and areas. Nodes have individual properties and can be given child nodes to further diversify their functions. Scripts hold the logic that nodes execute. Godot's scripting is standardly done in its proprietary GDScript language, though it can be modified to accept a variety of others. Godot's documentation describes GDScript as, "a high-level, object-oriented, imperative, and gradually typed programming language (Godot Engine. (n.d.). *GDScript reference* section*)*." It supports many of the standard features in a language, like conditional statements, looping, methods, and debugging, and was the language that I used for this project. Individual scenes can also be added to other scenes as nodes, for additional complexity. Put together, Godot's development tools and environment allow for very fine control over your game components. The workflow of the engine's scenes, nodes, and scripting systems encourage the reuse of assets as well as a fairly modular experience. The editor's UI allows for drag-and-drop

maneuvers in most all of its parts, so for example, assigning a script to a node is as simple as dragging that script on top of it. This nature of the engine lends itself much more to a beginner or hobbyist development mindset, which I found I benefited from greatly as so much of my process revolved around experimentation and discovery.

Upon this decision, Stemkoski provided me with Godot 4 tutorial resources, and through these guides, I ended up with a basic 2D platformer in Godot, created to help learn the fundamentals of the engine. One tutorial, published by the YouTube account "Brackeys", taught me how to interface with the Godot editor much as an overall project manager. It also walked me through certain engine-unique quirks, like setting a project's keybinds, changing scenes, structuring nodes, and importing textures and audio. While I may have the benefit of being a competent software engineer before going into this project, learning the environment-unique tools when beginning a brand new project is absolutely essential. Given Godot's somewhat unique node system, my time at this step was well spent. Completing this tutorial served as my own introduction to Godot and left me with a glimpse of everything a full project in the engine entails. Thes process allowed me to start envisioning what my game might really look like having seen many of the tools at my disposal.

Once again drawing inspiration from *Portal*, I had already decided that the game would play like a first-person shooter, and that the game would need to be 3D to facilitate that. I also had decided that the game would, in some way, revolve around magnetism. Working off of how *Portal* is centered around a concept that it gives the player the tool to directly manipulate, I decided my game would do the same. Conceptualizing this, I envisioned a "magnet gun" that can fire positive or negative charges (later changed to polarities, though positive and negative is still used as the visual metaphor in-game), at specialized "magnet boxes". These magnet boxes

are monopoles, a purely hypothetical concept of a magnet with only one polarity, and not both. The simulated monopoles would still interact with others as one would expect two halves of a magnet to, if separating them was possible. It should be noted that separating a traditional magnet at its mid-point would just produce two smaller magnets, each with a north and south end. The magnet gun can manipulate the polarities of these monopoles to influence their positions and how they interact with each other. The player interacting with the monopoles through the device would be the main focus of the game, with additional platforming being more of a sub-focus. The purpose of my work from August through the end of the semester was to compile both the knowledge needed and the decisions needed to facilitate game development. With these decisions on the game engine, genre, subject matter, and core gameplay mechanics set in stone, I began active development in January, meeting weekly with my advisor to discuss each week of development.

Character Controller

Development in Godot begins with creating a way for the user to interface with the game itself. This could be something as simple as recognizing cursor or mouse input, but in my case for a 3D game, requires creating a character controller that can interact with collision. For this process I referenced two tutorials on YouTube published by the account, “LegionGames” while modifying the guidelines it offers for creating a character controller to suit the needs of my game.

My character controller is composed of 4 child-nodes of a CharacterBody3D node that allows the user to fully interact with the world. The engine’s documentation defines this as, “a specialized class for physics bodies that are meant to be user-controlled.” It includes explicit methods for concepts like floor detection and movement to optimize it for that purpose. The children of this node consist of a MeshInstance3D, CollisonShape3D, Decal, and Node3D. The

first three supply the mesh, collision, and a simple drop shadow for the character. The drop shadow exists to assist the player in positioning themselves through platforming. Node3D is a special node type holds no unique function, but can be used to group other nodes under it. For the character controller, the Node3D holds an Area3D and a Camera3D node. Areas project a range outwards from their center and can detect when others intersect with them; I utilize this to handle tutorial text that must appear on screen when the player enters a certain area. The Camera3D node manages the viewport that the game is displayed through, and it has a child node consisting of my "magnet gun" scene (which handles the mesh and animations for the device) . Child nodes inherit properties like position and rotation from their parent, so attaching the "magnet gun" scene as a child of the Camera3D ensures that it follows camera movement–that way it never leaves the screen and can always be used by the player for aiming.

Attached to the CharacterBody3D node is a script; player.gd. The player script is responsible for handling player input and translating it into in-game actions via interacting with the nodes I previously described. This script relies primarily upon the "_physics_process()" method, which is integral to how the GDScript environment runs code. A script generally is initialized with a "_ready()" and a "_process()" method. Code under a "ready" method will execute once when the script is initialized and then never again. Code under a "process" method will execute every frame of gameplay until the game is exited. The "_physics_process()" method is specialized to be called at a fixed rate that aligns with the engine's physics calculations rather than its framerate, which ensures the code inside of it will run consistently and not vary due to differences in performance or hardware. Understanding and utilizing these paradigms are vital to implementing any scripted feature in Godot. A complete collection of the scripts I wrote for this project are included in the Appendix.

In Godot, It is imperative that we handle controls under a “process” method so that inputs can constantly be accepted–otherwise we would only ever check for them once and the game wouldn’t be interactive. The script defines certain constants; like WALK_SPEED, SPRINT_SPEED, JUMP_VELOCITY, and leverages them to compute movement based on user inputs. For example, the effective speed of the player is swapped between WALK_SPEED and SPRINT_SPEED on the holding of the key assigned to that action. The script also utilizes events to pick up on user input, like “InputEventMouseMotion”, which waits for cursor input, and when detected, translates that input into up-down and left-right movement on the Camera3D node (and it’s child “magnet gun” scene node), thus simulating the ability to look around and aim the device that the player holds fluidly in one set of motions.

Putting together the character controller and a simple level scene, allowed me to run the game and get a feel for what playing it might look like. This first portion of development created the framework that I would build the rest of the game off of. This process continued with creating the polarity-changing projectiles that the gun would fire and adding action inputs to allow this to occur. I assigned the north polarity to left-click and the south polarity to right-click, intentionally paralleling *Portal*’s control scheme that places the controls for firing the two portals to these same buttons. When the player.gd script receives one of these inputs, it creates a projectile at the barrel of the gun that has its own script attached. This bullet.gd script manages what must occur when the projectile collides with specific objects. In this case, this is modifying the polarity of a monopole that it collides with to be “more north” given a north polarity particle, and “more south” given the opposite. Projectile behavior is controlled using RayCast3D class nodes, which perform raycasting. Raycasting is a concept where a “ray” is cast from a point in 3D space and we check what it intersects (Godot Engine. (n.d.). *Ray-casting* section*)*. The

behavior here uses one ray to manage the path of the bullet and the direction it should be fired in. It uses another to detect what type of object is ahead of the bullet before it collides, allowing us to perform actions based on that behavior.

Creating the Gameplay Mechanic

What I now needed to create was the core of the game–interacting with the monopoles as well as their interactions with each other. This took shape in the form of my "mag_box" scene. Scenes in Godot can take on a variety of forms and in this case, they take the form of how we would typically treat instantiable classes or "objects" in object-oriented programming. The goal of the mag_box scene needed to be an object where multiple could exist at one time, all could have unique values in their properties, and all could interact with each other and the player fluidly. All were necessary to give the monopole mechanic enough use-cases to center a game around. The head node in the scene is of the RigidBody3D class, providing full physics simulations for the object and the functions that come with it. As with the PlayerCharacter3D, the mesh, collision, and object-to-object interactions are handled by MeshInstance3D, CollisionShape3D, and Area3D nodes. I decided to go with a simple cube for the mesh, due to the right angles allowing for predictable collisions. The majority of functionality is handled by the head node's mag_box.gd script. This script acts essentially as the class definition and defines the instance variables of the object. Prefacing a variable declaration with the "@export" annotation exposes it to the editor and makes it unique to that instance of the scene. The mag_box is managed by five variables–polarity; a float, mobile; a boolean, range; a float, mag_sens; a float, and size; a Vector3. Polarity aptly represents the polarity of the box, with negative one being fully south and positive one being fully north. Mobile determines if the box can move or not. Range controls the area around the box that other boxes can be affected in.

The “mag_sens”, or “magnet sensitivity” property controls how much the polarity is changed by a single projectile. The size property exists so that one value can control the size of both the collision and mesh of the box.

Creating the functionality for these objects involved both the visual half and a physics half. I wanted to convey the polarity to the user such that very little mental processing is required to understand and predict how two monopoles might interact with each other. For that reason, I chose to modify the color of its material based on the polarity property’s value. This means a value of negative one corresponds to a box that is “fully south” and a deep blue color while positive one corresponds to “fully north” and bright red color. Any values between negative one and one interpolate a color between the two ends of the spectrum based on the value. With this set up, I now moved to the physics component of the work.

Working off the age-old logic of “opposites attract”, I deliberated on how best to achieve this mechanic with Professor Stemokoski. My intention for how it would operate is that when a box is inside the range of another, if both polarities are not zero, movement should occur. That movement should only affect boxes that are mobile and have a range less than that of the box who’s range it is inside of. The range value acts essentially as a representation of the strength of magnetism. The type of movement that should occur, the weaker box moving either towards or away from the stronger one, is determined by their polarities. If one box’s polarity is a positive value and the other is a negative value, they move towards each other and otherwise, they move away from each other until the smaller box has exited the larger one’s range.

Due to the reliance of this mechanic on the range (the Area3D node) of the object, the _physics_process() code that manages this movement exists in the detection.gd script attached to the box’s Area3D node. This allows us to utilize specific checks on the status of areas. On every

physics calculation, every mag_box on the current scene checks if there are other mag_box objects within its range. If there aren't, nothing happens, but if there are, that mag_box will then check the movement for each one within its range. Our discussion laid out these points and concerns of mine, which we then transformed into a real plan. The plan became to handle movement by first computing a "polarity_modifier" as the product of the polarity of the box doing the repelling or attracting and that of the one it is moving. Utilizing basic laws of multiplication, if both values are negative or both values are positive, meaning the polarities were both northern or both southern, we are returned a positive polarity_modifier value. Otherwise, we are returned a negative polarity_modifer value. The difference in sign allows us to quickly determine what type of magnetic movement must occur based on the polarities of any two monopoles. We then compute the direction of a vector between the two boxes by taking the difference between their positions, the distance, and dividing it by the length of that vector. This gets us the normalized direction, which can be multiplied together with the polarity_modifier, delta, and a constant of 5 into a full movement vector that can be applied to the position of the moving box, thereby moving it.

Implementing that gave an almost completely functional implementation of the mechanic, barring a few hiccups that came up throughout development. The one that took up the most amount of my time came in the caveat of executing the movement code in a _physics_process() method. Since when an object has met the right conditions for movement (box is mobile, inside of another's range, has less range than that of what is inside of, and has a polarity not equal to 0), we constantly do that movement until one of the above conditions are no longer met. This created collision issues when the boxes are repelled into walls or attracted towards each other, as there was nothing in place to stop movement from occurring when

conditions are right if it wasn't actually necessary. When the issue occurred, the boxes would be repeatedly shoved into walls or other boxes with undesired behavior and movement. I half-solved this problem by implementing an additional movement condition that is true if the mag_box that needs to be moved has just collided with another mag_box. If it has, movement cannot occur. This stops mag_box movement as soon as they meet after being attracted. Every projectile collision on a mag_box briefly sets this condition to false, and if it is not immediately retriggered, it will remain that way and movement will occur when the other conditions are met and otherwise, it will just retrigger and prevent further unnecessary movement. Unfortunately, I was only able to devise a solution for box-to-box interactions and not box-to-wall interactions. Professor Stemkoski and I deliberated on an idea of separating the magnetic movement into separate attraction and repulsion functions, rather than tying everything up in one, to more finely tune specific conditions for each type of movement. In the end, I didn't move ahead with that idea as there was not enough time at the point to implement it without returning a far less complete end-product than I have now. Given extended development time, this area is where I would center my focus first in terms of worthwhile improvements to the work I was able to include in this current timeframe.

Level Design

Transitioning from feature development to level design, I quickly came upon a new problem. When designing levels with custom objects, you end up in circumstances where you need to visualize the capabilities of those objects on-the-fly. Implementing the visuals for the mag_box objects is incredibly helpful in-game, but is useless to me during development. If a level I am designing needs to use mag_box objects with a modified size, I can't position it correctly since the editor is forced to render it at its default 1-by-1-by-1 dimensions. While this

can be resolved by just running the game, it can be better done with developer tools. Godot implements this concept by allowing code to run inside of the editor if a script is headed with the "@tool" annotation (Godot Engine. (n.d.). *Running code in the editor* section*)*. Adding it without any additional edits will force the entire script to run, which can lead to unintended code effects. To combat this, I headed each method in the mag_box.gd script with a conditional statement that checks "Engine.is_editor_hint()" and includes a "pass" statement at the end of it. If the code is executed within the engine, before any method's actual content is run, the engine will enter this conditional statement and hit a "pass", which immediately exits the method. Any code that I actually want to run inside of the editor, like coloring and resizing the objects based on their polarity and size instance variables, can be placed safely inside of the conditional statements for in-editor execution. With the assurance that my developer tool code always runs separately from actual game code, I could leverage "@tool" to solve my problem and speed up level design by cutting down on the amount of times I need to re-run the game to test small changes.

The four levels, or puzzles, that I designed for this demo were each made with a concept or mechanic to teach to the player in mind. I felt it important to not overwhelm the player with new information, so the first level doesn't include the magnetism mechanics. It is designed such that in order to complete it, the player must learn the controls. You begin facing a wall that you can walk towards, and must learn to strafe and fully utilize 3D movement to move past. The player must then turn 90 degrees to face down the right way in a corridor, and jump up to a higher surface. They must combine these skills of moving and jumping to cross a gap and continue on to the next level. Each chunk builds upon the last skill.

The second stage continues this by presenting the player with two monopole objects–one positioned out of the player's reach high above another that blocks the player's path through the corridor. The player is confronted with the two monopoles, each repelling the other and blocking the path, and is given a brief description of how to fire the polarity projectiles and what the monopole objects are. This puzzle is designed to force the player to understand how to attract two monopoles together. Since both monopoles were initialized with a polarity value of negative one, the player will either try left-clicking first, firing a "north" projectile and completing the puzzle, or they will try right-clicking first, which won't help them, and implore them to try their other option. Designing a level this way–in a manner that implicitly guides the player to make a discovery on their own–is philosophy lifted directly from *Portal*. The game's developer commentary on its third puzzle makes note of this, stating, "We introduced a mandatory pause in the action - what we call a gate - to help ensure that players stop and notice the portal gun making a blue portal (Valve Corporation, 2007)." By giving the player a barrier that they can only move if they understand the magnetism mechanic forces there to be some amount of recognition of the concept by the player. Level design can guide actions just as easily as spoken directions.

The further two puzzles were designed to advance the player's ability by putting them in scenarios where they need to combine two previously taught skills. Up until this point, the player has only been asked to practice one skill to resolve the current situation that they are in. The third level presents the player with a wall that is too tall for them to climb, as well as a small monopole and a large one. Both begin at a neutral polarity, so the player must understand that they themselves are capable of applying both types of polarities to make the monopoles attract towards each other. Afterwards, the player must utilize platforming skills to ascend the wall

using the objects they moved. The following stage introduces another new mechanic–buttons–in the form of red squares that trigger white doors to open when a monopole reaches it. The mechanic is drip fed at first, by providing a puzzle identical to the second one, but without the monopole suspended above. This offers a familiar solution to a new problem, ensuring that the mechanic is picked up. The white door opens to a second room with a puzzle reflecting the third one, but with the addition of a button at one end of the room and a door blocking the way atop the tall platform. The player must combine the earlier solution with the one from the preceding puzzle in order to progress. Both sets of puzzles allow the player to develop new skills for themselves rather than the game explicitly teaching them. That aspect of *Portal*'s design is what I most frequently attribute to the game's approachability, and it was a core focus of mine in choosing what to include in the four puzzles.

Live Demonstrations

In each puzzle that I designed, the player was required to resolve a solution based on environmental cues as well as building off of what they learned from previous puzzles. I kept this concept in mind as I developed each level as my goal is to create a game that almost anyone could comfortably pick up and play without outside interference. Aside from the additions of some on-screen text to describe the actual controls and dispense simple hints, the actual gameplay informs the player. I was delighted to see most of these intentions carry through when I had the opportunity to have unfamiliar players blindly test the project at Adelphi University's 22nd Annual Scholarships and Creative Works Conference.

Over the course of my hour at the conference, I had five players ask if they could play the demo I had created while I observed. I offered as little intervention as possible, only intervening when players struggled with core aspects of the game, like utilizing the controller, that would

have severely impacted their ability to experience the game. All five players were able to complete each of the four puzzles I created. Across each session, players most commonly struggled with crossing the gaps I constructed to teach platforming in the first puzzle, and following that, the final puzzle slowed three of my players down. Based on my conversations and observations, players struggled with the platforming due to the precision that landing each jump required. The final puzzle requires the combination of almost every skill the game has asked the player to use thus far, so it didn't surprise me that some struggled. Regardless, each player was still able to complete the final puzzle. My experience at this conference highlighted both an immediate change I could make–widening the small platforms to make it easier to land on them–as well as a potential structural issue that, given extended development time, may require attention. I withheld from modifying the last puzzle any further, as I feel it was still effective at making the player think and come up with their own solution. In a full-length project, what changes I could make to the fourth puzzle could be more apparent if it has to contend with the content that follows as well as what precedes it.

Final Product and Conclusions

Each stage of development, as I've described in this report, was conducted consecutively over the course of 3 months of active development from late-January to late-April of 2025. I determined what I would work on next with the assistance of my advisor, Lee Stemkoski, who guided me through development piece by piece, with each step building on the last. Spending time familiarizing myself with the capabilities of different parts of the engine, spread out over 12 weeks of development, allowed me to grow confident with each tool and made me able to apply them better in the future. I felt this most with the discovery of the "@tool" annotation. Being able to run code in the editor as the intrinsic method for designing developer tools helped me

further my puzzle designs and allowed me to cut down on the time it took to prototype new puzzles and mechanics. The development process comprised two essential stages; first, initial setup alongside programming the monopole gameplay mechanic from late-January through the middle of March. Following that, development progressed into the tweaking and puzzle design phase that concluded with the completion of this project in April.

My goal from the onset was to finish development having produced a working demo that showcases my idea and teaches the player about basic concepts of magnetism. The game I created does just that, asking the player to solve puzzles directly tied to this concept that, without properly understanding the simulated magnetic mechanics at play, they would not be able to effectively do. The process of creating this game gave me an extreme fondness for the Godot engine as a tool for game development and experimentation. While I have past experience in game design through schooling and personal hobbyism, I have never undertaken a long term game development process such as this, focused on a single defined end result. The making of this project has left me with a newfound confidence in game design as well as in the evolution and cultivation of my own ideas into an even greater final product.

References

Brackeys. (2024, April 28). How to make a Video Game - Godot Beginner Tutorial. YouTube. https://www.youtube.com/watch?v=LOhfqjmasi0

Godot Engine. (n.d.). *Godot Engine documentation*. https://https://docs.godotengine.org/en/stable/

Godot Engine. (n.d.). *CharacterBody3D*. Godot Docs. https://docs.godotengine.org/en/stable/classes/class_characterbody3d.html

Godot Engine. (n.d.). *GDScript reference*. Godot Docs. https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html

Godot Engine. (n.d.). *Node*. Godot Docs. https://docs.godotengine.org/en/stable/classes/class_node.html

Godot Engine. (n.d.). *Ray-casting*. Godot Docs. https://docs.godotengine.org/en/stable/tutorials/physics/ray-casting.html

Godot Engine. (n.d.). *Running code in the editor*. Godot Docs. https://docs.godotengine.org/en/stable/tutorials/plugins/running_code_in_the_editor.html

LegionGames. (2023, May 19). Juiced Up First Person Character Controller Tutorial - Godot 3D FPS. YouTube. https://www.youtube.com/watch?v=A3HLeyaBCq4&list=PLQZiuyZoMHcgqP-ERsVE4x4JSFojLdcBZ&index=1

LegionGames. (2023, June 22). Complete 3D Shooting Mechanics - Godot 4 FPS Tutorial. YouTube. https://www.youtube.com/watch?v=6bbPHsB9TtI&list=PLQZiuyZoMHcgqP-ERsVE4x4JSFojLdcBZ&index=4

Lieberman, M. (2010). Four Ways to Teach with Video Games | Currents in Electronic Literacy. Currents.dwrl.utexas.edu. https://currents.dwrl.utexas.edu/2010/lieberman_four-ways-to-teach-with-video-games.html

Valve Corporation. (2007). Portal [Video game]. Valve.

Appendix: GDScript

bullet.gd

```
# SCRIPT FOR PROJECTILE BEHAVIOR

extends Node3D

const SPEED = 40.0 # Default speed
var velocity = Vector3(0, 0, -SPEED)
var hit: bool = false; # Set to true after a projectile collides; prevents
duplicate function calls

@onready var mesh = $MeshInstance3D # Reference to the mesh
@onready var ray = $RayCast3D # Reference to the ray; needed to detect a
collision
@onready var particles = $GPUParticles3D # Reference to particle emitter

# Called when the node enters the scene tree for the first time.
func _ready() -> void:
      pass # Replace with function body.

# Called every frame. 'delta' is the elapsed time since the previous frame.
func _process(delta: float) -> void:
      position += transform.basis * velocity * delta # Attempt movement each
frame
      if ray.is_colliding(): # Further actions need only performed if a
collision was detected
            # Visuals for projectile colliding
            mesh.visible = false
            particles.emitting = true

            if ray.get_collider().is_in_group("mag") && !hit: #If projectile
collides with a mag_box AND a collision is not currently active
                  if (is_in_group("Pos")): # If projectile was 'Pos'
                        ray.get_collider().pos()
                        hit = true;
                  else:
                        ray.get_collider().neg() # If projectile was 'Neg'
                        hit = true;

                  await get_tree().create_timer(1.0).timeout
                  queue_free()
            elif(ray.get_collider().is_in_group("level_change") && !hit): #If
projectile collides with a level_change object
                  print("Loading level #", ray.get_collider().level)
                  match ray.get_collider().level: # Switch-case for
level-changing debug
                        0:

get_tree().change_scene_to_file("res://Scenes/debug.tscn")
                        1:

get_tree().change_scene_to_file("res://Scenes/world1.tscn")
```

```
                        2:

get_tree().change_scene_to_file("res://Scenes/world2.tscn")
                        3:

get_tree().change_scene_to_file("res://Scenes/world3.tscn")
                        4:

get_tree().change_scene_to_file("res://Scenes/world4.tscn")
                        5:

get_tree().change_scene_to_file("res://Scenes/world5.tscn")



func _on_timer_timeout():
      queue_free() # Removes bullet after timer ends
```

button_detection.gd

```
# SCRIPT FOR BUTTON ACTIVATION DETECTION

extends Area3D

@onready var door0 = $"../MapEnvironment/CSGBox3D4" # Reference to a 1st door
on a level
@onready var door1 = $"../MapEnvironment/CSGBox3D6" # Reference to a 2nd door
on a level
@export var button = 0; # Instance variable to track what door this button
should open

func _ready():
      # Connects signals from Area3D to functions
      body_entered.connect(_on_body_entered)
      body_exited.connect(_on_body_exited)

func _physics_process(delta: float) -> void:
      pass

func _on_body_entered(body: Node):
      # Disables doors when a mag_box enters a button Area3D
      if body.is_in_group("mag"):
            print(self.name, " says: Object entered button range: ", body.name)
            match button:
                  0:
                        door0.visible = false
                        door0.collision_layer = 0
                        door0.collision_mask = 0
                  1:
                        door1.visible = false
                        door1.collision_layer = 0
                        door1.collision_mask = 0
```

```
func _on_body_exited(body: Node):
	# Enables doors when mag_boxes leave a button Area3D
	if body.is_in_group("mag"):
		print(self.name, " says: Object exited button range: ", body.name)
		match button:
			0:
				door0.visible = true
				door0.collision_layer = 1
				door0.collision_mask = 1
			1:
				door1.visible = true
				door1.collision_layer = 1
				door1.collision_mask = 1

		#print(body.position)
```

detection.gd

```
# SCRIPT FOR DETECTING MAG_BOX RANGES

extends Area3D

@onready var mag_box = $".."; # Reference to the mag_box responsible
@onready var collision = $"../Collision" # Reference to collision
@onready var collision_detection = $"../CollisionDetection" # Reference to
collision Area3D

var is_overlapping = false # Cube can only move if they aren't colliding or
intersecting in anyway

func _ready():
	# Connects signals from Area3Ds to functions, for debugging
	body_entered.connect(_on_body_entered_range)
	body_exited.connect(_on_body_exited_range)
	collision_detection.body_entered.connect(_on_body_entered_collision)
	collision_detection.body_exited.connect(_on_body_exited_collision)

func _physics_process(delta: float) -> void:
	var bodies = self.get_overlapping_bodies(); # Get all bodies inside the
Area3D
	var mag_bodies = [];

	# Separate out mag_boxes into special array
	for body in bodies:
		if body.is_in_group("mag") && body != mag_box:
			mag_bodies.append(body)

	if (mag_bodies): # If at least one mag_box was found
		for body in mag_bodies: # For each one found, assess movement to be
made
			if distance(mag_box, body) > 0.0 :  # Boxes are not at the
exact same position
```

```
                        if (mag_box.range > body.range): # If the box running
this check is the stronger magnet, then do movement
                                var polarity_modifier = mag_box.polarity *
body.polarity
                                if (polarity_modifier < 0 &&
!body.detection.is_overlapping): # ATTRACTION
                                        var movement = polarity_modifier *
direction(mag_box, body) * delta * 5
                                        body.position += movement
                                elif (polarity_modifier > 0 &&
!body.detection.is_overlapping): # REPULSION
                                        var movement = polarity_modifier *
direction(mag_box, body) * delta * 5
                                        body.position += movement

# FOR DEBUGGING PURPOSES
func _on_body_entered_range(body: Node):
      if body.is_in_group("mag") and body != mag_box:
            print(mag_box.name, " says: Object entered range: ", body.name)

func _on_body_exited_range(body: Node):
      if body.is_in_group("mag") and body != mag_box:
            print(mag_box.name, " says: Object exited range: ", body.name)

func _on_body_entered_collision(body: Node):
      if body.is_in_group("mag") and body != mag_box and mag_box.mobile !=
false:
            is_overlapping = true;
            print(mag_box.name, " says: Object entered collision: ", body.name)

func _on_body_exited_collision(body: Node):
      if body.is_in_group("mag") and body != mag_box and mag_box.mobile !=
false:
            is_overlapping = false;
            print(mag_box.name, " says: Object exited collision: ", body.name)

# Returns the direction between two bodies as a Vector3
func direction(body1: Node, body2: Node) -> Vector3:
      var position1 = body1.global_transform.origin
      var position2 = body2.global_transform.origin
      var direction = position2 - position1
      direction = direction / direction.length()
      return direction

# Returns the distance between two bodies as a Vector3
func distance(body1: Node, body2: Node) -> float:
      var position1 = body1.global_transform.origin
      var position2 = body2.global_transform.origin
      var distance = position2 - position1
      return distance.length()

# Method triggered to unstuck cubes and ensure proper movement logic
func action():
      is_overlapping = false
```

level_change.gd

```
# SCRIPT TO DEFINE LEVEL_CHANGE OBJECT

extends CSGBox3D
@export var level = 0 # Instance variable, stores the level this object should
warp the player to
```

local_player_detection.gd

```
# SCRIPT FOR DETECTING OBJECTS IN FRONT OF PLAYER,
# USED FOR DISPLAYING TUTORIAL TEXT

extends Area3D

@onready var player = $"../.."

func _ready():
	# Signals to connect Area3D to functions
	area_entered.connect(_on_area_entered)
	area_exited.connect(_on_area_exited)

func _physics_process(delta: float) -> void:
	var bodies = self.get_overlapping_bodies(); # Get all bodies overlapping
with player detection Area3D
	var text_bodies = [];
	for body in bodies: # Separate text
		if body.is_in_group("text") && body != player:
			text_bodies.append(body)

# Calls displayText() when player enters text region
func _on_area_entered(area: Node):
	if area.is_in_group("text") and area != player:
		area.displayText();
		print(player.name, " says: Object entered text reading range: ",
area.name)

# Calls hideText() when player exits text region
func _on_area_exited(area: Node):
	if area.is_in_group("text") and area != player:
		area.hideText();
		print(player.name, " says: Object exited text reading range: ",
area.name)
```

mag_box.gd

```
# MAIN SCRIPT FOR MAG_BOX OBJECTS

@tool # Allows script to run in-engine
```

```
extends RigidBody3D

@onready var model = $Model # Reference to mesh
@onready var collision = $Collision # Reference to collision
@onready var collision_detection = $CollisionDetection # Reference to collision
Area
@onready var detection = $Detection # Reference to range Area
@onready var det_range = $Detection/Range # Reference to Area shape
@onready var det_model = $Detection/RangeModel # Reference to Area model -- for
debugging

@export var polarity: float = 0.0: # Instance variable -- mag_box's current
polarity.  Includes setter/getter for @tool use
      get:
            return polarity;
      set(value):
            polarity = value;

@export var mobile: bool = true; # If true, box can move
@export var range: float = 0.5: # Defines range that box is magnetic for
      get:
            return range;
      set(value):
            range = value;
            update_range()
@export var mag_sens: float = 0.2; # Defines the scale at which polarity can
change

# Separating out the size to an instance variable was needed as the collision
and mesh must be changed at the same time.
# This automates that change.
@export var size: Vector3 = Vector3(1, 1, 1): # Defines the size of the object.
Includes setter/getter for @ tool use.
      get:
            return size;
      set(value):
            size = value
            update_size()

signal polarity_changed(new_polarity, mag_box); # Releases signal when polarity
changes

var pos_color = Color(3, 0, 0); # Stock color for most positive polarity
var neg_color = Color(0, 0, 3); # Stock color for most negative polarity

var bodies;
var mag_bodies = [];

# Inside the _ready() function
func _ready() -> void:
      # If in editor, run this and nothing else.
      if Engine.is_editor_hint():
            update_size()
            update_range()
            return

      # Set all functional properties on init
```

```
	mass = range
	det_range.shape.radius = range
	update_range()
	update_polarity()
	update_size()

# Called every frame. 'delta' is the elapsed time since the previous frame.
func _process(delta: float) -> void:
	if Engine.is_editor_hint():
		return
	pass

# Called to update mesh and collision at once
func update_size():
	if is_inside_tree():
		if model and model.mesh is BoxMesh:
			model.scale = size;

	if collision and collision.shape is BoxShape3D:
		collision.scale = size;

# Called to update range
func update_range():
	if det_model and det_model.mesh is SphereMesh:
		det_model.scale = Vector3(range, range, range)

# Called to update polarity
func update_polarity():
	if Engine.is_editor_hint():
		if model.material_override == null:
			model.material_override = StandardMaterial3D.new()
		model.material_override.albedo_color = neg_color.lerp(pos_color,
(polarity + 1)/2) # Interpolates a color based on polarity
		return

	emit_signal("polarity_changed", polarity, self);
	if model.material_override == null:
		model.material_override = StandardMaterial3D.new()
	model.material_override.albedo_color = neg_color.lerp(pos_color,
(polarity + 1)/2)  # Interpolates a color based on polarity
	#detection.action();

	bodies = detection.get_overlapping_bodies();
	if (bodies):
		for body in bodies:
			if body.is_in_group("mag") && body != self:  # Separate out
mag_boxes
				mag_bodies.append(body)

	# Hit when polarity is changed able to interact with other mag_boxes
	if (mag_bodies):
		for body in mag_bodies:
			print(self.name, " says: I am inside of: ", body.name)
			body.detection.action() # Unstucks box to resume movement

# Handles positive polarity change
func pos() -> void:
```

```
	if Engine.is_editor_hint():
		return

	polarity += mag_sens
	polarity = clamp(polarity, -1.0, 1.0);
	update_polarity();

# Handles negative polarity change
func neg() -> void:
	if Engine.is_editor_hint():
		return

	polarity -= mag_sens
	polarity = clamp(polarity, -1.0, 1.0);
	update_polarity();
```

mag_interaction.gd

```
# SCRIPT FOR DETECTING MAG POLARITY CHANGES
# NODE HOLDS ALL MAG BOXES IN A LEVEL

extends Node

var rigid_bodies;

func _ready():
	rigid_bodies = get_children() # Gets all mag_boxes and connects a signal
to watch for polarity changes
	for mag_box in rigid_bodies:
		mag_box.polarity_changed.connect(_on_polarity_changed)

# Notifies mag_boxes to briefly unstuck themselves to allow for movement
following a polarity change
# Otherwise, movement can't be updated following a polarity change
func _on_polarity_changed(new_polarity: float, mag_box: RigidBody3D):
	print("Polarity changed for ", mag_box.name, " to: ", new_polarity);
	print("Range for ", mag_box.name, " is: ",
mag_box.get_child(3).get_child(0).shape.radius)
	mag_box.get_child(3).action()
```

player.gd

```
# SCRIPT FOR CHARACTER CONTROLLER
# MANAGES THE PLAYER
# DEFINES MOVEMENT PARAMETERS AND HANDLES SHOOTING

extends CharacterBody3D

var speed # represents current applied speed
```

```
# CONSTANTS
const WALK_SPEED = 2.0
const SPRINT_SPEED = 3.0
const JUMP_VELOCITY = 3.7
const SENSITIVITY = 0.003
var DEADZONE = 0.2

# VIEW BOBBING
const BOB_FREQ = 2.0
const BOB_AMP = 0.08
var t_bob = 0.0

# FOV
const BASE_FOV = 75.0
const FOV_CHANGE = 6.5

var PosBullet = load("res://Scenes/PosBullet.tscn") # Reference to Pos
projectile
var NegBullet = load("res://Scenes/NegBullet.tscn") # Reference to Neg
projectile
var instance

@onready var head = $Head # Reference to Player head
@onready var camera = $Head/Camera3D # Reference to camera
@onready var gun_anim = $Head/Camera3D/gun/RootNode/AnimationPlayer # Reference
to gun animator
@onready var gun_barrel = $Head/Camera3D/gun/RootNode/RayCast3D # Reference to
gun ray; used for firing projectiles

func _ready():
	Input.set_mouse_mode(Input.MOUSE_MODE_CAPTURED) # use cursor input

func _unhandled_input(event):
	if event is InputEventMouseMotion: # When cursor is moved, move camera
accordingly
		head.rotate_y(-event.relative.x * SENSITIVITY)
		camera.rotate_x(-event.relative.y * SENSITIVITY)
		camera.rotation.x = clamp(camera.rotation.x, deg_to_rad(-90),
deg_to_rad(60)) # Camera cannot be rotated past 90 degrees up and 60 down.

func _physics_process(delta: float) -> void:
	# Right-stick Camera Rotation.
	var look_x = Input.get_joy_axis(0, JOY_AXIS_RIGHT_X)
	var look_y = Input.get_joy_axis(0, JOY_AXIS_RIGHT_Y)
	if abs(look_x) > DEADZONE:
		head.rotate_y(-look_x * SENSITIVITY * 25)
	if abs(look_y) > DEADZONE:
		camera.rotate_x(-look_y * SENSITIVITY * 25)
		camera.rotation.x = clamp(camera.rotation.x, deg_to_rad(-90),
deg_to_rad(60))

	# Add the gravity.
	if not is_on_floor():
		velocity += get_gravity() * delta

	# Handle jump.
	if Input.is_action_just_pressed("jump") and is_on_floor():
```

```
            velocity.y = JUMP_VELOCITY

      # Handle sprint.
      if Input.is_action_pressed("sprint"):
            speed = SPRINT_SPEED
      else:
            speed = WALK_SPEED

      var input_dir := Input.get_vector("left", "right", "up", "down") #
Creates a vector for movement based off digital inputs
      var direction = (head.transform.basis * Vector3(input_dir.x, 0,
input_dir.y)).normalized()
      if is_on_floor(): # Allow movement when on floor
            # Move if direction is applied, otherwise slowdown.
            if direction:
                  velocity.x = direction.x * speed
                  velocity.z = direction.z * speed
            else:
                  velocity.x = lerp(velocity.x, direction.x * speed, delta *
7.0)
                  velocity.z = lerp(velocity.z, direction.z * speed, delta *
7.0)
      else: # Slow horizontal movement when falling
            velocity.x = lerp(velocity.x, direction.x * speed, delta * 2.0)
            velocity.z = lerp(velocity.z, direction.z * speed, delta * 2.0)

      # head bobbing
      t_bob += delta * velocity.length() * float(is_on_floor())
      camera.transform.origin = _headbob(t_bob)

      # FOV
      var velocity_clamped = clamp(velocity.length(), 0.5, SPRINT_SPEED * 2)
      var target_fov = BASE_FOV + FOV_CHANGE * velocity_clamped
      camera.fov = lerp(camera.fov, target_fov, delta * 8.0)

      # Shooting
      if Input.is_action_pressed("PosShoot"):
            # Fires 'Pos' projectile only if one is not currently being fire
            if !gun_anim.is_playing():
                  gun_anim.play("shoot")
                  instance = PosBullet.instantiate() # Creates the projectile
                  instance.position = gun_barrel.global_position
                  instance.transform.basis = gun_barrel.global_transform.basis
                  get_parent().add_child(instance)

      # Shooting
      if Input.is_action_pressed("NegShoot"):
            # Fires 'Neg' projectile only if one is not currently being fired
            if !gun_anim.is_playing():
                  gun_anim.play("shoot")
                  instance = NegBullet.instantiate()
                  instance.position = gun_barrel.global_position
                  instance.transform.basis = gun_barrel.global_transform.basis
                  get_parent().add_child(instance)

      move_and_slide() # Perform movement
```

```
func _headbob(time) -> Vector3: # Calculate headbob movement with sin/cos
	var pos = Vector3.ZERO
	pos.y = sin(time * BOB_FREQ) * BOB_AMP
	pos.x = cos(time * BOB_FREQ / 2) * BOB_AMP
	return pos
```

text_detection.gd

```
# SCRIPT FOR MANAGING TUTORIAL TEXT

extends Area3D

@onready var tutorial_text = $"../../UI/tutorial text"
@export var text_num = 0

func _ready():
	pass

func _physics_process(delta: float) -> void:
	pass

# CALLED FROM local_player_detection.gd
# Regions are structured in levels such that multiple displayText() calls can
occur before a hideText() one does.
# This allows for variable lengths/structures in sets of text that are sent to
the player
func displayText() -> void:
	match text_num:
			0:
				tutorial_text.text = "Push the left control stick
up to walk forward."
			1:
				tutorial_text.text = "Push the left control stick
up to walk forward." + "\nPush left and right to strafe."
			2:
				tutorial_text.text = "Push the left control stick
up to walk forward." + "\nPush left and right to strafe." + "\nPush down to
walk backward." + "\nRotate the right control stick slowly to look around."
			3:
				tutorial_text.text = "Push the left control stick
up to walk forward." + "\nPush left and right to strafe." + "\nPush down to
walk backward." + "\nRotate the right control stick slowly to look around." +
"\nPress A or B to jump."
			4:
				tutorial_text.text = "Use your triggers to shoot.
Aim and fire at the red block to advance to the next level."
			5:
				tutorial_text.text = "The blue cubes ahead of you
are magnets." + "\nTheir color represents their polarity: " +"\n - Blue is
south." + "\n - Red is north." + "\n - Purple means the magnet has no
polarity." + "\n\nThe device you hold can polarize these magnets." + "\nPress
the Left Trigger to fire a positive polarity." + "\nPress the Right Trigger to
```

```
fire a negative polarity." + "\n\nReach the end of the room.  Remember that
opposites attract."
                        6:
                              tutorial_text.text = "Please note that the
magnetic cubes are perfectly safe to touch."
                        7:
                              tutorial_text.text = "The red mark on the ceiling
is a button.  Activate it."
                        8:
                              tutorial_text.text = "Magnets can be re-polarized
as much you'd like."
                        9:
                              tutorial_text.text = "You have completed the
demo!  Thank you for playing! \nYou can select any of the four levels by
\nshooting at the cubes on your left. \nOtherwise, you can experiment with the
magnets here or exit the game.  \nPress ESCAPE to exit."
      tutorial_text.visible = true;

func hideText() -> void:
      match text_num:
                        3:
                              tutorial_text.visible = false;
                        4:
                              tutorial_text.visible = false;
                        5:
                              tutorial_text.visible = false;
                        6:
                              tutorial_text.visible = false;
                        7:
                              tutorial_text.visible = false;
                        8:
                              tutorial_text.visible = false;
```

world.gd

```
# SCRIPT FOR ROOT NODE OF EACH LEVEL

extends Node3D

@onready var crosshair = $UI/Crosshair

# Called when the node enters the scene tree for the first time.
func _ready() -> void:
      Engine.max_fps = 60 # lock framerate
      crosshair.position.x = get_viewport().size.x / 2 - 36 # position
crosshair
      crosshair.position.y = get_viewport().size.y / 2 - 36
      print("Loaded ", self.name)

# Handles quitting and restarting so that these functions are global,
regardless of if the player failed to load.
func _input(event):
      if event.is_action_pressed("quit"):
```

```
            get_tree().quit()
      if event.is_action_pressed("restart"):
            get_tree().reload_current_scene()


# Called every frame. 'delta' is the elapsed time since the previous frame.
func _process(delta: float) -> void:
      pass
```