PROJECT WISP

Nathan Warren-Acord
California State University Monterey Bay

CST 499 – Computer Science Capstone

June 16, 2020

Executive Summary

Project Wisp is an action combat system for a 2D, top-down game made in the Unity Game Engine. Actors and objects move in real time until combat is initiated with one or more enemies, upon which time will appear to stop and objects in the scene (actors, projectiles, etc.) will move only when the player moves (exiting combat ends the effect). The intention of this mechanic is to slow down the speed of gameplay, reducing stress on the player and promoting strategic decision-making. To give testers an objective, a simple level was designed and constructed, along with a minimal UI.

The target audience are all persons interested in combat-oriented adventure games. A high skill level in gaming is not required to enjoy the experience, but it is not aimed at being someone's first entry into the hobby. These end users make up the stakeholders and affected individuals, as well as anyone that took part in its development (including testers and myself). As this is a combat system, one's sensitivity to cartoon violence determines whether they suffer negative effects from the gameplay. Game loops may also feed addictive behavior, but such situations are not a concern at this early stage and known tactics to employ such game loops have been avoided.

Development started on the project with a different approach, one focused more on tactical gameplay using grid-based movement and turn-like actions. This design was scrapped due to the rigidness of the gameplay it produced. After a pivot to full-directional movement and real time action, development became much smoother and features became easier to implement.

The final implementation includes a playable character with three abilities, three enemies to face, a simple level and UI, as well as a unique take on combat with the time-stop mechanic. Combat abilities include melee and ranged attacks, and area of effects can be instantiated in the

scene. Enemies use a simple AI approach with behavior trees, along with a powerful A* pathfinding asset that provides flexibility in their basic behavior. A custom Event Manager enables the time-stop mechanic to function, with all actors and moveable objects in the scene stopping their behavior upon receiving the appropriate signal (and resuming their behavior in the same fashion).

Feedback from the initial round of testing was positive and constructive. Areas that need improvement are melee combat and enemy behavior, but the uniqueness of the system was applauded overall. One feature suggested in the feedback made it into the game: a camera that changes the environment to grayscale to indicate time has stopped.

In the future, Project Wisp will be refined and expanded, with the end goal to use it within a larger scale game. The experience it has afforded me and its inclusion in my resume are invaluable.

Table of Contents

I. Introduction / Background

Project name and description

Project Wisp is a tactical-action combat system for a 2D adventure game developed in the Unity Game Engine. The initial design had actors using grid-based movement for tactical gameplay, but this was altered during production due to problems with the implementation. Actors now move freely within the scene and projectiles can fly in any direction, moved by the physics engine. As was the case with the proposed design, action occurs in real-time until the player enters combat with one or more enemies. Upon combat activation, all actors and objects stop their real-time behavior and only act whenever the player takes an action (movement, attack, etc.). Once all enemy combatants have been defeated, combat ends and real time play recommences.

The purpose of the project was to prototype and refine this combat system for use in a larger scale game or as an asset for a software portfolio. Because the project involves advanced skill in game design, software engineering, and project management, it highlights many essential game development skills in one package.

Problem and Challenges

The core feature of Wisp is the unique combat-state behavior - this is also the core problem facing the design and completion of the project. Actors and objects shift between free-movement and stalled-movement on-the-fly using a messaging system and shared interfaces. The initial design had a turn-based system that would leverage off the grid-based movement – something noted as a challenge within the proposal. It was a challenge, not only because of the technical implementation and subsequent issues that arose, but also because of the effect it had on gameplay. Movement felt jerky and unresponsive, a problem that became more pronounced once the camera-follow feature was implemented (keeping the player in the middle of the camera view). Success and failure of the project depended on the execution of the combat system, but, with grid-based movement, the result was less than satisfactory.

Solution to Problem Encountered

Changing the way movement works allowed the project to become much more focused. Forcing actors and objects onto a grid, while also needing the physics engine for some movement calculations, was causing bugs and unwanted behavior. With a pivot to free movement, everything could be handled by the physics engine and most of the bugs that the grid system developed had gone away.

The free motion also provided another benefit, in that calculations did not need to be as precise. If an actor moves a thousandth of a unit further than they should have in a single frame, the player would not notice, and gameplay would not break down in a free moving system. In the old system, when a floating-point value did not reach the threshold it needed to within some coroutine, a domino-effect would occur where other objects and actors in the scene would be thrown off, making the game unplayable in such a state. There were fixes that worked, but, as soon as one would be patched, another would rear itself. Free motion solved all of that, making the gameplay and development much smoother.

II. Project Goals and Objectives

Proposed Objectives and Goals

Table 1

Goals and Objectives, Proposed

Goals	Objectives
Grid-based Movement	- Create standardization of units for movement - Create methods for converting between world and grid space - Use raycasting for obstruction detection
Enemies are fun and interesting to fight	- Research AI techniques, specifically Utility Theory - Implement Utility curves for AI decision making - Unique AI behavior created for each enemy - Tweak behavior values based on testing feedback
Seamless transition between combat and non-combat states	- Creation of global turn system - Actors and objects affected by combat state need shared Interface for turns - Design messaging system for state broadcasts
Project plays like a tiny game demo	- Design win condition that player can achieve - Design single level - (Stretch) Simple, quick tutorial / guide
Limit user confusion and amount of “outside” information they need	- Create simple and informative UI - If in-game guide isn’t implemented, create clear prompts / screen that displays controls and explains mechanics - Player informed of objectives in-game

Discussion on changes to objectives and goals

With the pivot on movement, the goals and objectives of the project needed to be altered and updated. Grid-based movement is no longer a goal, so that would be omitted, but the seamless transitions between combat states is still relevant. Enemy AI has also seen a scaling back of features, resulting in changes to the objectives for goals dealing with enemies and the overall game design. What is left is a much more focused project that leaves out some of the polish in favor of a streamlined proof-of-concept.

Post-pivot additions

Table 2

Post-pivot Goals and Objectives, Added

Goals	Objectives
Movement is fluid and responsive	- Actors are not confined to moving on tiles - Objects will need to utilize the physics engine - Account for collisions
Enemies serve their purpose in demonstrating the combat system	- Use of behavior trees for AI - Enemies use similar components - Diversity achieved through the fine-tuning of values and mixing of abilities
Project serves as proof-of-concept	- Win condition is not overly complicated - Level designed around combat - Scope of polish limited to simple effects that help with responsiveness

III. <u>Stakeholders and Community</u>

End Users

End users are the primary stakeholders, as well as testers and me. Developers that interact with the project for reasons other than entertainment, such as for inspiration or to give feedback, are included (essentially testers).

The hope is that all stakeholders involved get enjoyment out of the project. As this game is in the realm of entertainment and art, what the end users experience is subjective, including whether they feel as if they gained anything from it. An obvious loss for the stakeholders is time. If the experience is not something they wished for or enjoyed, they cannot regain the time spent with the project. As with the gains, losses are be subjective and dependent on each user's experience. The project has no monetary investment for stakeholders, so their intent is key (whether they are playing the game looking for enjoyment, to give feedback, etc.).

To mitigate the losses of the stakeholders, the nature of the project has been addressed upfront and, since time cannot be refunded, participation remains voluntary throughout. Honesty - all I can do is be honest with any participant and myself.

IV. Environmental Scan / Literature Review

Game: Stoneshard

A roguelike game that uses grid-based movement and tactical combat. What makes *Stoneshard* standout is its survival system and character customization (Suther, 2020). The game is brutal, however, giving very few moments of forgiveness when entering the wilds of its world (Wilde, 2020). Out of combat gameplay is run on a turn-based system, unlike Wisp, and it touts a small, simulated world.

Game: SUPERHOT

Combat-focused 3D game that labels itself as "the FPS where time moves only when you move" (SUPERHOT Team). The game centers itself around that core combat-mechanic, with its success credited to the simplicity and refinement in its execution (Hamilton, 2016).

Game: Darkest Dungeon

A turn-based, roguelike RPG that takes inspiration from Lovecraftian and gothic themes (Red Hook Studios, 2020). The gameplay is dark and devastating, but that is the allure. Where things go wrong for it is the repetition and tedium it eventually falls into - creating an experience that is no longer spooky, but exhausting (Gach, 2016).

Action vs. Tactical Combat

Players who prefer role playing games do not have a strong opinion on the question of action versus turn-based combat. Action combat is regarded as more reflexive and turn-based as more cognitive - the choice to pick one style over the other varies based on the player's mood (Hovermale, 2019). Regardless of genre, players don't want to be overwhelmed by systems (Johnson, 2009).

V. Feasibility Discussion

The initial environmental scan provided inspiration for the systems that would be combined in the project: tactical movement, a time-stop mechanic, and turn-based fighting. Each of these elements are feasible on their own, shown by the amount of game projects others have created with them.

Stoneshard uses pure grid movement, but only for actors and major objects, not for projectiles or attacks. The entire game is turn-based, even when the player is out of combat, meaning the grid-movement can take focus over anything that would need to use physics. Turns are also much more defined, with an entire action taking up a turn, instead of turns being time-based, as they were in Wisp. With time-based turns, there was a much higher chance for timings to get off and actions to get overwritten (such as an enemy moving, but then wants to attack before they finished their move turn). Choosing a pure grid or physics-based approach would allow for the system of choice to get the attention it needed.

Since movement already felt awkward and jerky, the contrasting smoothness that the physics engine promised was intriguing. Looking to take more inspiration from games like *SUPERHOT*, the concept needed to be translated into a 2D environment. *SUPERHOT* also has no switching of states, with movement slowed while the player is standing still regardless of combat scenario. This contrast between combat and noncombat gameplay is closer to the dynamic present in *Darkest Dungeon*, with the player's party switching from a simple exploration system, to a much more complex combat system when faced with enemies. To get the precise flow for the game, fine-tuning is needed and much of it, but a proof-of-concept has proven to be more than feasible under the time constraints.

VI. Design Requirements

Functional decomposition of the project

The specifications of the project revolve around the combat system since this is a focused prototype of that system. The player's abilities, the enemies they encounter, and the presentation (level, UI, etc.) makeup the core functions and focus of the prototype.

Table 3

Descriptions of Major Functions

Major Function	**Description**
Level Selection	The current state of the prototype includes one level, but the ability to have multiple levels is implemented. Based on the feedback from the first level, additional levels can be created to test and experiment with added features.
Combat Abilities	Players and enemies need multiple abilities to give the combat system depth.
Enemies Combatants	Players are presented with enemies to fight, each with unique behavior that promotes varied and tactical gameplay.
Combat Time-Stop	When entering combat, all actors and objects in the scene stop and only continue to move when the player is moving.
Full directional attacks and movement	Players can attack and move in any direction on a 2D plane.
Functional level	A complete, functional level is given to the player that showcases the other functions of the prototype. Additional elements, such as locking rooms and health pickups, are added to increase the enjoyment of the user and bolster the other systems.
Basic UI	Players are given basic information (health, status of abilities) in a

	clean and minimal interface. This serves a functional role in that players do not need to guess what their health is or when an ability is able to be used. While a mere prototype, allowing a user to concentrate on the systems at hand is still paramount.

Selection of design criterion

Prototypes do not need to meet the same quality assurance standards that products for release do. The intent here is to build the systems, as clean and fast as possible, to see if an idea works – proving the ideal concept in your mind within the unidealistic reality you must work in. Performance and reliability are not top priorities, with the systems needing to be functional and presentable enough that one might judge the merit of the idea.

Games tend to require a lot of code, and a lot of varied resources, which can translate to many hours of development. Having visual and audio assets that bring enough immersion for testing, but are not too polished, was desired as a middle ground. The GUI needed to portray enough information to allow concentration to remain on the systems being prototyped, but things such as tutorials and introductory prompts were omitted. Placing that information in a README file gives testers access to it, but also saves time spent implementing something more formal – this is important since things in the prototyping stage can change and pivot quick, as seen with this very project.

Final Deliverables

The final deliverable will be the latest test build of the project. This build will include a playable character, three unique enemies, and a completed level. Features of the enemies are pathfinding, simple AI routines, and the ability to be destroyed / spawned in the scene. The player will include three abilities to use: a melee attack, a ranged projectile attack, and a thrown-destructible item that leaves an area of effect on the ground. These features are demonstrated in a test level which includes rooms that lock the player in until enemies are destroyed and an end condition (either losing all health or defeating all enemies).

Combat, with its time-stop mechanic, gives the unique spin to the features above. Sending event messages to components based on game state, the included combat system only allows actors and objects to move when the player does whenever combat is active.

Feedback from a focus group, comprised of users within the target audience, is included. The data collected covers specific aspects of combat, such as pathfinding and time-stopping, as well the overall user-experience, including the game input, level of enjoyment, and responsiveness.

Approach / Methodology

As this is a solo project, time management and task prioritization were critical in its success. The steps below outline the general format that development took, though deviations were made where applicable.

1. Complete a rough sketch of the project’s mechanics and systems.
2. Create custom assets for testing, reducing the time searching for assets during development.
3. Create a barebones scene that includes the frame of future systems.

4. Choose one system or mechanic to focus on and prototype.
5. Perform research for recommended approaches and tools to be used in implementing that feature.
6. Code the feature as quickly as possible, focusing on a working implementation.
7. Assess the feature and its effect on the system it belongs to.
8. Refactor (if needed). If features are complete, go to step 9. Else, go back to step 4.
9. Perform stress testing and solo quality assurance, fixing priority issues as they are found.
10. Based on the results from step 9, compile a test plan for testers.
11. Compile a build and hand it out to volunteer testers.
12. Receive feedback and address concerns.
13. Fix bugs and errors discovered during the feedback stage. If time allows, redistribute build and go back to step 12.
14. Prepare and add extra polish for submission.

This structure was built to reflect an agile or spiral development cycle, repurposed for a solo project. A core feature would be implemented and then a refactor and reassessment of relevant systems would take place. The process worked well early in development, but much of the later polishing steps were reduced – having a feature-complete project was more important and there was less bug-fixing needed than expected.

Ethical Considerations

Major Ethical Concerns

Users may be sensitive to displays of violence, even in a fantasy setting. The project is focused on a combat system, meaning users are required to observe and take part in acts of violence. Cartoon violence is removed from more realistic portrayals of violence by enough of a margin that a user's sensitivity would need to be severe to experience adverse effects, but it is worthy of consideration.

Video game addiction is a recent topic in psychology and health. Gameplay loops that provide similar stimuli and rewards to a Skinner Box can manipulate players and cause psychological harm, with potential financial consequences occurring as a result.

Those Negatively Impacted

Video games with similar scope as this project rarely impact individuals outside of the users and developers (and any parties related to them). Users that feel pressured into experiencing the game against their will or who do not have adequate knowledge of the project's specifics will have a higher chance of being impacted in a negative way. As the developer, there is also the chance that a project can have a negative impact on you, such as succumbing to pressure to implement features out-of-scope or enduring criticism from users that is toxic and not productive.

Addressing Ethical Concerns

An individual's intent is not under the control of anyone else by definition, but measures can be taken to give someone a greater opportunity to act within their own volition. The project details, content, and nature have been exposed entirely to all involved in the testing phase of development. A strong effort has been made to reduce any pressure one might feel to partake in testing, the hope being that everyone is a willing participant. Users who may play this game at a later date, and in a more complete form, will be given the standard amount of information for such a game so that they may make a conscious decision on whether to play it or not (noting any violence, mature themes, and so on that they may encounter).

As for video game addiction, a subject that is dear to me, there was not much needed to curb such a risk at this time. The most that could be done was to keep it in mind - to be vigilant that development did not go into avenues that could exploit addictive behavior. Addictive game loops are not present in the final product and, being a prototype of small scale, there was always a small risk for their accidental presence.

The ethical concerns that arise from actual development must not be overlooked. Being a person that cares for their own well-being, the responsibility to protect myself from unethical practices is high. Maintaining a healthy work-balance, keeping to the project scope, and only allowing stretch features to be tackled with extra time are a few of the points within my plan of keeping development ethical.

Legal Considerations

Copyright Infringements

There should be no issues with copyrights as all assets used in the project are either original works or under a Creative Commons license (with the necessary attributions where necessary). The software used was Unity, with a free license that covers projects until they accrue $100,000 in sales. There is one code snippet that has been sourced from a third-party, but the code was given as part of a tutorial with the intent that individuals would use and modify it. A third-party asset was used for the pathfinding – a free version of the asset that the developer allows free and open use of.

Localization Issues with Content

There are no obvious legal issues that would arise from the content of the project. The game depicts cartoon violence and may contain mature themes in the story when it is implemented further in development. For most territories, stating the nature of the content upfront is enough - making such games in the United States is not illegal and internationally, the most a body could do is block the game from being available.

VII. Timeline and Milestones

Table 4

Timeline of Development

Design Phase	Description
Game foundations I (Completed)	Grid-based movement implemented with coroutines
Game foundations II (Completed)	Basic “attacking” between player and enemy
Combat I (Completed)	Ranged attacks / projectiles
Turn system (Completed)	Turn system and interfaces on actors for stalling their behavior
Combat II (Completed)	Area of Effect attacks / destructible projectiles
UI Stage (4/26 - 4/29)	Simple UI (mainly for testing) Camera-follow script
Asset creation (4/29 - 5/3)	Revamp of assets for final build
Enemy I (5/4 - 5/6)	Unique enemies
Pivot (5/6 – 5/8)	Switch to full-directional movement and real-time combat
Enemy II (5/9 – 5/11)	Enemy AI implemented with behavior trees.
Polish I (5/12 - 5/20)	Add miscellaneous features and refine

Audio system (5/20 - 5/24)	Sound effects and cues
Level Design (5/25 – 5/28)	Simple level with objective and a win condition
Main Menu (5/29 – 5/30)	Create main menu assets and implement scene loading
Compile (5/31)	Creation of first test build
Test phase (6/1 - 6/7)	Collect testing feedback and make necessary changes (repeating as needed and able)
Polish II (6/7 - 6/13)	Polish build for turn-in, added last camera feature for time-stop visual

Comparing the actual development timeline (see Table 4) with the proposed timeline (see Appendix Table B1), much of the design and implementation needed to be shifted after the pivot. Focus was placed more on level design, visuals, and polish than AI because it had the highest net impact based on implementation time. Major milestones (see Appendix B) were still applicable after the pivot and all of them were met.

VIII. <u>Usability Testing / Evaluation</u>

Feedback for Project Wisp was conducted remotely, due to distance and the current COVID-19 pandemic. Testers were given a test build of the project and a link to the compiled survey as a Google Form (see Appendix A). The test builds were accessible as a zip file on Google Drive, meaning that a new file needed to be uploaded every time there was a change made to the build. Scheduling conflicts disallowed for face-to-face testing sessions, so a short debriefing with the testers, when they were available, was the compromise made. While initial reactions were missed, feedback still proved valuable for a project at this early stage.

Tester Game Experience

With no formal client, a focus group of individuals within the target audience was chosen – their interest and experience with games ranged from hobbyist to professional game developer. This relationship with game development influenced the feedback received from the testers. Tester 3 was the most critical of the overall gameplay, expressing indifference to many of the features. This person is a hobbyist, with no game development experience, and was not briefed prior to testing on the scope of the project. When debriefed after submitting the survey, this tester stated they did not know what a game prototype should contain. Tester 4 does have experience in professional game development and, while they indeed had criticism to offer, it was much more constructive and took the fact that this is a prototype into consideration.

Black Box and White Box Testing

Black box testing was conducted with three of the four testing participants – the fourth was available to conduct white box testing on a few occasions. The purpose behind black box testing was to get focused feedback from testers, without the running systems in the Unity inspector distracting from the gameplay. The one tester who did conduct white box testing (tester 1) has extensive Unity experience, which offered a different perspective on the project.

Having one tester, and not multiple, conduct white box testing turned out to be a good choice. The white box tester was able to help more with bugs and implementation issues, thanks in large part to their Unity experience, but having another tester perform a similar role would have felt redundant. Testers that conducted black box testing had feedback that focused much more on gameplay, as was intended, which is much more valuable in higher volumes. If there were multiple people looking over the actual code and implementation, the feedback could get overwhelming, whereas gameplay feedback has little value if it is only coming from one person (they may not be a good representation of a whole demographic). That said, more testing is still needed – one of the testers had issues playing the game correctly because they did not consult the README as instructed and time limitation disallowed an additional attempt.

Feedback Conclusions

Game controls fared well and the UI, as presented, did not garner criticism. There was mention of UI elements lacking, but that was expected as certain quality of life pieces were left out of the prototype. Ranged combat had a very positive response, though melee combat was expressed as needing much more refinement. Testers felt that the overall system was unique and fluid, with only tester 4 using ‘clunky’ to describe the experience.

In response to the feedback received, melee combat was tweaked once more before the final test build, giving the player more incentive to use melee over ranged attacks by increasing the damage it dealt. The input was also tuned to make melee feel more responsive. As a last piece of polish, the camera-grayscale effect was added to signify when time had stopped in-game. Mentioned explicitly in one review, and conferred from the others, the time-stop mechanic needed some way to prompt the player that it had been activated.

Reviewing the completed surveys, feedback held very few surprises, with many criticisms placed on known issues. There were a couple of epiphanic suggestions, such as Tester 4 mentioning the camera effect for when time stops in combat, but even those comments that were expected were constructive – seeing feedback that is expected tells you two things: you have a good understanding of your audience and have thought thoroughly about the project.

IX. Final Implementation

The intent of Project Wisp was to have a functional prototype of a combat system that could be added to and refined for use in a future, large scale project. Features necessary for the design include: a player, unique enemies, event system, combat abilities, and elements to build a test level. Polish of these features was not a priority, but adhering closer to the design pattern of composition, new features can be added and tested quicker even in a rough state.

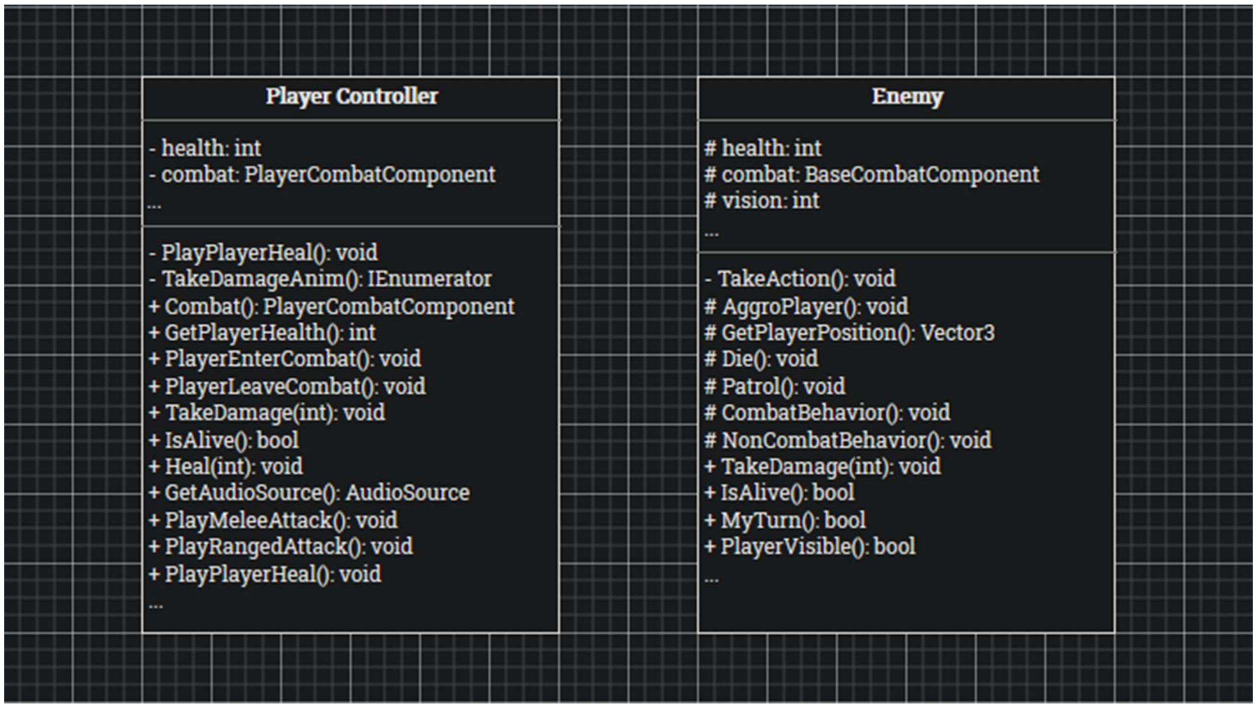
Actors

Actors within this prototype include the player and enemies. The player does not differ much from enemies, with all actors implementing a unique version of the same base components, but the player object is accessed and referenced from a User Input object, giving the user control over that specific actor. Coupling between the User Input object and the Player Controller component is present, but, with little in the way of decoupling, the User Input object would allow for user control over any passed in actor in the game (something valuable for a future mind-control ability, etc.). In their role as an actor, much of the player character's functionality lies in the Player Controller class, serving as a repository for components, much like the Enemy base class, and a communication switchboard between them.

To keep the inheritance hierarchy shallow, enemies all inherit from a single base Enemy class (See Appendix). This parent class was designed to hold as much boilerplate functionality as possible, forcing the child objects, through C# Abstract, to implement all unique behavior themselves. The upside to this design choice allows the use of polymorphism when an Enemy needs to be referenced within another system – when acting upon an Enemy object, external systems do not need to know what type of Enemy it is referencing, only that the object is an Enemy. An obvious drawback is the loss of flexibility in creating unique enemies. This problem was mitigated by designing the enemies beforehand, so that only the absolute necessary functionality would be shared between them.

Figure 1

Class definitions of Player Controller and Enemy



Actor Components

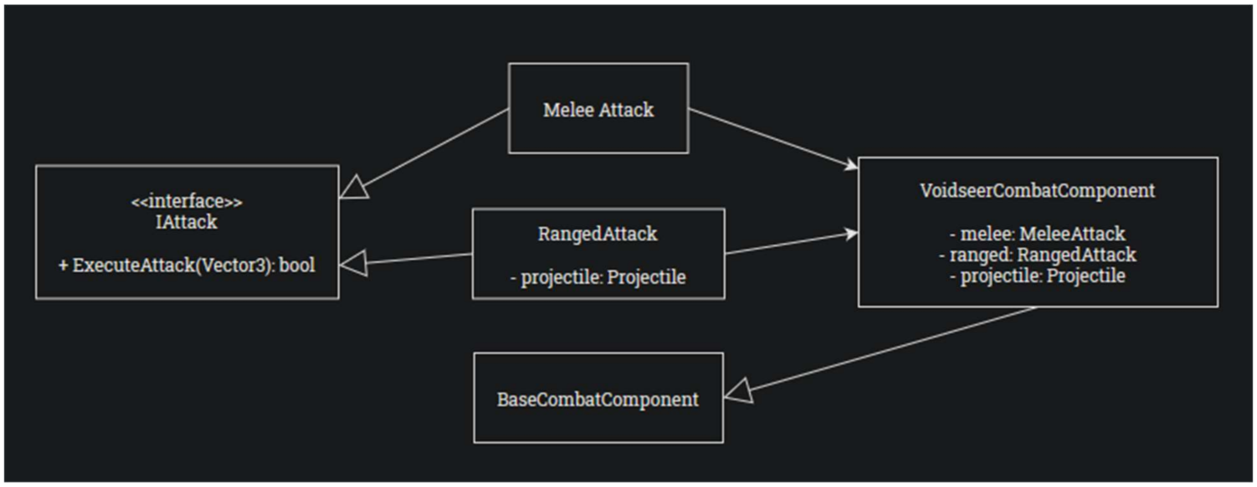
The number of components actors implement was reduced after the pivot, due to some being obsolete, and there was not enough time to do a proper refactor on all the added systems to separate out each into its own component. Much of the combat-side of gameplay does retain its composition design, with each actor-type instantiating their own unique Combat Component and each Combat Component instantiating the desired ability objects. Utilizing a public enum for the Attack Type, an actor’s Combat Component can implement a single Perform Attack function that takes an Attack Type and three-dimensional vector representing the target. This, once again, gives Combat Components the ability of polymorphism, with outside systems needing only to call the Perform Attack method on the component without knowing what unique actor version of the component they are dealing with.

Functionality that had their own components, but were removed after the pivot, include movement and turn behavior. Movement is now handled by the physics engine, meaning very little unique movement behavior needed to be created between actors – values to velocity and speed could be changed on the actor without the need for a whole new component. Turn behavior became obsolete once the turn system was removed. The enemy AI is an example of a system to be refactored in the future, allowing for the creation of its own component.

Minor components on actors that have straightforward behavior are Loot Pools (to allow enemies to drop given items upon death), Pathfinding (attached in the Unity Inspector and accessed in the base Enemy script), and Audio Sources.

Figure 2

Combat Component Implementing Attacks



Note. The melee attack and ranged attack objects implement the IAttack interface. The Voidseer combat component implements the base combat component and initializes attacks.

Items, Abilities, and Area of Effect

There is no specific script attached to an object that makes it an item, but rather the concept of an item is mainly used for distinguishing objects when rendering (what objects are sorted in front of what in the 2D scene). Health pickups, projectiles, and the thrown poison phial are the extent of this concept, all having their own unique script to define them. Once again, this composition approach allows for flexibility in designing future objects. The health pickup script gives any attached Game Object the ability to heal an actor – the projectile script gives any Game Object the ability to be instantiated and affected by physics (dealing damage upon collision). The thrown phial, for instance, is a Game Object with the same projectile script on it as the projectiles proper, but it has an added script for instantiating an area of effect upon impact.

Combat abilities include melee and ranged, with the ranged attack instantiating a projectile in the defined direction. Melee works similar, but the melee attack instantiates an Overlap Box in the given direction, dealing damage to any actor within its boundaries. Visually, this is represented by a Sprite that is attached to the character Game Object, enabling it and disabling it where directed to represent an attack being made. Area of Effect objects differ slightly from melee and ranged, in that they are not accessed directly by any combat script. Instead, Area of Effect is its own Game Object that can be instantiated at any point. The current usage is to have an Area of Effect instantiated at the point where a Thrown Destructible (the phial, in this case) is destroyed.

Movement

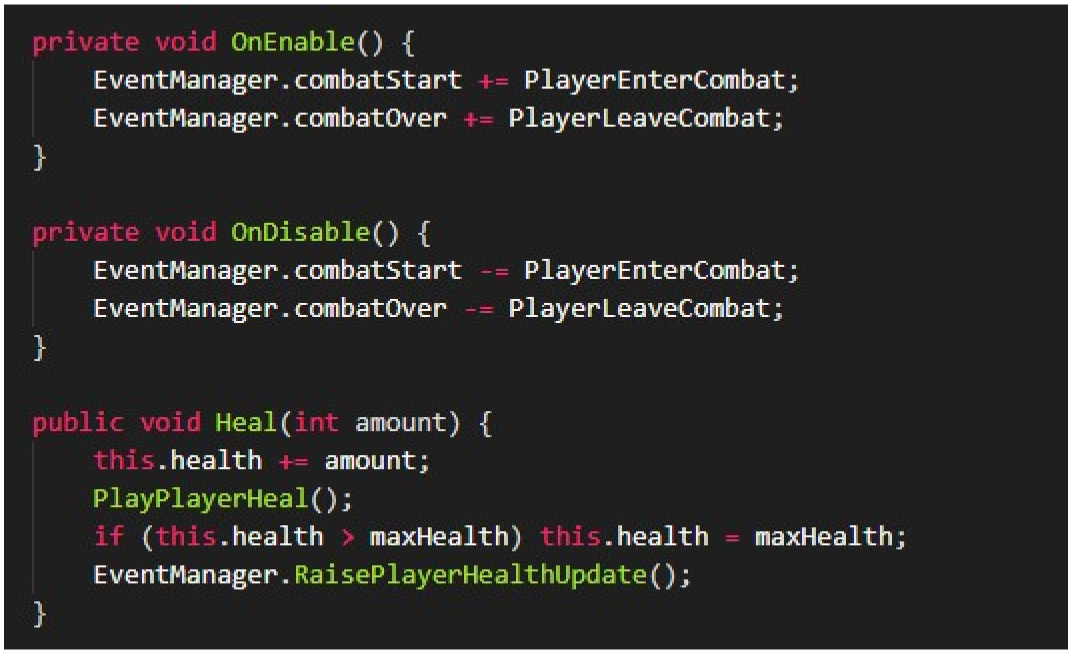
With the shift to full-directional movement (on the 2D plane), movement is now handled by the physics engine. Any object that is to be moved has a Rigidbody component attached, which is necessary in Unity if Game Objects want to interact with physics, and the velocity of the object is manipulated. When an object needs to stop, the velocity is set to zero, otherwise a Speed variable is used to calculate the amount of velocity the object has. Projectiles have an added step: they need to be rotated in the proper direction before their velocity can be set. This is done with an appropriate Quaternion object, to rotate the object on its z-axis.

Event Manager

To decouple systems in an elegant way, a custom Event Manager script was created – a static object that serves as a switchboard for messages being passed between other objects and scripts. For a message to exist, it must be created in the Event Manager class, with a corresponding method that allows for the event to be raised. Once that is in place, other scripts and objects can subscribe to these events, attaching a callback method to be initiated when the event is raised. Objects must unsubscribe from events upon being disabled to notify the Event Manager that there is one less listener for that event. To raise an event, subscription to it is unnecessary, and all an object or script must do is invoke the static Event Manager class and the method upon it to raise the desired event. Examples of subscribing, unsubscribing, and raising an event can be seen in Figure 3 – code was taken from the Player Controller class.

Figure 3

Subscribing, Unsubscribing, and Raising an Event



```
private void OnEnable() {
    EventManager.combatStart += PlayerEnterCombat;
    EventManager.combatOver += PlayerLeaveCombat;
}

private void OnDisable() {
    EventManager.combatStart -= PlayerEnterCombat;
    EventManager.combatOver -= PlayerLeaveCombat;
}

public void Heal(int amount) {
    this.health += amount;
    PlayPlayerHeal();
    if (this.health > maxHealth) this.health = maxHealth;
    EventManager.RaisePlayerHealthUpdate();
}
```

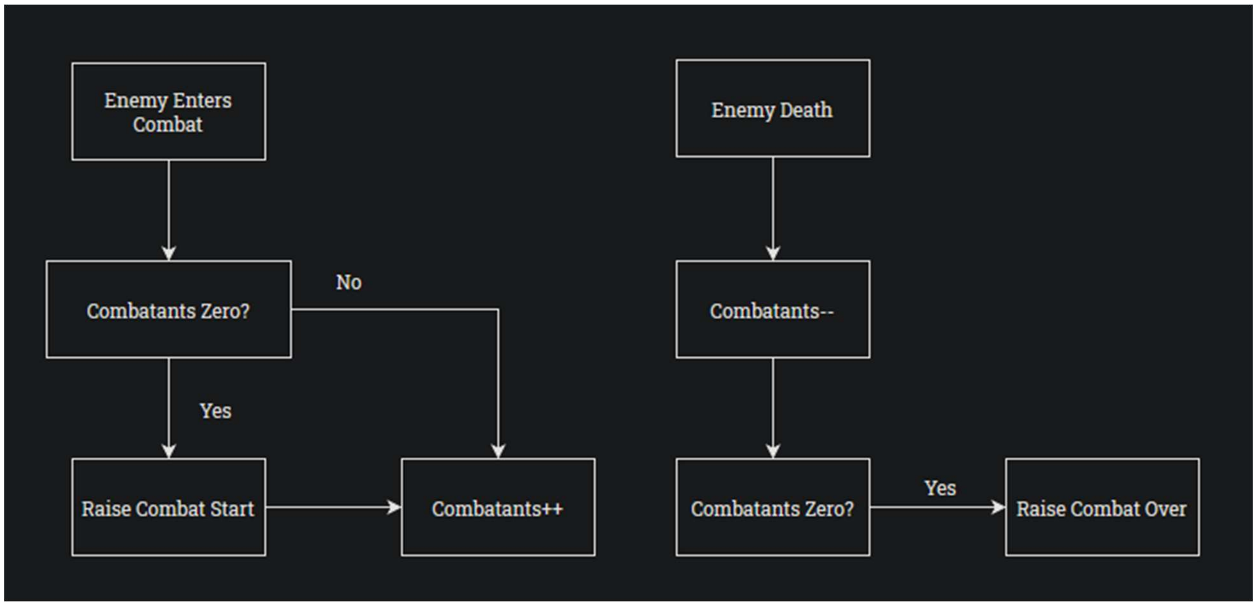
Note. The += and -= are used for the subscribing and unsubscribing of a method to an event (respectively). The Heal method contains an example of an event being raised on the last line.

Combat System and Time-Stop Mechanic

After the pivot, there is little in the way of a centralized “combat system” present in the design. Many of the actors and objects handle their own functionality and interactivity between themselves, leaving little reason to have a system to babysit everything. There is still a Game State object present in the scene, which does serve some purpose in terms of combat behavior, but it also serves the important role of loading and unloading resources at scene start. That said, the combat functionality present in the Game State is important, with the entire state of combat being held and handled within that object. When enemies enter or exit combat with the player, that message is received in the Game State, increasing the number of Combatants tracked – when that number reaches zero, combat is over. It is the Game State that notifies the actors and objects of the combat state, allowing them to handle that event however they must.

Figure 4

Determining Combat State

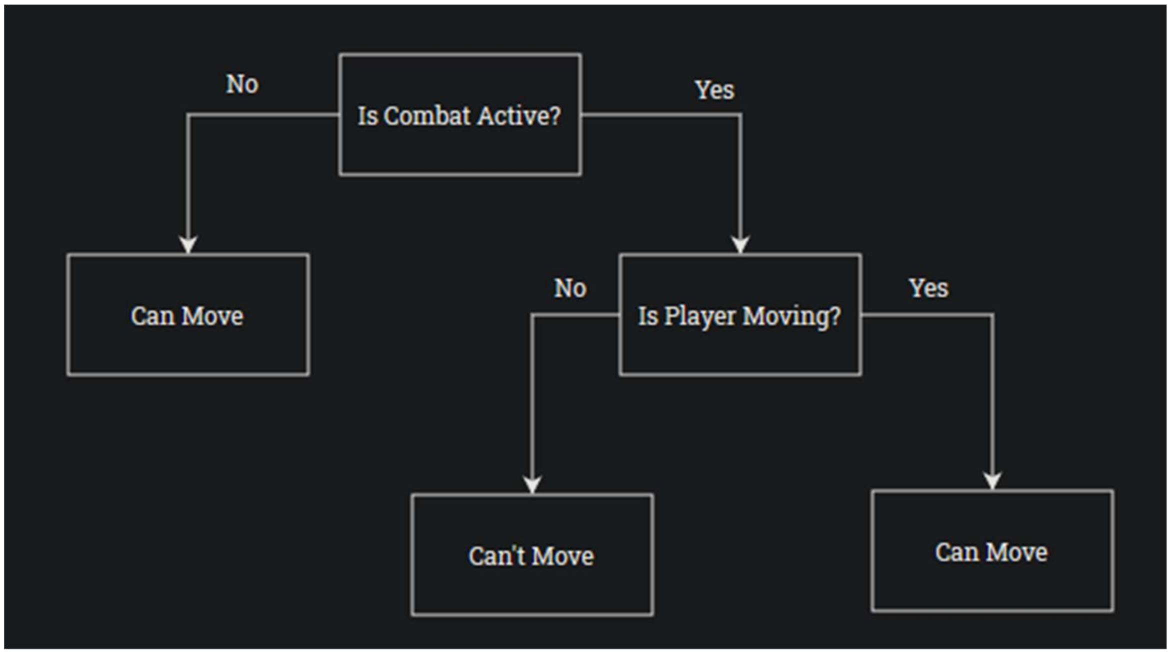


As of now, the combat state is only used by the actors and objects to execute their behavior for the time-stop mechanic. Though it is a major feature of the game, the time-stop mechanic does not exist in a singular form anywhere in the code but is instead a set of behavioral rules that an object must follow. There is no enforcement of this behavior in the current design, but the next refactor will determine whether this feature should be a standalone component or a set of methods within an Interface that relevant objects must implement.

When combat is active, the responsibility to notify objects in the scene to stop movement falls onto the player, with an event being raised based on the user's input. When the axes of movement are null, objects are told that the player is not moving, and vice-versa. Much like the time-stop behavior on affected objects, the player's role in this system can be removed and a standalone system for triggering the time-stop can be implemented.

Figure 5

Decision Tree for Moving Objects



Systems Interacting

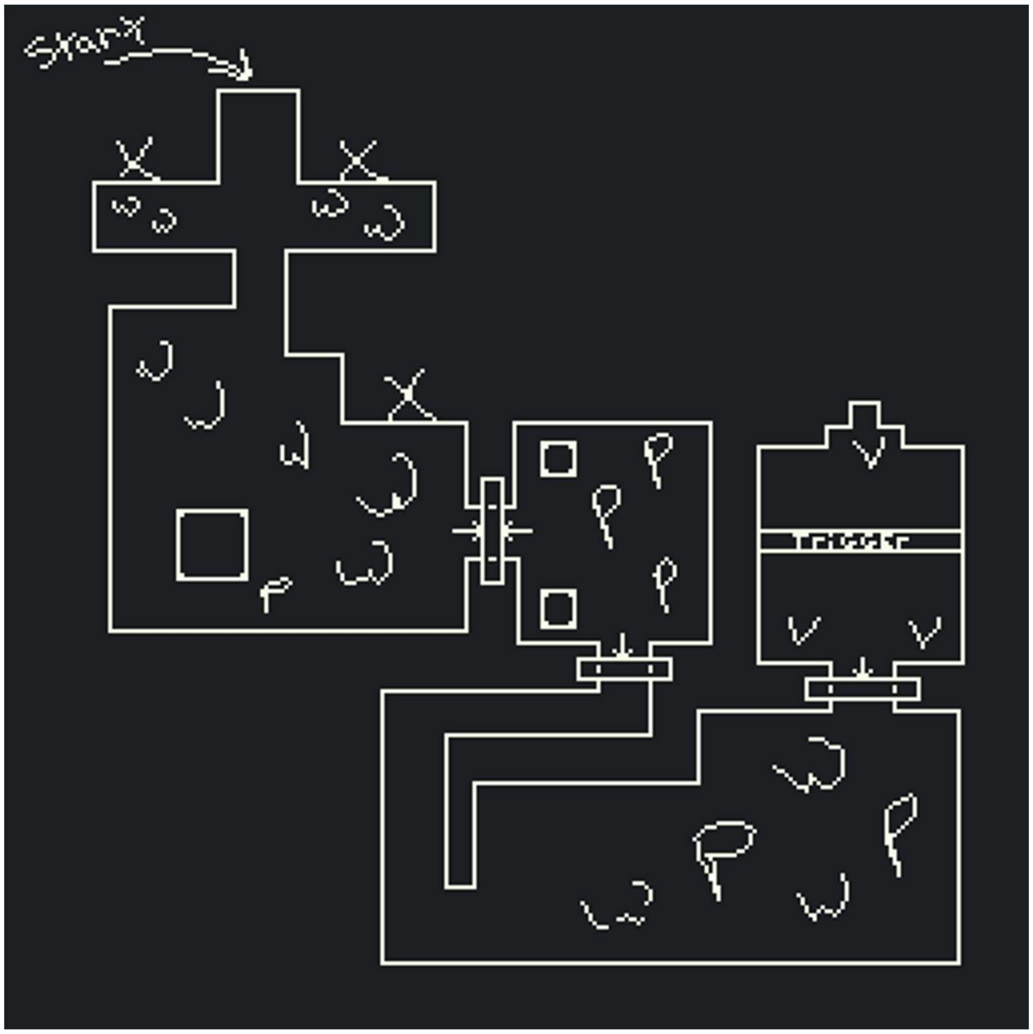
Compared to the first design (with the Turn System), the current design has loose coupling between systems - the Game State notifies objects when combat is active, and the player notifies objects when it has stopped moving (when combat is active). Collisions, effects, damage, etc. are all handled by the individual objects themselves, with most complexity coming in the way of the Event Manager. Since there is not explicit coupling, the downside to this design is that it is not always obvious what objects are affected by an event, making debugging that much harder. The flexibility it gives makes up for that, with objects only needing to subscribe to the events they must and unsubscribing when they are disabled.

Level Design and Obstructions

When the time came for designing a level, the main goal was showcasing all the different features of the project in one package. Enemies were dispersed amongst the rooms based on difficulty and an objective was added (kill all enemies) that would ensure the player would have to traverse the whole level. Triggers were created for spawning enemies and locking the player in rooms – this was done to keep the player from seeing what enemies were coming up next and to add a fun element of surprise to an otherwise straightforward level.

Figure 6

Original Sketch of First Level



The doors that barricade the player in a room are simple Game Objects with Colliders that are enabled and disabled based on their corresponding trigger. Triggers can be in the form of an enemy death or a trigger collider that the player walks through. The wiring of the triggers and the doors they enable, or the enemies they spawn, is handled by an Activated Room script which can have as many instances in the scene as necessary.

Scene Transitions

The current design has two scene transitions: one is handled by a script within the main menu scene, and the other is handled in the Game State script. There is no current need for data persistency, so scene transitions are hard resets. There are a few static variables that need to be managed between states (on the Game State script), but all other information is not needed between scenes. As it stands, adding another level, or multiple levels, is possible since the appropriate scene is loaded from the main menu based on the scene index in the build order – level one is at index one and level two would be at index two and so on since the main menu takes up the zero index. Any scene that wishes to return to the main menu will always call the same scene index (zero) within the Scene Manager to return to the main title screen. The Scene Manager is built into Unity, with proper scene order being the only manual maintenance needed with the current design.

X. <u>Discussion</u>

Initial Design and Problems

The motivation behind the initial design was to create a mix of tactical and action combat. To achieve this, movement was to be restricted to a grid and actors would execute turns instead of real-time action. Using a global turn-system, it was thought that objects could take turns while staying in sync with each other, with all behavior stopping once the player entered combat – objects would only take a turn once the player took a turn while combat was active.

Table 6

Design Features – Pre-pivot vs. Post-pivot

	Movement	**Time-Stop**	**Player Abilities**	**Enemies**	**AI**	**Features for Testing**
Initial / Pre-pivot	Grid-based	Turn-System	Three	Three	Utility Curves	Simple UI and test level
Post-pivot	Full Directional	Event System	Three	Three	Behavior Trees	Simple UI and test level

Confidence was strong in this design, but the need to keep the scope limited created issues when deciding what to include. Any feature added to the design was another potential element to interact with, and possibly derail, the turn-system. This heavy reliance on one system to maintain the entire game flow was a justified concern. Combat abilities, cooldowns, AI routines, all depended on the implementation of the turn-system.

Initial Implementation and Problems

Movement was done with coroutines, instead of snapping objects to tile coordinates – by using coroutines, smooth movement from tile-to-tile could be accomplished, giving the illusion that everything was moving in slow motion during combat. The problem this caused was an object's turn now took a measurable amount of time (if it was moving, then this would be the time it took to move the object on the screen). With this delay, many opportunities arose for objects to fall out of sync and the resulting bugs became time consuming to deal with.

Gameplay and the responsiveness of actions also suffered from the turn-system implementation. A camera-follow script was added to the player, allowing large levels to be traversed seamlessly, but this meant that the camera was having to stop and start along with the player – the camera was now making the game feel rigid and movement to feel stiff. It became increasingly clear that a pivot was needed, meaning some features would need to be scaled back due to the loss of time.

Post-pivot Design and Problems

Pivoting to full-directional movement meant many of the existing systems had to be redesigned for the new approach. The turn-system was removed in its entirety, leaving only the Event Manager to communicate with actors and thus caused an issue where inexplicit dependencies were created between systems. Enemy AI needed an overhaul, since the old system relied on tile coordinates, and so simple behavior trees were chosen over utility curves for both time and practicality reasons – it became clear, once development was underway, that utility curves for an AI in such a game would have been overkill, even without the pivot.

The pivot design resulted in more problems being solved than created. Leaning on the physics engine for movement and allowing objects free movement turned out to be a simpler solution in the end, thanks to my prior knowledge of the Unity Engine.

Post-pivot Implementation and Problems

With the resulting changes to the design, more systems needed to be condensed rather than replaced. Colliders in the scene saw a heavy scaling back, with actors needing only one collider (as opposed to the two they needed under the turn-system). With a simpler design for enemy AI, implementation was straightforward, but was not able to be refactored into composition. Time limited, the emphasis was put on functionality and not modularity of code. The implementation proved to be simple enough that the lack of composition was not much of an issue. Pathfinding was the last major system to reconstruct, with a proper A* algorithm being needed to guide the enemies. Unity has built in pathfinding capabilities for 3D movement, but not for 2D movement – this issue was easily resolved by using a well-established and well-documented asset that adds such functionality to the engine. The asset is powerful, with many avenues for customization, so time needed to be spent learning the tool before implementation could occur. In the end, the knowledge became a valuable addition to my skillset.

Project Rollout to Users

Distributing the test builds to testers was not a major issue. The game would build in Unity, creating all necessary directories and, of course, the executable, which is easily zipped into a compressed folder. This zipped file was added to Google Drive, with a shareable link created and given to the testers. In terms of efficiency, this workflow worked fine enough for the small-scale testing that was being conducted, but would not scale well – every time a new test build was made, the process of compressing the file, uploading it, and getting a link would need to be replicated.

XI. Conclusion

The problem Project Wisp sought to address, the question it hoped to answer, was if a combat system could be both tactical and action oriented – a proposed solution was to stop time in the game whenever the player entered combat, but the initial implementation of that solution was less than satisfactory. A pivot from grid-based movement and turn-style actions to real-time, full directional movement was the needed approach. Discipline was needed in how the project was approached, with an iterative development process being used, though the work was done independently.

Features of the final design include a playable character with three abilities, three enemies to face, a simple level and UI, as well as a unique take on combat with the time-stop mechanic. Combat abilities include melee and ranged attacks, and area of effects can be instantiated in the scene. Enemies use a simple AI approach with behavior trees, along with a powerful A* pathfinding asset that provides flexibility in their basic behavior. A custom Event Manager enables the time-stop mechanic to function, with all actors and moveable objects in the scene stopping their behavior upon receiving the appropriate signal (and resuming their behavior in the same fashion).

Being a solo game project, there was a lot that could go wrong (and did) in development, making Project Wisp a risky decision. There were no collaborators to fall back on if progress was stalled, there was no safety net – the project's success would be determined by my output alone, creating pressure on top of the work needing to be done. Confidence in my skills and decision-making was needed. When the outlook was grim, being able to trust myself to decide on an appropriate course of action supplied the courage needed to pivot – that pivot being the single best decision made during development. Confidence and trust in myself, both lessons worth learning.

Project Wisp has a future, though it may not be in the exact form presented for the course. The intent was always to make a game system that could be placed into a larger-scale game and that is still the intent. This other game will not have combat as the sole focus, but I believe every system included in a game should be fun, therefore creating the need to prototype the combat. There are still refinements to be made and experimentation to be done. More enemies need to be tested, enemy AI needs to be smarter, simple animations for combat abilities – the difficulty will be in prioritizing features, not devising them. Though it is only a prototype, a first step into game development, Project Wisp has given me, not only an item for my resume, but a direction to follow in pursuing my passion.

XII. References

Gach, E. (2016, October 9). Darkest Dungeon Ruins Its Endgame. Retrieved March 31, 2020, from https://kotaku.com/darkest-dungeon-ruins-its-endgame-1787578526

Hamilton, K. (2016, February 25). Superhot: The Kotaku Review. Retrieved March 31, 2020, from https://kotaku.com/superhot-the-kotaku-review-1760902123

Hovermale, C. (2019, January 20). Do you prefer turn-based RPGs or action RPGs? Retrieved March 31, 2020, from https://www.destructoid.com/do-you-prefer-turn-based-rpgs-or-action-rpgs--539499.phtml

Ink Stains Games. (n.d.). Stoneshard. Retrieved March 31, 2020, from https://stoneshard.com/

Johnson, S. (2009, November 6). Analysis: Turn-Based Versus Real-Time. Retrieved March 31, 2020, from https://www.gamasutra.com/view/news/116864/Analysis_TurnBased_Versus_RealTime.php

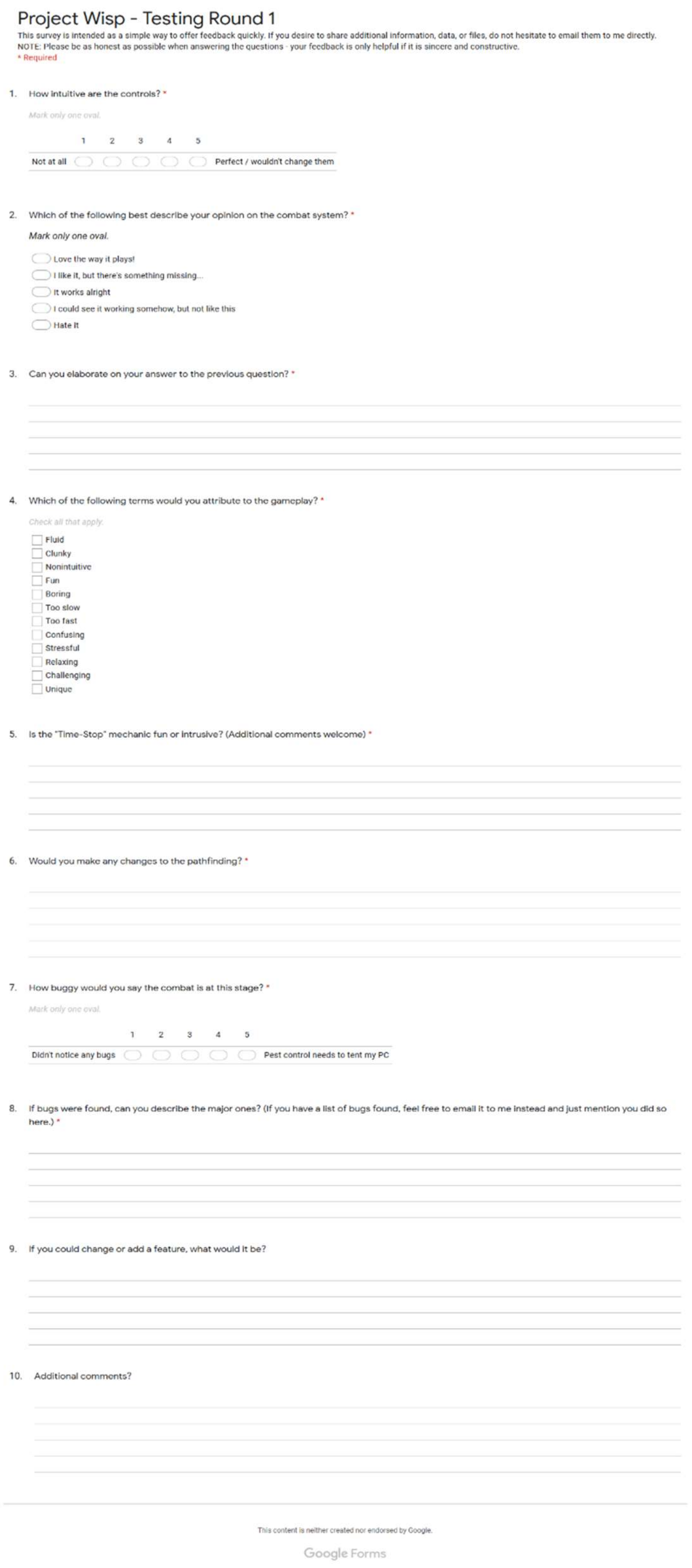
Red Hook Studios. (2020, March 16). Darkest Dungeon | A Red Hook Studios Game. Retrieved March 31, 2020, from https://www.darkestdungeon.com/

SUPERHOT Team. (n.d.). SUPERHOT - The FPS where time moves only when you move. Retrieved March 31, 2020, from https://superhotgame.com/

Suther, A. (2020, February 10). Stoneshard is Promising, but Too Brutal for Its Own Good. Retrieved March 31, 2020, from https://techraptor.net/gaming/previews/stoneshard-preview

Wilde, T. (2020, February 8). Wounded, intoxicated, and covered in leeches: a normal day in Stoneshard. Retrieved March 31, 2020, from https://www.pcgamer.com/wounded-intoxicated-and-covered-in-leeches-a-normal-day-in-stoneshard/

Appendix A



Project Wisp - Testing Round 1

This survey is intended as a simple way to offer feedback quickly. If you desire to share additional information, data, or files, do not hesitate to email them to me directly.
NOTE: Please be as honest as possible when answering the questions - your feedback is only helpful if it is sincere and constructive.
* Required

1. How intuitive are the controls? *

   Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Perfect / wouldn't change them

2. Which of the following best describe your opinion on the combat system? *

   Mark only one oval.

   - ◯ Love the way it plays!
   - ◯ I like it, but there's something missing...
   - ◯ It works alright
   - ◯ I could see it working somehow, but not like this
   - ◯ Hate it

3. Can you elaborate on your answer to the previous question? *

4. Which of the following terms would you attribute to the gameplay? *

   Check all that apply.

   - ☐ Fluid
   - ☐ Clunky
   - ☐ Nonintuitive
   - ☐ Fun
   - ☐ Boring
   - ☐ Too slow
   - ☐ Too fast
   - ☐ Confusing
   - ☐ Stressful
   - ☐ Relaxing
   - ☐ Challenging
   - ☐ Unique

5. Is the "Time-Stop" mechanic fun or intrusive? (Additional comments welcome) *

6. Would you make any changes to the pathfinding? *

7. How buggy would you say the combat is at this stage? *

   Mark only one oval.

	1	2	3	4	5	
Didn't notice any bugs	◯	◯	◯	◯	◯	Pest control needs to tent my PC

8. If bugs were found, can you describe the major ones? (If you have a list of bugs found, feel free to email it to me instead and just mention you did so here.) *

9. If you could change or add a feature, what would it be?

10. Additional comments?

This content is neither created nor endorsed by Google.

Google Forms

Appendix B

Milestone List from Proposal

- Basic implementation of movement and interactions on a grid
- Combat systems implemented
- Turn system created and refined
- Enemy variation through unique AI
- Creation of a small level, suitable for testing
- Testing process and further refinements
- Project completion

Table B1

Initial Timeline of Development

Design Phase	**Description**
Game foundations I (Completed)	Grid-based movement implemented with coroutines
Game foundations II (Completed)	Basic "attacking" between player and enemy
Combat I (Completed)	Ranged attacks / projectiles
Turn system (Completed)	Turn system and interfaces on actors for stalling their behavior
Combat II (Completed)	Area of Effect attacks / destructible projectiles
UI Stage (4/26 - 4/29)	Simple UI (mainly for testing)
Enemy stage (4/29 - 5/6)	Enemy AI using utility curves
Enemy stage (5/6 - 5/11)	Unique enemies (count dependent on time)
Level Design (5/11 - 5/16)	Simple level with objectives and a win condition

Audio system (5/16 - 5/19)	Sound effects and cues
Polish I (5/19 - 5/22)	Clean up and preparation for test build
Compile (5/22)	Creation of first build
Test phase (5/22 - 5/25)	Disperse build to testers with supplementary material
Test phase (5/25 - 6/6)	Collect testing feedback and make necessary changes (repeating as needed and able)
Polish II (6/6 - 6/13)	Polish build for turn-in, organize test data, debrief with self upon project completion