

Software Requirements Specification (SRS)

Kiwi: The Mathematical Mage

Team: 7

Authors: Kaden Gardiner, Lily Grippo, Roberto Mercado, Riley Miller, Allen Zammer

Customer: 4th Grade Students

Instructor: Dr. James Daly

1 Introduction

This document is the Software Requirements Specification (SRS) for the game *Kiwi: The Mathematical Mage*. It provides a detailed description of the game's intended functionality, constraints, system behavior, and overall purpose.

1.1 Purpose

The purpose of this SRS document is to provide a comprehensive description of our functional and non-functional requirements, design constraints, and intended features. This document will serve as a reference for development teams, testers, and users to get information on the software. The primary goal of the software is to provide an educational game intended to teach and reinforce basic math concepts (specifically addition, subtraction, multiplication, and division) to 4th grade students.

1.2 Scope

The software product being produced is the game *Kiwi: The Mathematical Mage*. The application domain for this software is an educational game. This game is intended to reinforce the player's understanding of 4th grade math concepts by having them solve addition, subtraction, multiplication, and division problems. The game's aim is to build confidence in these skills by rewarding players for solving problems. The game will not teach students specifically how to perform these mathematical operations. It is assumed that the students have learned how to perform them from previous grades.

1.3 Definitions, Acronyms, and Abbreviations

- Kiwi: The Mathematical Mage: The title of the game.
- Player: The user.
- Kiwi: The player character, who happens to be a magical kiwi bird.
- Egg: An object that the player is tasked with protecting.
- Predator: Enemies that seek eggs to consume them. The player must defeat them to avoid losing the game.
- Predator Wave: An in-game event that spawns several predators. The number of waves the player has previously gone through determines how many predators are spawned. May be abbreviated to wave.
- Track: Any path that a predator may follow to reach the player's eggs.
- Spells: Magical projectiles cast by the player to damage and apply status effects to predators within an area of effect.
- Status Effect: Effects inflicted by spells on predators which impair them in some way.
- Mana: The resource the player must consume to cast spells, which is generated passively at a fixed rate and actively by solving math problems.
- Berry: The resource the player may use to upgrade spells, which is gained at the end of each predator wave.

1.4 Organization

The rest of the SRS contains information about the software in a more technical sense. It discusses the perspective of our game in terms of hardware and software, as well as the functions of the product. The document then outlines the specific requirements that the game is intended to achieve. After the document contains the UML diagrams for the game. Then, finally, the document discusses the game prototype and how to use it.

The rest of the SRS structure is as follows:

- Product perspective
- Production Function
- User Characteristics
- Constrains
- Assumptions and Dependencies
- Apportioning of Requirements
- Specific Requirements
- Use Case Diagram
- Sequence Diagrams
- State Diagrams
- Prototype Discussion

2 Overall Description

This section covers the context of our software. It will discuss the interfaces through software and hardware, and the constraints as well. Then the characteristics of the expected users will be discussed along with the assumptions about them. Then finally discussed will be the asportation of the requirements.

2.1 Product Perspective

Our product should be used in the context of a 4th grade classroom to reinforce basic math concepts. It should be used alongside the regular teaching of math; it is not built to replace the learning of these concepts. The game is built on the Godot engine and does not require any larger system to run. It is designed to run on lower-end school computers so the student can have access to the game in the classroom. The game interfaces with a screen display, keyboard, and a mouse.

2.2 Product Functions

The software will perform the core function of reinforcing learning for 4th grade students. It will consist of a main menu where the user can select the type of math they want to focus on during gameplay. Once the player hits the start button, the player will go into the game loop where they will be presented with the main game screen. From here, the player can answer a math question and gain mana that will then be spent. The player will start on the first wave where the predators will spawn. Once the wave is over, they will gain a berry to upgrade their spells. Afterwards, they are presented with an option to move onto the next wave.

2.3 User Characteristics

Generally, we expect our users to have limited skills in using computers because of our 4th grade audience. They should, however, be expected to be able to use a keyboard and mouse for basic functions of the game, such as typing in answers to math problems and dragging and dropping the spells onto the predators. The skills they need to progress and succeed within the game are expected of them, but not much more than that. These skills include basic 4th grade level addition, subtraction, multiplication, and division.

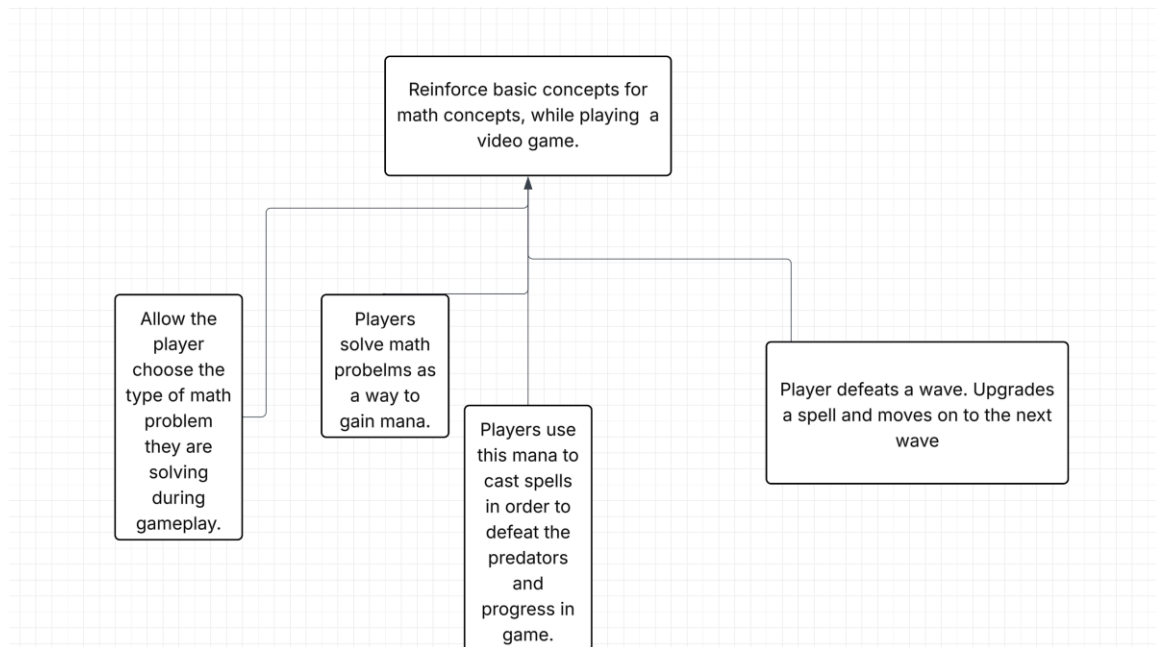


Figure 1 - High Level Goal Diagram.

This diagram illustrates the high-level goals of our game. The main goal of our game is to provide a way to reinforce 4th graders' math skills. They can choose the math topic they want to focus on during their gameplay. The players can solve a math problem to gain mana to afford to cast spells. They cast spells to defeat the predators on the track headed towards their eggs. Once they defeat the predators of the wave, they gain a berry and can upgrade a spell.

2.4 Constraints

- The system should be a computer that can run the Godot game engine.
- The player must have access to a keyboard and mouse to play the game.
- The game should only include basic math problems that a 4th grade student in Massachusetts should be able to solve.
- The game must be able to run on lower end school computers without a dedicated GPU.
- The game must not have content that is inappropriate for a 4th grade audience.
- The game must be able to be played locally on the user's device without an internet connection.

2.5 Assumptions and Dependencies

It is assumed that the player will play the game from a computer that has Windows, Linux, or Mac. It is also assumed that the user will have access to a keyboard and mouse to play the game. It is also assumed that the system will have a screen that will be able to display the game. The user is assumed to have basic computer skills where they can use a keyboard to type in the answer to the math problem and use a mouse to cast the spell.

2.6 Apportioning of Requirements

Currently, there are no specific additional requirements beyond the scope of this initial prototype. There will not be any more maps for this initial prototype. There are no other modes or math levels in the game other than what is initially added.

3 Specific Requirements

1. The game shall incorporate 4th grade math problems into gameplay.
 - 1.1. The game shall allow the player to focus on addition, subtraction, multiplication, or division problems.
2. The player shall help Kiwi defend a clutch of three eggs from endless waves of predators.
 - 2.1. Predators shall travel along a specific path to reach the clutch of eggs.
 - 2.1.1. Predators shall begin their path from one of three cave entrances.
 - 2.1.2. Predators shall select a random path that ends at the clutch of eggs.
 - 2.1.3. Predators shall damage and consume eggs one by one once they reach the end of their selected path.
 - 2.1.4. The game shall end when the third egg is consumed by a predator.
 - 2.2. The player shall assist Kiwi with casting spells to defeat predators.
 - 2.2.1. The player shall select the spell Kiwi should cast from a deck of cards.
 - 2.2.2. The player shall direct Kiwi to cast the selected spell at a location of their choice.
 - 2.2.3. Spells shall inflict damage and potential status effects on predators within an area of effect when cast.
 - 2.2.4. Spells shall enter a cooldown state for a short period of time after they have been cast.
 - 2.3. Kiwi shall require mana to cast spells.
 - 2.3.1. Spells shall cost a specific amount of mana to be cast.
 - 2.3.2. Kiwi shall generate mana passively at a fixed rate.
 - 2.3.3. The player shall help Kiwi generate mana actively by solving math problems.
 - 2.4. Predators shall be varied in terms of health, strength, and speed.
 - 2.4.1. Health shall be the measure of how hardy a predator is against spells.
 - 2.4.2. Strength shall be the measure of how quickly a predator can consume eggs.
 - 2.4.3. Speed shall be the measure of how fast a predator can travel along a path.
 - 2.4.4. Small-sized predators shall have low health, low strength, and high speed.
 - 2.4.5. Medium-sized predators shall have moderate health, moderate strength, and moderate speed.
 - 2.4.6. Large-sized predators shall have high health, high strength, and low speed.
 - 2.5. The player shall be rewarded with berries for helping Kiwi fend off waves of predators.

- 2.5.1. Berries shall be used to upgrade spell damage, status effect potency, cooldown time, and mana cost.
 - 2.5.2. The player shall be given time to upgrade spells between waves.
 - 2.6. The game shall keep track of how many waves of predators the player has helped Kiwi fend off.
 - 2.6.1. The game shall become more difficult as the player helps Kiwi fend off waves of predators.
 - 2.6.2. The game shall display the number of waves the player has helped Kiwi fend off alongside their greatest record after the game ends.
 - 2.6.3. The game shall record the greatest number of waves the player has helped Kiwi fend off across sessions.
- 3. The game shall provide guides that help the player learn how to play it.
 - 3.1. The game shall display a how-to-play guide when the player launches the game for the first time.
 - 3.2. The game shall provide hints during gameplay to help the player improve their skills.
- 4. The game shall display an engaging user interface that is suitable for 4th grade students.
 - 4.1. The user interface shall present information that can be easily understood at a 4th grade reading level.
 - 4.2. The user interface shall be free of clutter to avoid overwhelming the player.
 - 4.3. The user interface shall be stylized to match the whimsical feel of the game.
- 5. The game shall play music that invokes the emotion of whimsy in the background.
 - 5.1. The game shall allow the player to set the volume of the music.

4 Modeling Requirements

The following sections contain diagrams that model the project's functionality. These diagrams include one use case diagram, one class diagram, two sequence diagrams, and one state diagram. Each subsection will include one of these diagrams along with supporting content that describes what they are showing.

4.1 Use Case Diagram

The following diagram shows the interactions between the player and the underlying system of Kiwi: The Mathematical Mage. The player is represented by a labeled stick figure. Use cases are represented by circles with text labels at their center. Interactions between the player and individual use cases are represented by solid lines. Interactions between use cases are represented by labeled dashed arrows. More specifically, the `<<includes>>` label indicates that the target use case is achieved when the source use case is achieved.

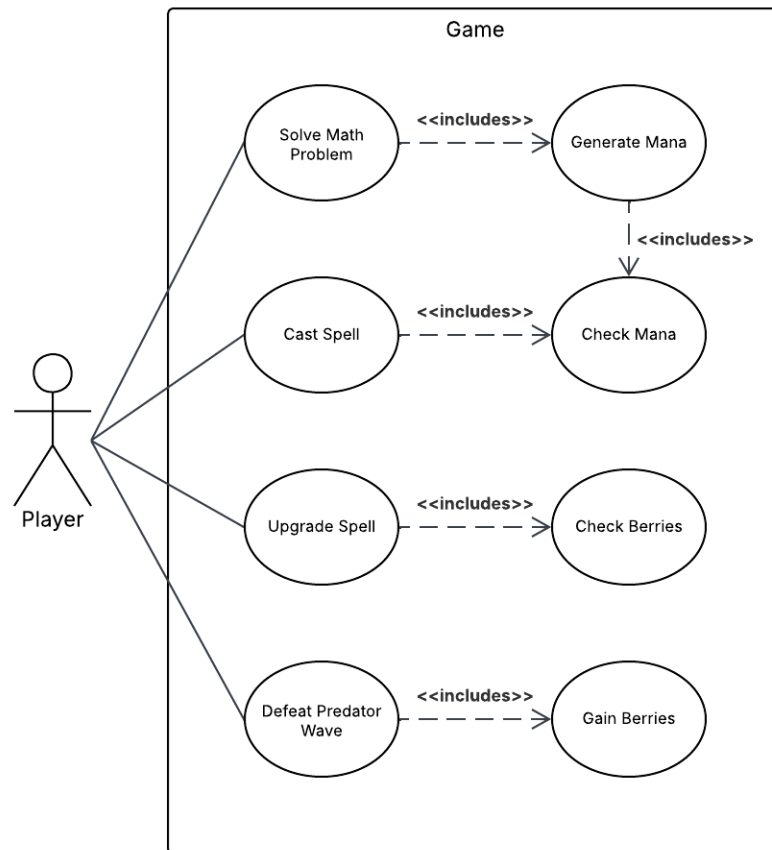


Figure 2 - Use Case diagram

Use Case Name:	Cast Spell
Actors:	Player
Description:	The player spends mana to cast a spell at predators.
Type:	Primary
Includes:	Check Mana
Extends:	None
Cross-refs:	Requirement 2.2
Uses cases:	Check Mana

Use Case Name:	Check Berries
Actors:	System
Description:	The system ensures that the player has enough berries to upgrade a spell.
Type:	Secondary
Includes:	None
Extends:	None
Cross-refs:	Requirement 2.5
Uses cases:	None

Use Case Name:	Check Mana
Actors:	System
Description:	The system ensures that the player has enough mana to cast a spell. In addition, the system ensures that the player does not exceed their maximum mana limit.
Type:	Secondary
Includes:	None
Extends:	None
Cross-refs:	Requirement 2.3
Uses cases:	None

Use Case Name:	Defeat Predator Wave
Actors:	Player
Description:	The player defeats a predator wave.
Type:	Primary
Includes:	Gain Berries

Extends:	None
Cross-refs:	Requirement 2
Uses cases:	Gain Berries

Use Case Name:	Gain Berries
Actors:	System
Description:	The system rewards the player with berries for defeating a predator wave.
Type:	Secondary
Includes:	None
Extends:	None
Cross-refs:	Requirement 2.5
Uses cases:	None

Use Case Name:	Generate Mana
Actors:	System
Description:	The system rewards the player with mana for solving math problems. In addition, the system supplies the player with a steady flow of mana.
Type:	Secondary
Includes:	Check Mana
Extends:	None
Cross-refs:	Requirement 2.3
Uses cases:	Check Mana

Use Case Name:	Solve Math Problem
Actors:	Player
Description:	The player is prompted to solve a math problem.
Type:	Primary
Includes:	Generate Mana
Extends:	None
Cross-refs:	Requirement 1
Uses cases:	Generate Mana

Use Case Name:	Upgrade Spell
Actors:	Player
Description:	The player spends berries to upgrade a spell.
Type:	Primary
Includes:	Check Berries
Extends:	None
Cross-refs:	Requirement 2.5
Uses cases:	Check Berries

4.2 Class Diagram

The following diagram shows information regarding the classes that make up Kiwi: The Mathematical Mage and the relationships between them.

Classes are represented by cards with two to three compartments. The top compartment holds the name of the class. The class name may be labeled with <<interface>> to indicate that the class is purely virtual. Alternatively, the class name may be italicized to indicate that the class is abstract. The compartment seen below the class name holds the class's list of attributes (this compartment may be omitted when the class has no attributes). Attribute names are prepended with a symbol that indicates its access modifier, where - indicates private access, + indicates public access, and / indicates derived access (meaning that the attribute is computed from other attributes). Following the attribute name is the type of the attribute, which includes types provided by the Godot game engine. The compartment below the class's attribute list holds the class's list of operations (this compartment may also be omitted if the class has no operations). Similarly to attributes, operation names may be prepended with a symbol indicating its access modifier. However, operations may also include a list of parameters within a pair of parentheses. The notation used for parameters is like the notation used for attributes, but without the access modifier. Additionally, the return type of an operation may be included after the closing parenthesis, though it may be omitted if the operation does not return anything. Operations may also be italicized to indicate that they are abstract and must be overridden by subclasses.

Relationships between classes are indicated by labeled lines and arrows, which indicate bidirectional and unidirectional relationships respectively. Additionally, multiplicities may be provided to show the number of participants on either side of the relationship. Arrows with open arrowheads indicate an association between the source class and the target class. Arrows with black diamond heads indicate that the target class is made of instances of the source class. Arrows with white diamond heads indicate that instances of the source class are parts of the target class. Arrows with white triangular heads indicate that the source class is a subclass of the target class, which may be either an abstract class or an interface. To make the distinction clearer, arrows with solid lines are used to indicate inheritance from an abstract class while arrows with dashed lines are used to indicate inheritance from an interface.

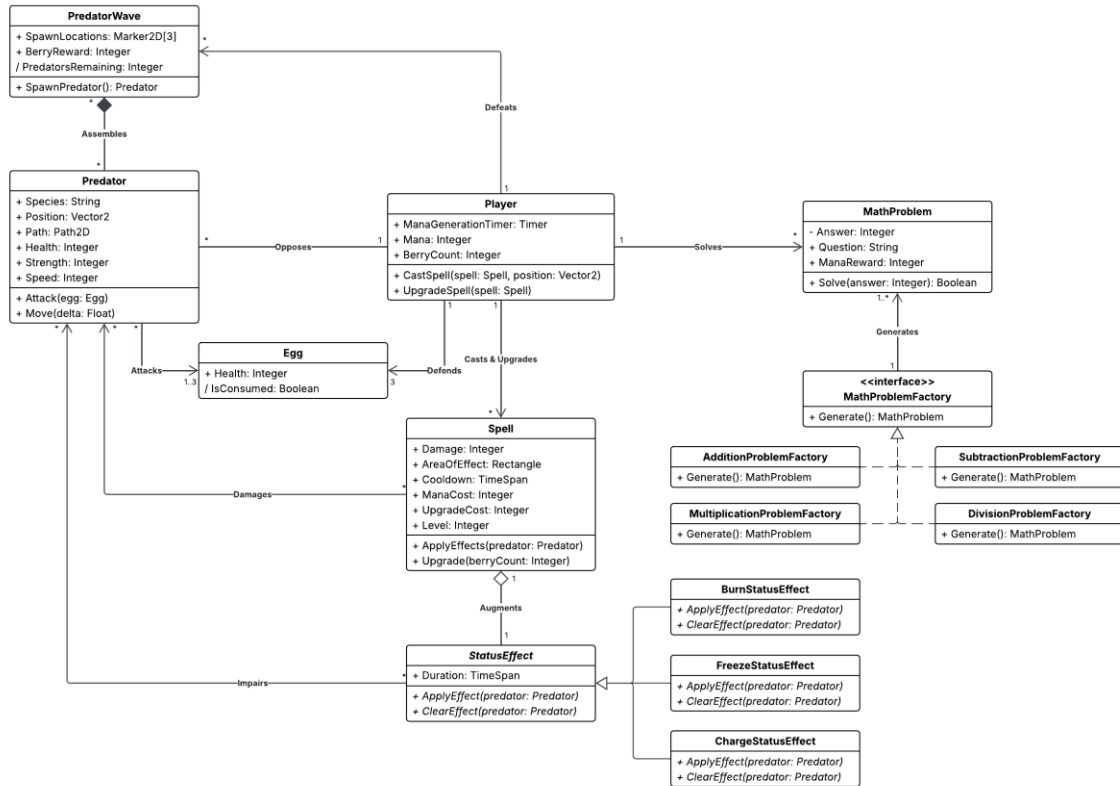


Figure 3 - Class Diagram

Element Name		Description
AdditionProblemFactory		Represents a factory that generates math problems involving addition.
Operations		
	+ Generate(): MathProblem	Generates an instance of the MathProblem class that represents an addition problem.
Relationships	Inherits from the MathProblemFactory interface.	
UML Extensions	None	

Element Name		Description
BurnStatusEffect		Represents a status effect that causes inflicted predators to receive additional damage over time.
Operations		

	+ <i>ApplyEffect(predator: Predator)</i>	Applies the Burn status effect to the specified predator.
	+ <i>ClearEffect(predator: Predator)</i>	Clears the Burn status effect from the specified predator.
Relationships	Inherits from the StatusEffect abstract class.	
UML Extensions	None	

Element Name		Description
ChargeStatusEffect		Represents a status effect that causes inflicted predators to generate sparks that damage nearby predators.
Operations		
	+ <i>ApplyEffect(predator: Predator)</i>	Applies the Charge status effect to the specified predator.
	+ <i>ClearEffect(predator: Predator)</i>	Clears the Charge status effect from the specified predator.
Relationships	Inherits from the StatusEffect abstract class.	
UML Extensions	None	

Element Name		Description
DivisionProblemFactory		Represents a factory that generates math problems involving division.
Operations		
	+ Generate(): MathProblem	Generates an instance of the MathProblem class that represents a division problem.
Relationships	Inherits from the MathProblemFactory interface.	
UML Extensions	None	

Element Name		Description
Egg		Represents an egg that the player is defending.
Attributes		

	+ Health: Integer	Stores the amount of health the egg has. When the egg is attacked by a predator, this attribute will be reduced. When this attribute is reduced to zero, the egg is consumed.
	/ IsConsumed: Boolean	Evaluates to true when the egg's health is 0. Evaluates to false otherwise.
Relationships	One to three Egg instances may be attacked by any number of Predator instances and three Egg instances may be defended by one Player instance.	
UML Extensions	None	

Element Name		Description
FreezeStatusEffect		Represents a status effect that inflicts predators with an effect that stops them in their tracks.
Operations		
	+ <i>ApplyEffect(predator: Predator)</i>	Applies the Freeze status effect to the specified predator.
	+ <i>ClearEffect(predator: Predator)</i>	Clears the Freeze status effect from the specified predator.
Relationships	Inherits from the StatusEffect abstract class.	
UML Extensions	None	

Element Name		Description
MathProblem		Represents a math problem that the player can solve to generate mana.
Attributes		
	- Answer: Integer	Stores the answer to the math problem.
	+ ManaReward: Integer	Stores the amount of mana that is rewarded to the player when the math problem is solved.
	+ Question: String	Stores the prompt that is shown to the player.
Operations		

	+ Solve(answer: Integer): Boolean	Returns true when the specified answer solves the math problem. Returns false otherwise.
Relationships	One to many MathProblem instances may be generated by one MathProblemFactory instance. Any number of MathProblem instances may be solved by one Player instance.	
UML Extensions	None	

Element Name		Description
MathProblemFactory		Provides an operation that allows specific kinds of math problems to be generated.
Operations		
	+ Generate(): MathProblem	Generates an instance of the MathProblem class.
Relationships	One MathProblemFactory instance generates one to many MathProblem instances. Subclasses may decide on what the MathProblem instances represent.	
UML Extensions	None	

Element Name		Description
MultiplicationProblemFactory		Represents a factory that generates math problems involving multiplication.
Operations		
	+ Generate(): MathProblem	Generates an instance of the MathProblem class that represents a multiplication problem.
Relationships	Inherits from the MathProblemFactory interface.	
UML Extensions	None	

Element Name		Description
Player		Stores attributes and operations the player has access to in-game.
Attributes		

	+ BerryCount: Integer	Stores the number of berries the player has earned.
	+ Mana: Integer	Stores the amount of mana the player has generated.
	+ ManaGenerationTimer: Timer	Stores a timer that determines the interval at which the player passively generates mana.
Operations		
	+ CastSpell(spell: Spell, position: Vector2)	Consumes the player's mana and casts the specified spell at the specified position where the spell will apply its effects. This operation will terminate early if the player does not have enough mana to cast the spell or if the spell is on cooldown.
	+ UpgradeSpell(spell: Spell)	Consumes the player's berries to upgrade the specified spell. This operation will terminate early if the player does not have enough berries to upgrade the spell.
Relationships	One Player instance may solve any number of MathProblem instances, oppose any number of Predator instances, defeat any number of PredatorWave instances, defend three Egg instances, and cast/upgrade any number of Spell instances.	
UML Extensions	None	

Element Name		Description
Predator		Represents a predator that seeks out and consumes the player's eggs.
Attributes		
	+ Health: Integer	Stores the amount of health the predator has. When the predator is damaged by a spell, this attribute will be reduced. If this attribute is reduced to 0, it is eliminated.
	+ Path: Path2D	Stores the path the predator will travel to reach the player's eggs.
	+ Position: Vector2	Stores the position of the predator.
	+ Species: String	Stores what species the predator belongs to.

	+ Speed: Integer	Stores the speed at which the predator will travel along its selected path.
	+ Strength: Integer	Stores the strength of the predator's attacks while damaging the player's eggs.
Operations		
	+ Attack(egg: Egg)	Attacks the specified egg when the predator has reached the end of its selected path.
	+ Move(delta: Float)	Moves the predator along its selected by the specified number of units.
Relationships	Any number of Predator instances may assemble into any number of PredatorWave instances, attack one to three Egg instances, oppose one Player instance, be damaged by any number of Spell instances, and impaired by any number of StatusEffect instances.	
UML Extensions	None	

Element Name		Description
PredatorWave		Represents a group of predators whose goal is to consume the player's eggs.
Attributes		
	+ BerryReward: Integer	Stores the number of berries that are rewarded to the player when the predator wave is defeated.
	/ PredatorsRemaining: Integer	Computes the number of predators remaining in the predator wave.
	+ SpawnLocations: Marker2D[3]	Stores the locations of three cavern entrances at which predators spawn from.
Operations		
	+ SpawnPredator(): Predator	Spawns a predator at a random spawn location and returns an instance representing that predator.
Relationships	Any number of PredatorWave instances may be assembled from any number of Predator instances and be defeated by one Player instance.	
UML Extensions	None	

Element Name		Description
Spell		Represents a spell that the player can cast to defeat predators.
Attributes		
	+ AreaOfEffect: Rectangle	Stores the area of effect in which predators are damaged and potentially inflicted with a status effect by the spell.
	+ Cooldown: TimeSpan	Stores the amount of time the spell will be placed on cooldown after it has been cast.
	+ Damage: Integer	Stores the amount of damage the spell will deal to predators.
	+ Level: Integer	Stores the level of the spell, which is increment when the spell is upgraded.
	+ ManaCost: Integer	Stores the amount of mana required to cast the spell.
	+ UpgradeCost: Integer	Stores the number of berries required to upgrade the spell.
Operations		
	+ ApplyEffects(predator: Predator)	Damages the specified predator and inflicts it with a status effect.
	+ Upgrade(berryCount: Integer)	Decreases the upgrade cost of the spell by the specified berry count. When the upgrade cost is reduced to 0 or below, the spell's level will be incremented and its effectiveness will be enhanced. Any excessive berries used will be applied to the upgrade cost(s) of the next level(s).
Relationships	Any number of Spell instances may be casted/upgraded by one Player instance and damage any number of Predator instances. One Spell instance may be augmented by one StatusEffect instance.	
UML Extensions	None	

Element Name		Description
StatusEffect		Represents an abstract status effect that impairs predators.
Attributes		

	+ Duration: TimeSpan	Stores the duration of the status effect when inflicted on a predator.
Operations		
	+ <i>ApplyEffect(predator: Predator)</i>	When overridden by a subclass, applies the status effect to the specified predator.
	+ <i>ClearEffect(predator: Predator)</i>	When overridden by a subclass, clears the status effect from the specified predator.
Relationships	One StatusEffect instance may augment one Spell instance. Any number of StatusEffect instances may impair any number of Predator instances. Subclasses may decide how to impair a predator and how to reverse the effects.	
UML Extensions	None	

Element Name		Description
SubtractionProblemFactory		Represents a factory that generates math problems involving subtraction.
Operations		
	+ Generate(): MathProblem	Generates an instance of the MathProblem class that represents a subtraction problem.
Relationships	Inherits from the MathProblemFactory interface.	
UML Extensions	None	

4.3 Sequence Diagrams

The following sequence diagrams show two scenarios involved with the core gameplay loop of Kiwi: The Mathematical Mage. Objects are represented by labeled boxes. Actors are represented by labeled stick figures, though the only actor in the diagrams is the player themselves. The lifetimes of objects and actors are represented by lifelines, which are dashed lines that stretch vertically. The lifetimes of some objects may be terminated at a certain point, which is marked by an X. Objects may enter periods of activity during their lifetime, which are represented by solid blocks on the active object's lifeline. These blocks may be nested within another block to indicate the execution of a sub-activity. Messages between objects are represented by arrows connected to their corresponding lifelines, which may spur periods of activity for the receiving object. Synchronous messages are represented by arrows with closed arrowheads while asynchronous messages are represented by arrows with open arrowheads. Sender messages are represented by arrows with solid lines while return messages are represented by dashed lines. Some sections of

the diagrams are enclosed in labeled frames, which represent scopes guarded by conditions. Frames labeled with Option indicate that operations within the scope will be executed when its condition is true. Frames labeled with Loop indicate that operations within the scope will be executed repeatedly so long as its condition is true.

4.3.1 Cast Spell Sequence Diagram

The following sequence diagram shows the interactions between the player and other objects when the player casts a spell. In this scenario, the game controller waits for the player to select a spell. Once the spell is selected, the game controller will compare the player's mana and the selected spell's mana cost to determine whether they can cast it. If they can cast the spell, the game controller will reduce the player's mana to cover the cost of the spell and go forward with obtaining the spell's area of effect. Afterwards, the game controller will query the predator wave to determine which predators will be affected by the spell's area of effect. The spell's effects will then be applied to every affected predator, inflicting them with damage and a status effect should they survive. If the predator was defeated by the spell, the game controller will remove it from the predator wave. These operations will be repeated until every predator has been defeated.

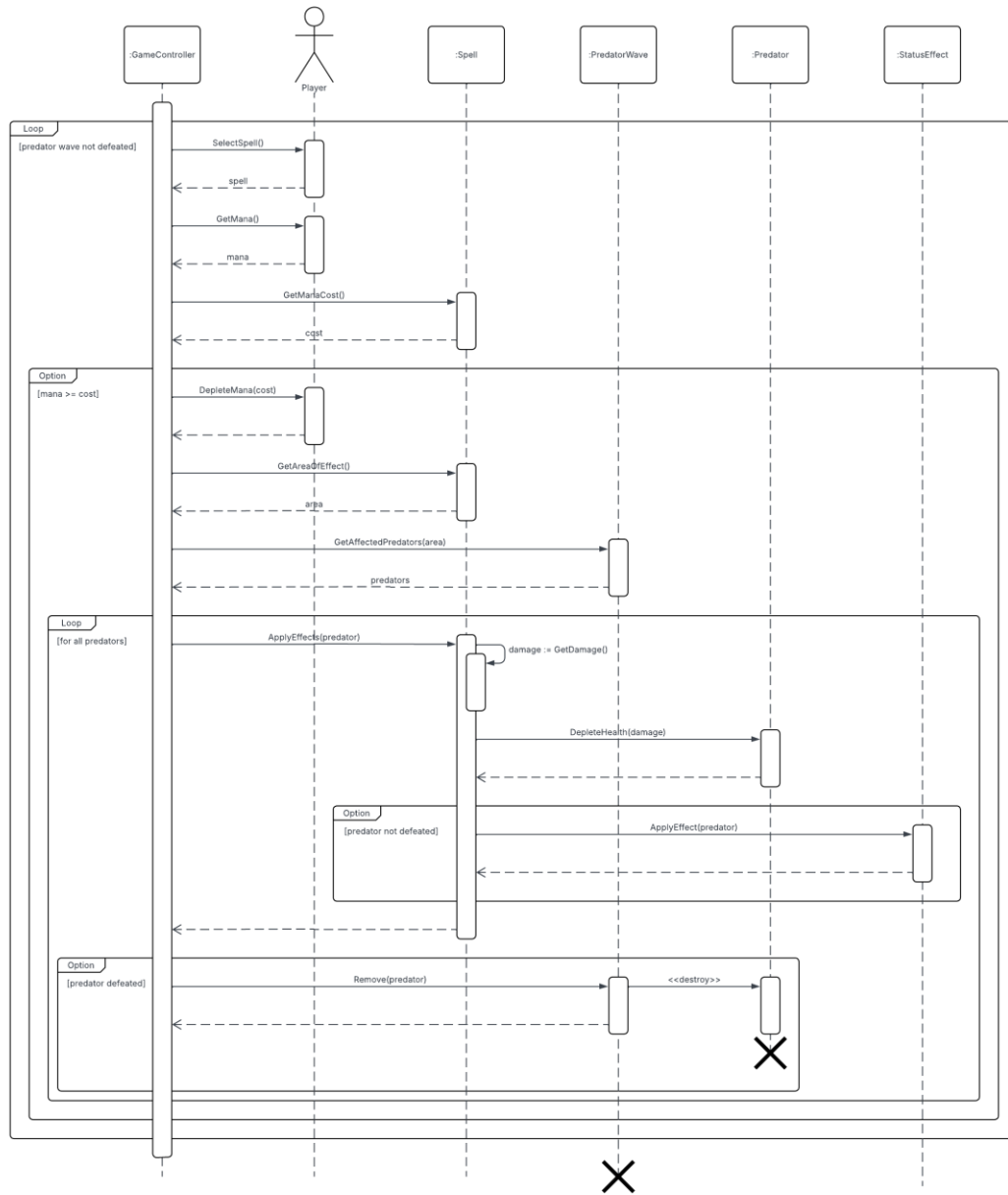


Figure 4 - Cast Spell Sequence Diagram

4.3.2 Generate Mana Sequence Diagram

The following sequence diagram shows the interactions between the player and other objects when the player generates mana. In this scenario, the game controller will supply the player with a fixed amount of mana every second. Meanwhile, the game controller will call upon the math problem factory to generate a math problem for the player to solve. The game controller will then prompt the player to solve the problem and perform background work until the player has entered an answer. If the player has answered the problem correctly, the game controller will be notified of this event and obtain the mana reward from the math problem before removing it. The game controller will then supply the player with the mana reward. If there is at least one predator remaining, the game controller will

call upon the math problem factory to generate another math problem and repeat these operations.

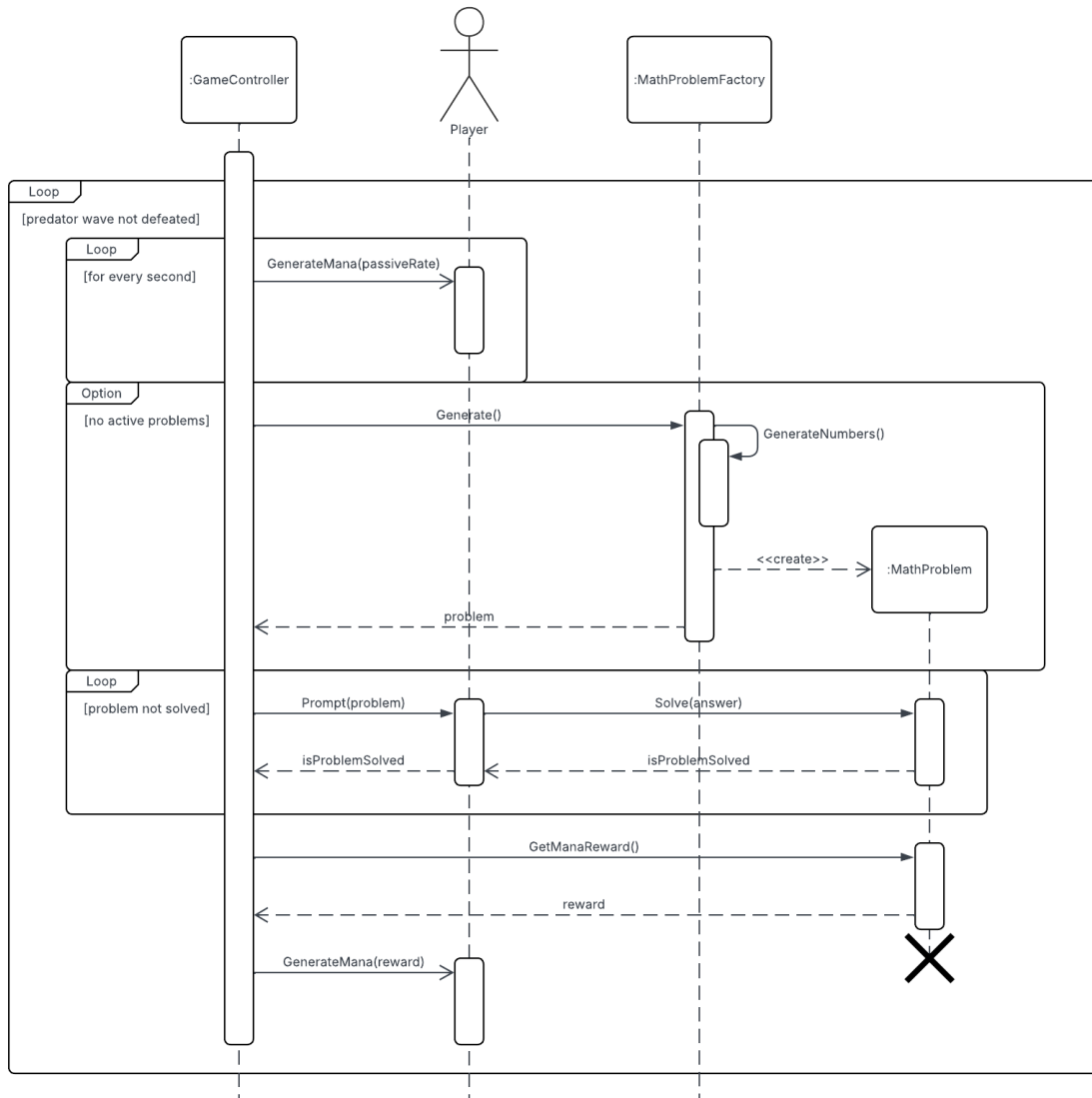


Figure 5 - Generate Mana Sequence Diagram

4.4 State Diagrams

The following state diagrams show the behavior of a couple key components of Kiwi: The Mathematical Mage. State entry points are represented by a filled circle. State exit points are represented by a smaller filled circle encompassed by a hollow circle. States are represented by labeled rectangles with rounded edges. State transitions are represented by labeled arrows, which may be annotated with guards to indicate conditional transitions. States with branching guarded transitions have a single transition to a choice pseudo-state, which are represented by diamonds.

4.4.1 Game State Diagram

The following state diagram shows the behavior of the game itself. When the game is executed, it will enter the main menu screen. From there, the player has the option to select what kind of math problems they want to practice and play the game, open the options menu, or exit the game. In the options menu, the player can set the volume of the audio and confirm their changes. Once they click on the play button in the main menu, the game will initiate a predator wave and enter the main gameplay loop when complete. During an ongoing predator wave, the player can enter the pause menu by clicking on the pause button. From the pause menu, the player can either resume the game or quit the game (which forfeits the ongoing wave). When the game is resumed, the player can generate mana passively over time and by solving math problems. The player can then use the mana they've generated to cast spells at predators and defeat them. Should the player fail to protect the eggs from the predators, the game will end, show their score, and go back to the main menu. When all predators within a predator wave have been defeated, the game will open a menu that allows the player to upgrade their spells. Once they are ready, the player may start the next wave and continue fighting the predators or quit the game.

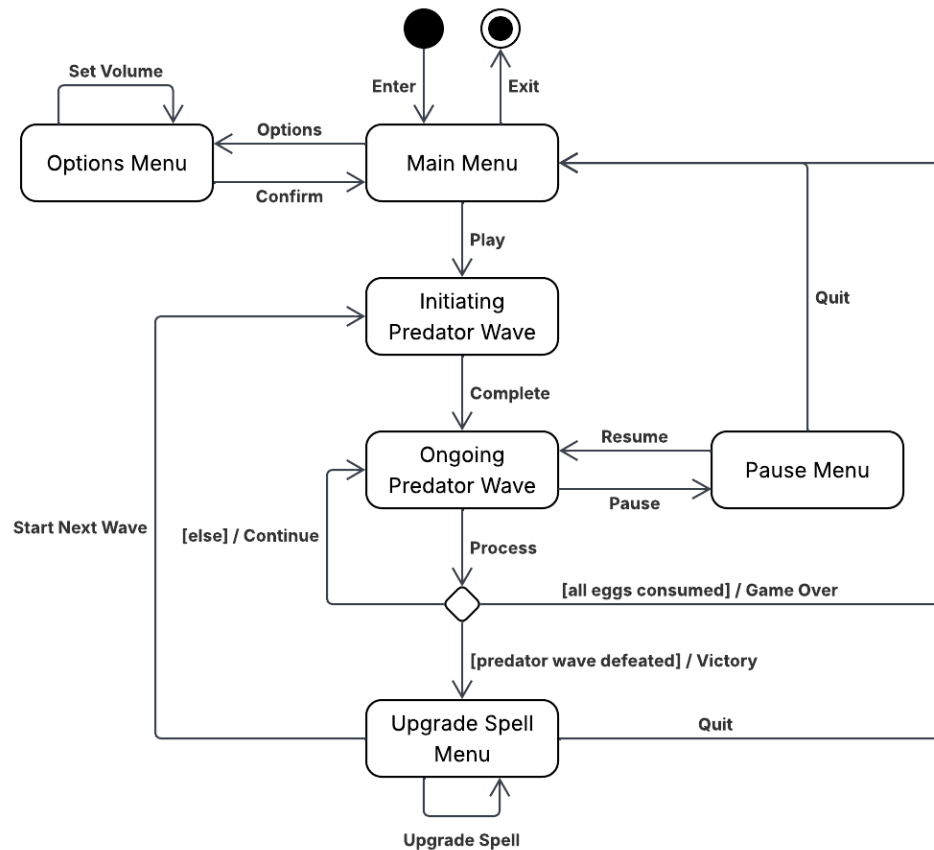


Figure 6 - Game State Diagram

4.4.2 Predator State Diagram

The following state diagram shows the behavior of a predator when it has spawned during a predator wave. After a predator is spawned, they will select a random path that

will lead them to the clutch of eggs the player is defending. Once the path has been selected, the predator will move along their chosen path until they have reached the eggs. From there, the predator will begin chomping on the eggs, reducing their health. The predator will continue chomping on the eggs periodically until all the eggs have been consumed. Any spawned predators will follow this behavior until they have been defeated by a player's spell or claimed victory by eating all the eggs.

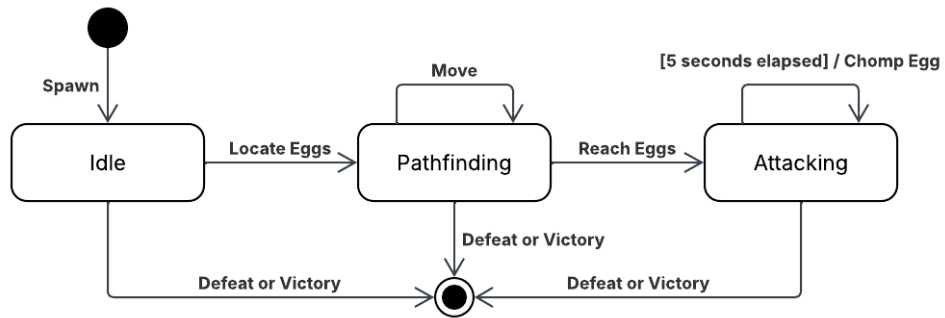


Figure 7 - Predator State Diagram

5 Prototype

The first part of the prototype is the main menu screen. On the main menu screen there is an option to quit the game, start the game, and options to bring up options for the game. There is also select problem style where the player can select the math topic of what they want to do. They can choose addition, subtraction, multiplication, and division. Once the player hits the start button, it will bring them to the game screen. On the game screen, the player can see the upgrades button, start wave button, buttons for the spells, and egg health and mana. Currently the spells do not work, and neither does the mana generation since the prototype is just the UI. Also, the predators do not spawn yet.

5.1 How to Run Prototype

To run the prototype, you will need to download the Godot IDE, then clone the repository from the Kiwi: Mathematical Mage repository on GitHub from the given link. Once you have this, open the project in Godot and then you have two options you can either press F5 to run the program or press the play button in the top right corner.

5.2 Sample Scenarios

The following screenshot shows the main menu of the game. It will allow the user to start the game, open the options menu and quit the game. The main menu also allows the user to select a problem style such as addition, subtraction, multiplication, and division.

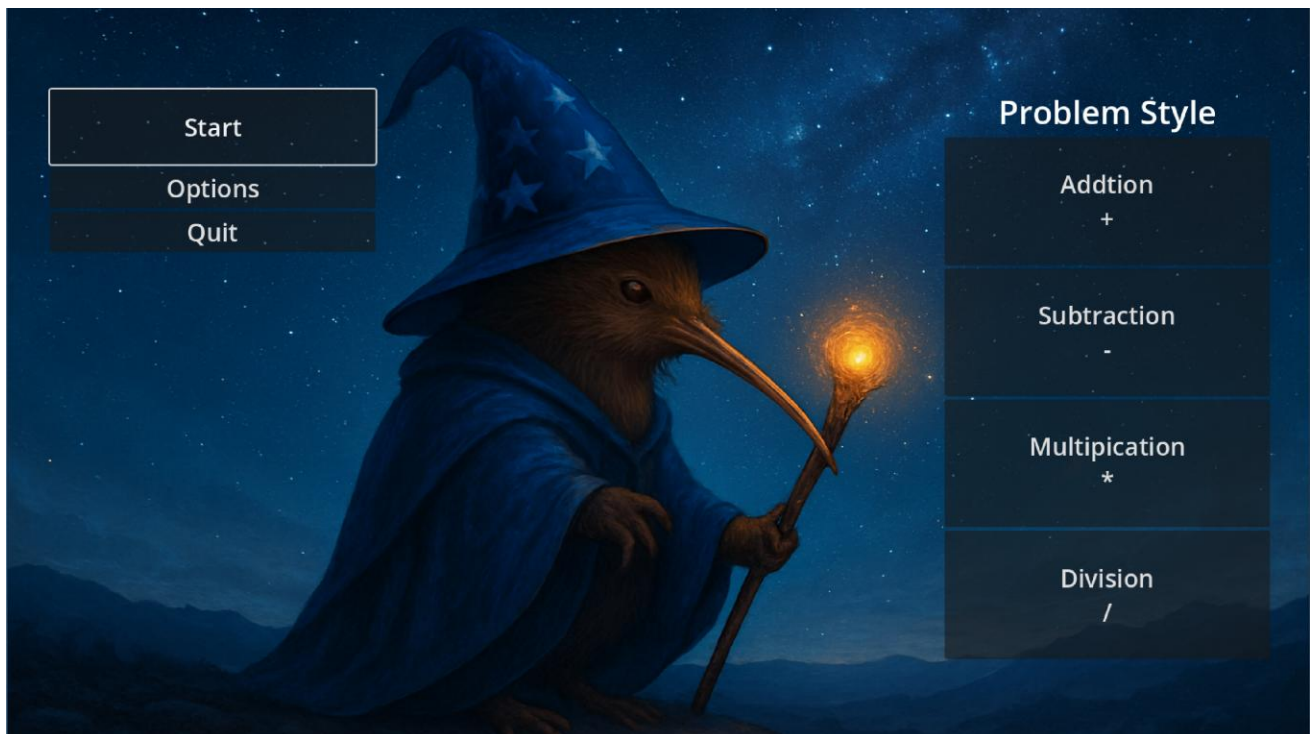


Figure 8 - Main Menu

The following screenshot shows the main game scene. This is where the entire game takes place, the user can cast spells, start the wave, open the upgrade menu, and answer

math questions to earn mana. Other information that is displayed is the wave number, egg health and Mana amount.

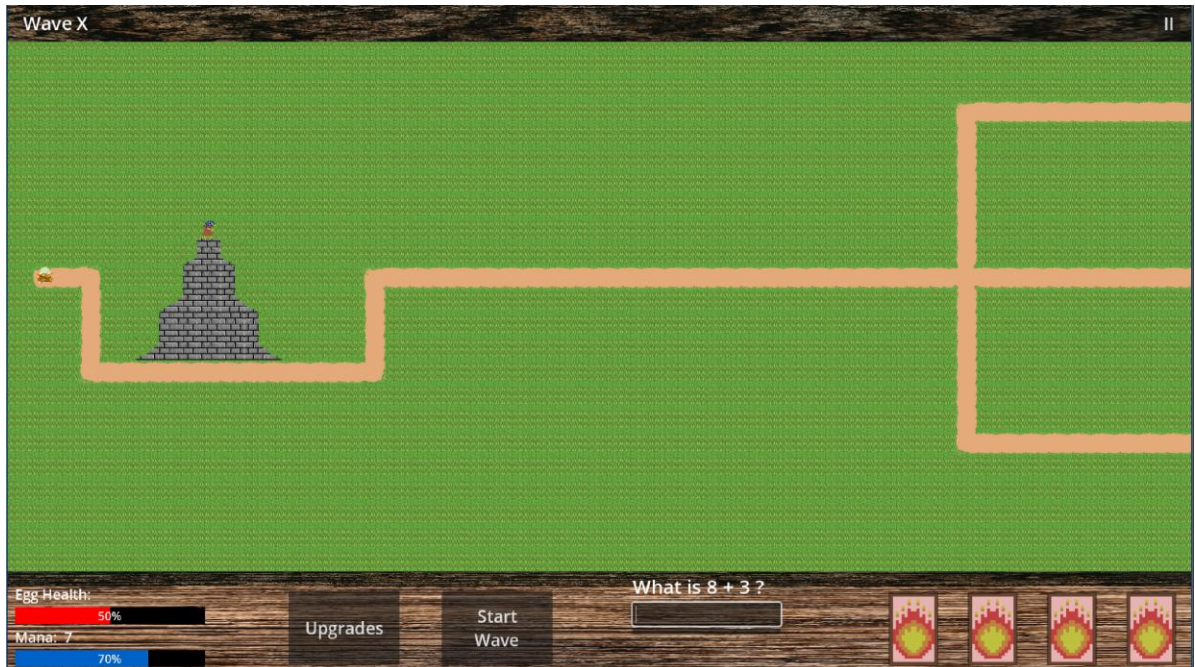


Figure 9 - Main Game Scene

The following screenshot shows the upgrade spell menu. This is where the user can select upgrades for each of the four spells using berries.

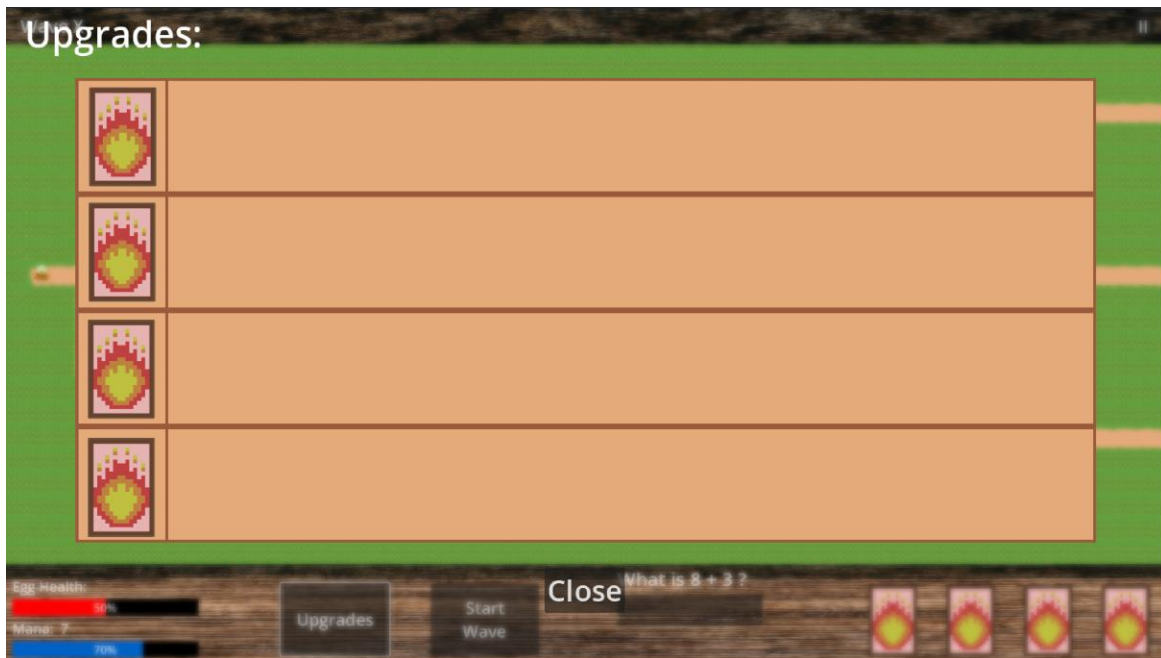


Figure 10 - Upgrade Spell Menu

The following screenshot shows the pause menu. This is where the player can choose to resume the game, go into the options menu, or quit to the main menu.

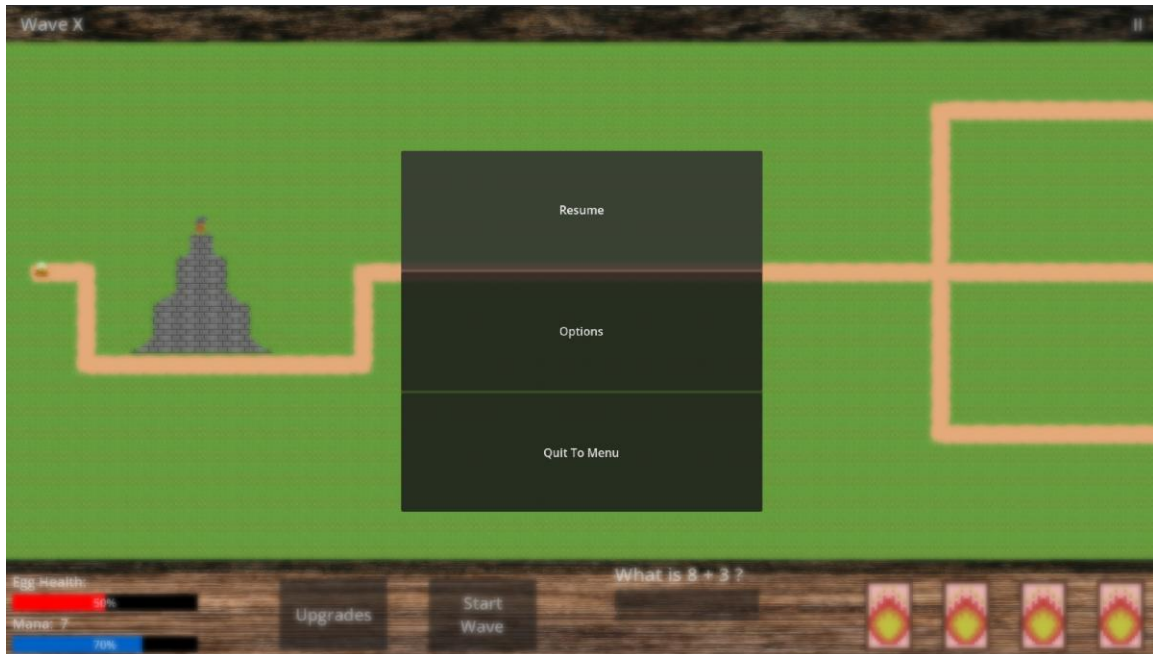


Figure 11 - Pause Menu

6 References

- [1] “Corestandards.” *Common Core State Standard For Mathematics*, 2023, corestandards.org/wp-content/uploads/2023/09/Math_Standards1.pdf.
- [2] Massachusetts Mathematics Curriculum Framework — 2017, <https://www.doe.mass.edu/frameworks/math/2017-06.pdf>.
- [3] A. Zammer, R. Miller, K. Gardiner, L. Grippo, and R. Mercado, “Kiwi: The Mathematical Mage,” *Kiwi: The Mathematical Mage*, Nov. 12, 2025. <https://swe-group3.github.io/KiwiTheMathematicalMageWebsite/>.

7 Point of Contact

For further information regarding this document and project, please contact **Prof. Daly** at University of Massachusetts Lowell (james_daly at.uml.edu). All materials in this document have been sanitized for proprietary data. The students and the instructor gratefully acknowledge the participation of our industrial collaborators.