Tower Defense iOS Game

Master's Project Report

William Elmes

Student ID: 806166732

Advisor: Dr. Michael Shafae

Reviewer: Dr. Mikhail Gofman

CPSC 597-11 (13137)
Spring 2016

Department of Computer Science
California State University, Fullerton

May 18, 2016

Abstract

In this project, I have created a Tower Defense game as a mobile application. My goal was not to use an existing game engine to help create the game, but to design the game engine and architecture all myself from the ground up. I chose a Tower Defense game because I am familiar with the design of such games and it is well-suited to be a mobile application. I decided to program my game as an iOS application using Xcode, the Swift programming language, and the SpriteKit framework. The key challenges of this project were creating a flexible game engine that could easily be expanded with additional features, designing a finite state machine that would handle user input and interface with the game engine's various components, and writing a parser that would be able to load game levels from text documents.

Keyword List

The following is a list of keywords that can be used in a database to help faculty and students locate projects addressing specific topics:

OS X

Xcode

iOS

Touch Screen

Swift

SpriteKit

Game Development

Game Design

Game Programming

Mobile Game

Video Game

Tower Defense

User Interface

Sprite Animation

2D Graphics

File Parser

Finite State Machine

Pathfinding Algorithm

Table of Contents

Introduction

Problem Description

For my Master's Project, I have developed a video game as a mobile application. This game is in the style of a Tower Defense game, which is well-suited to mobile devices both in terms of performance and functionality. First, I will clarify some concepts regarding what this style of game entails and why I chose it.

Figure 1: GemCraft - A screen shot of an example Tower Defense game.

One of the most basic concepts that needs to be defined is that of a Tower Defense game. A Tower Defense game is a simple strategy game in which the player tries to stop single-minded enemies from reaching a predetermined destination. The enemies simply follow the shortest path from their start location to their destination, and, in most scenarios, will not retaliate against attackers. The player's task is to destroy these enemies, and can be accomplished by constructing stationary Towers which will automatically attack nearby enemies. The player is given a static amount of health points at the beginning of a level, and each time an enemy reaches their destination, the player loses a number of hit points corresponding to that enemy. If too many enemies reach their destination, causing the player to reach zero health points, the player loses the level and may play again. By carefully choosing where to place Towers, the player can both increase the distance enemies have to traverse by forming a sort of maze or labyrinth in which the Towers make up the walls. Additionally, Tower

placement can increase the window of opportunity in which the Tower can attack enemies, due to the fact that towers have a fixed position and can only attack enemies that are within their attack range. By destroying enemies and completing rounds of a level, the player is rewarded with resources that can be used to construct additional towers or increase the effective power of existing towers. Each round of a level becomes increasingly difficult, consisting of enemies with both higher health points, armor value, movement speed, or even spawning in a shorter time frame. Higher amounts of health points means enemies will take more attacks to destroy. A higher armor value means that each attack deal less damage to the enemy. A higher movement speed will result in the enemy reaching the destination more quickly, allowing less time for Towers to attack. Finally, a shorter spawn window will result in enemies being able to progress farther, as Towers can usually only attack one enemy at a time even if there are multiple enemies within their attack range at the same time. Once all rounds of a level have been completed, the player is able to move on to the next level, which will clear the playing field of all of the player's Towers, resources, health points, and score, and present a new playing field to the player. Levels will begin with a lower difficulty level and few rounds, with little sensitivity to elements of strategy, however, later level will quickly escalate in difficulty and number of rounds, requiring a well-formed plan in order to survive each individual round of the level and ultimately win.

The next concept that needs to be clarified is why I chose for my project to be a mobile application rather than a PC or OS X application. While Tower Defense games are viable as traditional computer games, they are an ideal match for mobile devices in many ways. Most notably, mobile games are mostly played in brief, fragmented sessions. Tower Defense games are especially well-fitted to this niche, as a Tower Defense game is broken down into levels which are usually further broken down into rounds, granting the player frequent opportunities to put down the game and pick up where he or she left off at a later time with minimal interruption to game progress. Another reason why Tower Defense games work well as a mobile application is that user interaction on mobile devices works well with a Tower Defense user interface. Mobile applications typically utilize a touchscreen as the primary source of user input, and Tower Defense games tend to have very simple controls. Finally, Tower Defense games can work well as a mobile application due to their light performance requirements. A common art style that works well with Tower Defense games is two-dimensional sprite-based graphics, which many mobile devices can handle. With the mechanical simplicity and natural fragmentation of gameplay, a Tower Defense game is ideally suited for a mobile application.

With these concepts explained, I can lay out the objectives of my project and the activities I have undertaken to accomplish them.

Project Objectives

The overall objective of my project is straightforward. It is to create a Tower Defense game as a mobile application that will utilize two-dimensional graphics, have a flexible game engine that can easily be expanded with additional features, and is able to use a parser to load levels from external files. Additionally, my application had to be targeted at a specific demographic. Originally, I had other objectives that included implementing a profitable monetization model, however, the application is not yet ready to be published, and such goals became beyond the scope of my project.

The most prominent goals of my project have been achieved. The most basic of these goals was to create a Tower Defense game as a mobile application. M y application is a working Tower Defense game with a win and loss state that follows all of the traditional characteristics of a Tower Defense game, including a pathfinding algorithm to allow enemies to traverse the map from their spawn location to their destination, a finite state machine to handle user interactions via the touchscreen interface, and the ability to place Towers on the map that

will attack enemies. Furthermore, I used the SpriteKit framework in order to implement two-dimensional sprite graphics for my application. However, I did not use any other frameworks other than Apple's Foundation framework, allowing me to design my own custom game engine from the ground up. Finally, I created a parser that is able to read text files containing details for levels of my game in a specific format. This allows me to not only easily and quickly create new levels or modify existing levels, but also enables users to potentially create their own custom levels. This is also targeted at my key demographic of traditional gamers who have some spare time away from their computer or console. The difficulty level and various cues common in many video games also make my application a natural choice for this audience. While the application is not yet ready to be published, it is certainly meets the core requirements of my project.

Development Environment

For the development of my application, I used the Xcode IDE on a Macbook Pro in the Swift programming language with the Foundation and SpriteKit frameworks. Initially, I had planned on using the Objective-C programming language and the SpriteKit frameworks as I had some experience in developing iOS applications in Objective-C. However, thanks to the highly human-readable nature of Swift and some very helpful tutorial resources, I was able to learn the various differences between Swift and Objective-C very quickly, and found it much easier to use. Below is a table showing the specifics of my development environment.

Figure 2: Development Environment Specifications

Device	Macbook Pro	Model	9.2
Operating System	OS X El Capitan	Version	10.11.3
IDE	Xcode	Version	7.2
Programming Language	Swift	Version	2.1.1
Test Device	Apple Simulator	Version	9.2

Operational Environment

The target device of my application was originally going to be the iPhone. However, during the definition of the requirements of my project, I decided that the screen size of the iPhone alongside the touch screen interface would not be ideal for my game. In a Tower Defense game, the screen is broken up into a grid, which dictates where Towers can be place and how enemies will traverse the map. Each of these cells of the grid must be able to be individually selected without accidentally selecting neighboring adjacent cells. The screen of an iPhone was too small for this requirement, so I decided to instead target my application at the iPad. Even the smallest iPad model would have a screen size large enough to satisfy this requirement, so I chose the iPad Air 2 as my target device with the intention of being able to scale up the view size for iPad models with a larger screen at a later time. I tested my application using Apple's Simulator application. Below is a table that outlines the specifics of my target operational environment.

Figure 3: Operational Environment Specifications

Target Device	iPad Air 2		
Target Operating System	iOS	Version	9.2
Test Device	Apple Simulator	Version	9.2

Requirements Description

As outlined earlier, the goal of my project was to produce a Tower Defense game as a mobile application. This means that it must fulfill all of the requirements which define a Tower Defense game, while also being able to function as a mobile application and be usable by my target audience of people who have experience with video games. The most basic of these requirements include being able to utilize user input from a touch screen device, being able to run on a mobile device at an acceptable performance level, designating specific win and loss states within the game, implementing the various functions of a Tower Defense game, employ two-dimensional animated sprite graphics, and be able to parse an input file to load a level.

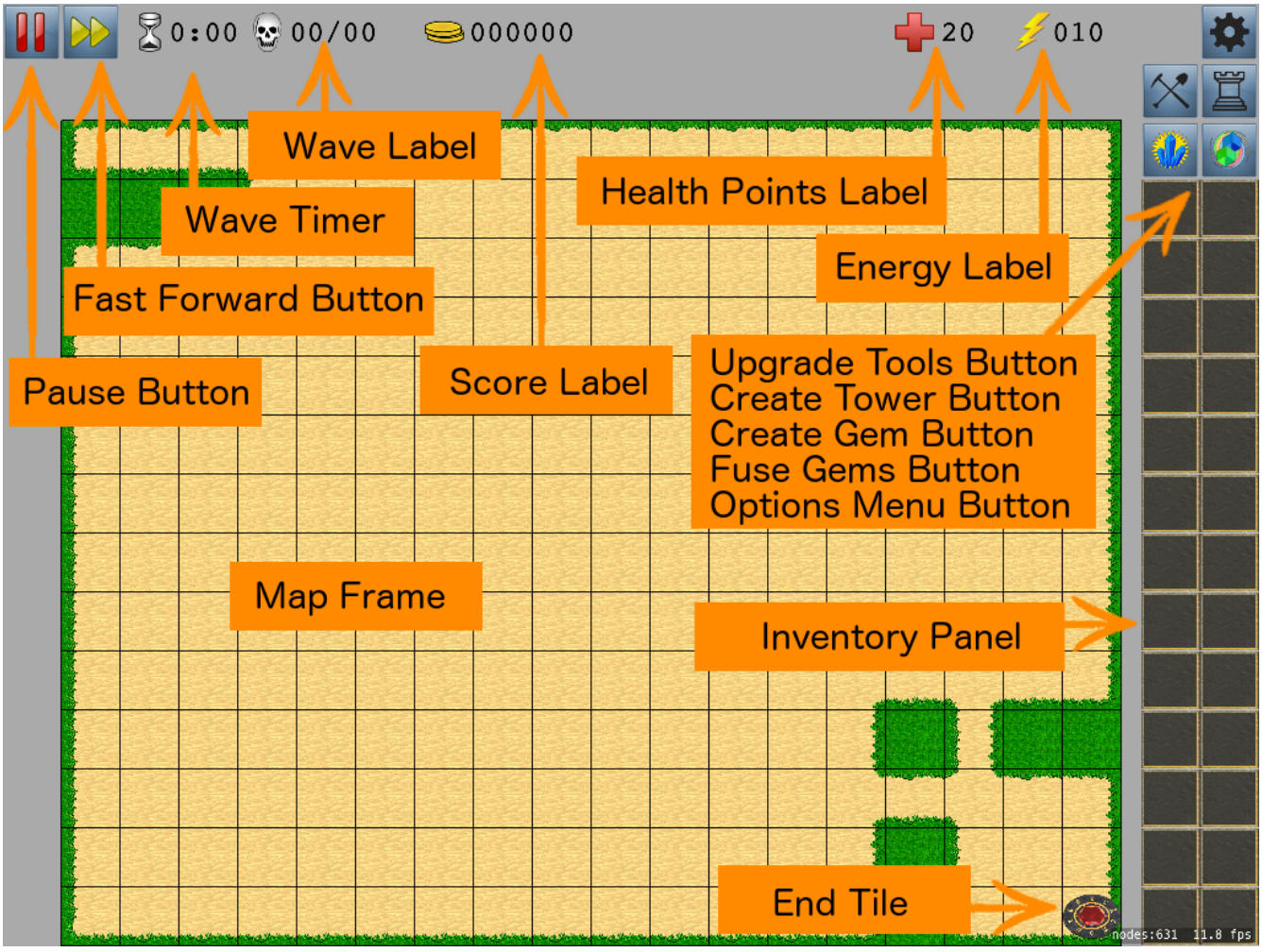

Figure 4: User Interface – A screen shot illustrating the basic user interface of my game.

The simple user interface of a Tower Defense game lends itself well to the touch screen inputs of many mobile devices. My game's user interface is broken up into three distinct parts which the user can interact with. First, there are several buttons across the top of the screen which can be pressed to give the user access to the functions of the game itself. Second is the map, which occupies most of the screen and is broken up into a grid of cells, each of which can be individually selected and interacted with. Last is the inventory, which contains the player's Gems when they are initially created and is also broken up into a grid of individual cells that can be selected. Each of these elements work well with a touch screen interface and give the user a clear idea of what is happening within the game.

The application must also be able to run well on a mobile device. Due to the use of low-resolution two-dimensional sprites in my project and the lack of a need for any complex calculations such as physics, my application runs well on its target device, the iPad Air 2. It requires far less memory and processing power than the target device, and can maintain an acceptable level of performance. However, despite the low performance requirements of my application, it still suffers from a low frame rate and longer than expected loading times. These issues may be addressable in the future.

Primarily, my application must be a Tower Defense game with a clearly defined win and loss state. I believe my project has achieved tremendous success in this area. A win and loss state are both clearly defined as either destroying all enemies without running out of health points, and letting too many enemies reach their destination causing you to reach zero health points, respectively. My game also embodies all of the requirements of a Tower Defense game. The player may place Towers on the map, which enemies are not able to traverse, forcing them to go around the Towers in order to reach their destination. Towers will automatically attack nearby enemies and destroy their target if its health points are reduced to zero. Enemies are created in groups, staggered over a short period of time. Enemies provide resources when destroyed, which can be used to create additional Towers or increase the power of existing Towers. Lastly, enemies are removed once they reach their destination, reducing the player's health points and eventually causing the player to lose. All of these are basic requirements for any Tower Defense game, and my game meets all of these criteria.

One of the most unexpectedly time-consuming parts of my project was obtaining and modifying legal-to-use art assets for the simple two-dimensional graphics of my game. With Swift and SpriteKit, implementing the graphics in a way that would easily allow me to modify or even switch out the graphical assets was fairly simple. SpriteKit allowed me to simple import each image or animation frame into my project as individual image files. Accessing these images for non-animated objects was very straight forward. For animated objects, SpriteKit automatically creates a sprite atlas from image files grouped together in the same atlas folder, and each image can be loaded into an array of textures which can be used as an animation sequence. However, obtaining these image files and modifying them for use in my application was very time-consuming. While I am satisfied with the result, it was very costly in terms of development time.

A requirement for my game which was not crucial, but added a great deal of value to the project overall, was a parser that could load levels from external text files. In my game, levels consist of several key attributes, including the level map, the tileset of the terrain, and the attributes of the rounds, the waves they consist of, and the groups of enemies they ultimately end up creating. My parser can read in all of these attributes, and allow the game engine to load a level in the game, allowing me to much more easily create and modify levels for my game. First, the parser loads in the game map, which is formatted as a block of characters. Each character in the block represents a tile on the map and whether that tile is pathable for enemy units or allows for a Tower to be built on that tile. Additionally, one tile must be designated as the start tile and another the end tile, and there must be a path connecting the two tiles. Next, the name of the tileset is loaded. The tileset is the group of image files which are used to draw the tiles of the map. Each tileset will have its own visual theme and this allows for a more interesting set of environments for the game, rather than a single static one. Finally, the number of rounds is read, and each round is then processed. Rounds are broken up into waves, which consist of a group of enemies. Each round indicates how many waves it contains and the corresponding waves. Each wave consists of how many enemies will be created over the course of the wave, the images that will be used to draw and animate those enemies, the interval between creating each enemy, the amount of time until the next wave is automatically started, and all of the attributes of the enemies, including health, armor, movement speed, and so on. Once the rounds are loaded, all of the data that has been read can be passed to the game engine and it will load the level. This system allows me great flexibility in designing new levels for my game, and even potentially allows players

to design their own custom levels. This trait is usually appreciated by my target audience of experienced gamers.

The final, and most difficult to achieve, set of requirements involve the game's usability for its target audience of experienced gamers. The user interface must be familiar and intuitive by being similar to that of other games. There must be an appropriate difficulty curve over the course of the game. Lastly, the game must have some degree of replayability to keep users engaged for a longer period of time.

These requirements have all been met to various degrees in my project. My game definitively meets the requirements of a Tower Defense game, however, usability is still a work in progress, but influenced the design of many different parts of the game. With further development, my game will be able to meet every aspect of these requirements.

Design Description

The design of my application revolves around a central game engine. This game engine is an object which interfaces with the SpriteKit Scene, which in turn interfaces with the application's View. This allows all internal interfaces to be controlled via a single object.

The game engine itself acts as a finite state machine in order to handle user input. For example, pressing the Create Tower button will put the game in a Create Tower state, which will allow the player to create a Tower on the map by selecting a cell. Pressing the Create Tower button again while in the Create Tower state will cause the game to exit that state. All user inputs are processed by the game engine based on the current state of the game. While the buttons themselves handle user input events, they, in turn, trigger a corresponding function of the game engine. The game engine is also tasked with initializing and storing all other components of the application. After its members have been initialized, the game engine invokes the parser to load the level from a text file. From there, the rest of the game's functionality is driven by timers and user input handled by the game engine.

After the level is loaded, the user may create Tower and Gems before starting the first round. As described above, pressing the Create Tower button allows map cells to be selected in order to create a Tower. Pressing the Create Gem button brings up an additional set of buttons overlaid on the inventory panel. From here, the player can select the type and rank of Gem they wish to create. Selecting a white colored Gem will have a reduced resource cost, but generate a randomly selected type of gem. After selected the rank of Gem, the Gem button panel will become hidden again, and the new Gem will be placed in the inventory panel. Selecting an inventory cell or a map cell while not in Create Tower mode will select the Gem located in that cell in order to allow the player to select a second cell to move the Gem to that cell. These events are all handled by the game engine's finite state machine.

The map is responsible for storing all of its cells and a list of all of the Towers within. The map uses these to calculate the pathing across every cell. The pathfinding algorithm starts at the end cell and works its way backwards one cell at a time. Every cell on the border of each cell being processed is added to the list of cells to be processed. As a cell is processed, it records its distance from the end cell and the direction to its parent cell which first added it to the list to be processed. Eventually, every cell that has a path to the end cell is processed, giving it a distance score and direction to the next cell in the path. This is used not only to guide enemies from the start cell to the end cell in the shortest path possible, but also allows the game to ensure that there remains a path from the start cell to the end cell, as that is a rule of the game itself.

The Tower themselves also utilize an algorithm for their attack behavior. In order for a Tower to attack an enemy, it must first acquire an enemy as a target that is within that Tower's attack range. While each Tower is not currently attacking, it continuously checks every active enemy to see if there are any enemies within attack range. If multiple potential targets are found, the Tower uses a prioritization function, which selects the enemy that has traveled the farthest distance. Once the Tower has a valid target, it may begin attacking. The Tower begins a recursive attack function, which first checks to make sure the target is still within in range and has not been destroyed, then launches its attack at the target, usually in the form of a projectile which will repeatedly move in small increments toward its target until they meet, when damage will be dealt to the enemy and the projectile is removed. After the attack is launch, the Tower waits for a duration corresponding to its attack rate, and then the function repeats itself. If the target is found to have become invalid at the start of the function, the Tower clears its target and enters back into the target search loop. This was the most processing power demanding part of the application. Every Tower is constantly either performing the target search loop or the

attack loop. While the attack loop is fairly light, the target search loop can be very demanding with a large number of Towers and enemies active.

The enemies themselves also interface with a large number of objects within the application. Each enemy must keep track of its attributes, including its health points, a list of Towers currently attacking that enemy, the wave object that created the enemy, in addition to any sprites anchored to the enemy from Tower attacks. Whenever an enemy is attacked by a Tower, it must check its remaining health points to see if it has been destroyed. Once an enemy is destroyed, it notifies all Towers attacking it and its parent wave. This allows the attacking Towers to clear their target and search for a new target, and the wave to keep track of how many enemies are currently active on the battlefield. If an enemy reaches its final destination, it will tell the game engine to deduct health points from the player and the enemy will be removed.

These interactions ultimately allow the game engine to keep track of what is happening within the game and to handle the appropriate response. In the next section, I will outline each of the functions and members of these objects.

Implementation

In this section, I will outline each of the objects used within my application, including their members, methods, and interfaces with other objects.

Object:	**Description:**
GameEngine	Created as a global variable by the GameScene, handles user input, and manages all other objects
Members:	**Description:**
UI Buttons	Button objects for game functionality
UI Icons	Images accompanying labels to track waves, enemies, and player attributes
UI Labels	Text accompanying icons to track waves, enemies, and player attributes
MapFrame	Provides a view of the battlefield
InventoryFrame	Stores Gems not placed in Towers
gameState	Tracks state changes triggered by button presses and how to manage following user input
selectedGemRank	Tracks the rank of Gem to be created while in Create Gem mode
selectedGemColor	Tracks the type of Gem to be created while in Create Gem mode
selectedMapCell	Tracks which map cell is selected while moving Gems
selectedInventorySlot	Tracks which inventory cell is selected while moving Gems
rounds	Stores a list of all rounds for the loaded level
timerCounter	Tracks the time remaining in the wave countdown timer
playerScore	Tracks the player's score attribute
playerHp	Tracks the player's health points attribute
playerEnergy	Tracks the player's energy resource attribute
Methods:	**Description:**
init()	Initializes the buttons, icons, and labels, then adds all members as children nodes so they may drawn on the screen within the application's view
InitUi()	Assigns functions to all user interface elements to allow them to notify the game engine of user input
LoadLevel(fileName) → Bool	Invokes the parser to load the designated level from the text file
StartNextWave()	Trigger the current round to begin creating enemies and the map to begin the target search loop for all Towers
Press Button Methods	Passed to the members to be triggered on user input to allow the game engine to handle a response

SelectMapCell(cell)	Select a map cell in order to move a Gem or create a Tower
SelectInventorySlot(slot)	Select an inventory cell in order to move a Gem
SwapGems(firstCell, secondCell)	Move the Gem contained in the first cell to the second cell and vice versa
CancelState()	Take the game out of its current state and return it to its default state
StartWaveTimer(time)	Begin the countdown timer until the next wave
TickWaveTimer()	Loop to update the countdown timer and check if it has expired
Set and Modify Attribute Methods	Change the value of the attribute and update its corresponding label, then handle any events triggered by the change in value

Object:	**Description:**
LevelParser	Collection of methods to extract level attributes from a text file in order to allow the game engine to load the level
Methods:	**Description:**
ParseLevel(fileName) → Bool	Parse the file until an attribute is found, then call the corresponding function to parse that attribute Returns whether it was successful in parsing the file after setting the game engine's level attributes if it was successful
ParseMap(lineArr, pmIndex) → (lineIndex, pathMap, buildMap, startCell, endCell)?	Continues parsing the file from the line it was called, reading in the pathability and buildability of each map cell and the start and end cells Returns the line of the file where it finished parsing the map attributes and returns those attributes or nil if it was unsuccessful
ParseTileset(wordArr, wIndex) → Tileset?	Continues parsing the line of text containing the name of the tileset Returns the value of the tileset or nil if it was unsuccessful
ParseStartAttributes(lineArr, startIndex) → (lineIndex, startHp, startEnergy)?	Continues parsing the file from the line it was called, reading in the starting attributes of the player Returns the line of the file where it finished parsing and the starting attributes of the player or nil if unsuccessful
ParseRounds(lineArr, rIndex) → (lineIndex, rounds)?	Continues parsing the file from the line it was called, reading in the number of rounds, the number of waves for each round, and calling ParseWave for each of those waves Returns the line of the file where it finished parsing and the list of rounds for the level or nil if unsuccessful
ParseWave(lineArr, wIndex) → (lineIndex, wave)?	Continues parsing the file from the line it was called, reading in the attributes of the wave and the enemies to be spawned by that wave Returns the line of the file where it finished parsing and the parsed wave or nil if unsuccessful
ParseNextWord(wordArr, wordIndex) → (wordIndex, word)?	Continues parsing the current line of text from the word where it was called, reading in the next word on that line Returns the index of the following word on the line and string value of the read word or nil if unsuccessful

Object:	**Description:**
EnemyWave	Stores attributes of enemies to be created, a list of active enemies, and gives Towers a list of enemies within their attack range
Members:	**Description:**
Enemy Attributes	Stores the attributes of enemies that are to be created
incomingEnemies	The number of enemies that have yet to be created
spawnInterval	The time delay between creating each enemy
wavePeriod	The maximum duration after a wave starts until the next wave is automatically started
Methods:	**Description:**
init(type, hp, armor, speed, scale, bounty, score, numberEnemies, interval, period)	Initializes the wave by storing enemy and wave attributes
StartSpawning(startCell)	Start the create enemy loop
SpawnEnemy(startCell)	Check if any more enemies need to be created, if so create an enemy at the start cell and begin its pathing loop, then if more enemies need to be created, loop after the spawnInterval until all enemies have been created
GetEnemiesInRange(tower) → [EnemyUnit]	Get a list of enemies from the wave's list of active enemies that are within the given Tower's attack range Returns a list of enemies
EnemyKilled(enemy)	Remove the killed enemy from the list of active enemies and notify the game engine to update the wave label

Object:	**Description:**
LevelRound	Manages the waves of a round
Members:	**Description:**
activeWaves	List of waves that are currently in their create enemies loop
incomingWaves	List of waves that have not begun creating enemies yet
Methods:	**Description:**
init(waves)	Adds given list of waves to incomingWaves
GetEnemiesInRange(tower) → [EnemyUnit]	Loops through each activeWaves to get a list of enemies that are within the given Tower's attack range from all active waves Returns a list of enemies
StartNextWave()	Takes the next wave from incomingWaves, adds it to activeWaves, begins the waves create enemies loop, and notifies the game engine to update the wave label and start the wave timer

Object:	**Description:**
ButtonPanel	Manages a group of buttons that can be hidden or shown Used to hide and show the Create Gem buttons
Members:	**Description:**
btns	The list of buttons in the panel
Methods:	**Description:**
Add(btn)	Adds the given button to the panel's list of button
Btn(i) → ButtonNode?	Returns the button from the list of buttons corresponding to the given index or nil if no button exists at that index
Hide()	Hide all buttons on the panel
Show()	Show all buttons on the panel

Object:	**Description:**
MapFrame	Manages the cells of the map, enemy pathfinding, and Towers
Members:	**Description:**
basePosition	The position from which cells of the map are offset
cells	The list of map cells in a two-dimensional array
startCell	The map cell from which enemy pathfinding starts
endCell	The map cell where enemy pathfinding ends
towers	The list of all Towers
End Cell Sprites	Visual indicators and animations for the end cell
Methods:	**Description:**
init()	Create the map cells, link each cell to its adjacent neighbors, draw the map grid, and initialize the end cell sprites
DrawGrid()	Draw the lines across the map separating the cells of the map vertically and horizontally
Load(tileset, pathMap, buildMap, start, end) → Bool	Reset each cell's tile type, recalculate each cell's tile type, set each tile's sprite texture, and draw the end cell sprites Returns whether or not the map was able to find a path on the map
SetPathing(startCoords, endCoords) → Bool	Set the start and end cells, calculate the actual distance between the center of each cell and start and end cell, then update pathing Returns whether the UpdatePathing method was successful
UpdatePathing() → Bool	Reset the pathing of all cells, then run the pathfinding algorithm Start at the end cell, add all adjacent cells of given cells to a list, recording each cell's parent and incrementing distance the parent Returns whether the CheckPathing method could find a path

CheckPathing(testCell) → Bool	Runs a simple version of the pathfinding algorithm to detect if the start cell has a path to the end cell by checking if the start cell is ever processed Returns whether a path is found from the start to end cell
EnemyEndAnimation()	Trigger the animation on the end cell for when an enemy reaches their destination
SetPressedAction(action)	Assigns the method to be called when a map cell is pressed to each map cell
CreateTower(cell)	Check if a Tower can be created on the given cell and create a Tower on that cell and update the map's pathing if so
AddTower(tower)	Add a Tower to the map's list of Towers
StartTowerTargetSearch()	Begin the Tower target search loop for all Towers

Object:	**Description:**
MapCellNode	Cells within the map grid
Members:	**Description:**
Adjacent Cells	References to each adjacent cell in the map grid
pathable	Whether enemies can traverse this cell
buildable	Whether a Tower can be built on this cell
startDist	Actual distance from this cell to the start cell
endDist	Actual distance from this cell to the end cell
tileType	Index of which texture this cell's sprite will use
parentCell	Reference to the parent cell of this cell along the path from start to end cell
pathDir	The direction from this cell to its parent cell
pathDist	The number of cell along the path from this cell to the end cell
tower	A reference to this cell's Tower, nil if the cell has no Tower
AddTower	A handle to a function to be called when a Tower is created
PressedAction	A handle to a function to be called when this cell is pressed by the user
Cell Sprites	The sprites of the cell that draw its textures
Methods:	**Description:**
init(highlightTexture, selectedTexture)	Initializes the map cell and sets the textures for when the cell is highlighted and selected
touchesBegan(touches, withEvent)	Handles when the user begins a touch on the cell
touchesMoved(touches, withEvent)	Handles when a touch within the cell moves

touchesEnded(touches, withEvent)	Handles when a touch that started in the cell ends
Depress()	Highlight the cell as it's being pressed
Release()	Remove the highlight from the cell as a press is released
Press()	Trigger the cell's PressedAction when a press completes
CreateTower()	Create a Tower at the cell and notify the MapFrame that a Tower has been created
SetTile(tileset)	Set the texture of the cell's sprite based on where it falls along the edges of the path
Set Specific Tile Methods	Checks if the cell falls within a specific tile type and sets its texture
==(lhs, rhs) → Bool	Overloaded equality comparison operator
<(lhs, rhs) → Bool	Overloaded less than comparison operator

Object:	**Description:**
TowerUnit	Holds a Gem object in order to search enemies for a valid target and attack its target
Members:	**Description:**
gem	Gem object held by the Tower, nil if no Gem is present
isAttacking	Whether the Tower is currently running its attack loop
isSearching	Whether the Tower is currently running its target search loop
targetEnemy	The currently targeted enemy which the Tower is attacking
unitSprite	The Tower's sprite which draws its texture
Methods:	**Description:**
init()	Initializes the Tower's sprite
SearchForTarget()	Recursively loops to search the current round for a valid target
FireWeapon()	Recursively loops to check if the current target is still valid and launch attacks at that target
StopTargetSearch()	Halt the target search loop
TargetIsInRange(target) → Bool	Check if the given target enemy is still within attack range of the Tower
ClearTarget()	Clear the Tower's current target without directly halting the attack loop
TargetKilled()	Notify the Tower that an enemy has been killed, and clear the Tower's target if that enemy was its target

Object:	**Description:**
InventoryFrame	Manages inventory slot cells
Members:	**Description:**
basePos	Top left corner of the inventory frame from where inventory cells will be drawn downward and to the right
slots	List of all inventory cells in a two-dimensional array
Methods:	**Description:**
init()	Create all inventory cells at the proper locations
GetOpenSlot()	Get the first inventory slot which does not contain a Gem
CreateGem(color, rank)	Create a Gem of the given type and rank in the first available inventory slot
SetPressedAction(action)	Assign the action that will be triggered when an inventory cell is selected to each inventory cell

Object:	**Description:**
InventorySlotNode	Inventory cell of the inventory frame, holds a Gem object and can be interacted with by the user
Members:	**Description:**
gem	Reference to a Gem object contained in the cell, nil if not present
Cell Textures	SpriteKit texture objects of the images used for the cell's face
Cell Sprites	Sprite objects that manage the cell's textures
pressedAction	Reference to a function which is invoked when the cell is selected
Methods:	**Description:**
init(slotTexture, slotActiveTexture, selectedTexture)	Initializes the cell's textures and sprites
touchesBegan(touches, withEvent)	Handles when the user begins a touch on the cell
touchesMoved(touches, withEvent)	Handles when a touch which began on the cell moves
touchesEnd(touches, withEvent)	Handles when a touch which began on the cell ends
Depress()	Show the cell texture for when the cell is being pressed
Release()	Show the cell texture for when the cell is idle
Press()	Trigger the cell's pressed action when the cell is selected

Object:	**Description:**
GemItem	Object which enables a Tower to attack and stores the attributes of the attack based on the type and rank of the Gem
Members:	**Description:**
color	The type of the Gem or its color
rank	The rank of the Gem, dictating its overall power
charges	Tracks the number of attacks the Gem is able to launch, used for yellow Gems which generate attack charges slower than they can launch attacks
itemSprite	Manages and draws the texture for the Gem
launchSprites	A list of sprites which draw the animation for the launch effect of attacks
projectileSprites	A list of sprites which draw the projectile for attacks
Methods:	**Description:**
init(gemColor, gemRank)	Initializes the Gem's type, rank, and sprites
SetColor(gemColor)	Sets the Gem's type
SetRank(gemRank)	Sets the Gem's rank
RandomizeColor()	Sets the Gem's type to a random value
RandomizeRank(upgradeLevel)	Sets the Gem's rank to a random rank which has been unlocked by the current tools upgrade level
GetDamage() → Double	Get the amount of damage which this Gem deals with an attack
GetRange() → CGFloat	Get the attack range of the Gem
GetAttackPeriod() → Double	Get the period the Gem must wait in between launching attacks
AcquireTarget(enemies) → EnemyUnit	Select the highest priority target from a list of valid enemy targets
LaunchAttack(tower)	Launch an attack from the given Tower at that Tower's target enemy
SetTexture()	Update the Gem's sprite based on its current texture after the Gem's color or rank has been changed
RedAttackLaunch(tower)	Launches a red Gem style attack from a given Tower at that Tower's target enemy, including damaging all enemy targets within attack range of the Tower, drawing an impact effect on all affected enemies, and drawing an attack animation on the Tower
BlueAttackLaunch(tower)	Launches a blue Gem style attack from a given Tower at that Tower's target enemy, including creating a projectile and starting its move loop toward the target and drawing an attack launch animation on the Tower
BlueAttackImpact(tower, target, projectile)	Inflict a blue Gem style attack on the given target from the given projectile, including applying the Tower Gem's slowing effect and damage to the target, and drawing the impact effect

GreenAttackLaunch(tower)	Launches a green Gem style attack from the Tower at its target enemy, including creating a projectile and starting its move loop toward the target and drawing an attack launch animation
GreenAttackImpact(tower, target, projectile)	Inflict a green Gem style attack on the given target from the given projectile, including applying the Tower Gem's poison effect and damage to the target, and drawing an attack impact effect on the target enemy
YellowAttackLaunch(tower)	Launches a yellow Gem style attack from a given Tower at that Tower's target enemy, including creating a projectile and starting its move loop toward the target, tracking the number of remaining attack charges for the Gem, starting the attack charge replenish loop if not yet started, and drawing an attack launch animation on the Tower
YellowAttackImpact(tower, target, projectile)	Inflict a yellow Gem style attack on the given target from the given projectile, including applying the Tower Gem's damage to the target, and drawing an attack impact effect on the target enemy
MoveProjectile(tower, target, projectile, flightPeriod, impactEffect)	Move the given projectile a short distance toward its target over the given period repeatedly until the projectile impacts with its target, rotating the projectile to face toward its target with each increment, and triggering the given impact effect function on impact
StandardTargetPrioritization(enemies) → EnemyUnit	Select the enemy from a list of valid enemy targets with the highest value for distance traveled
SlowedTargetPrioritization(enemies) → EnemyUnit	Select the enemy from a list of valid enemy targets with the highest distance traveled and that does not have an active slow effect if possible
PoisonedTargetPrioritization(enemies) → EnemyUnit	Select the enemy from a list of valid enemy targets with the highest distance traveled and that does not have an active poison effect if possible
LoadLaunchAnimation(color) → [SKTexture]	Load the array of textures which will be used for the launch effect of attacks from Gems of the given type
LoadProjectilAnimation(color) → [SKTexture]	Load the array of textures which will be used for the projectile of attacks from Gems of the given type
LoadImpactAnimation(color) → [SKTexture]	Load the array of textures which will be used for the impact effect of attacks from Gems of the given type
LoadEffectAnimation(color) → [SKTexture]?	Load the array of textures which will be used for the status effect of attacks from Gems of the given type; only blue and green Gems have a status effect, so other Gem types return nil

Object:	**Description:**
EnemyUnit	Manages an enemy as it is created, traverses the map, and is attacked by Towers
Members:	**Description:**
unitType	The type of the enemy as defined by enumeration
currAnim	The current animation the enemy is performing
facingDir	The facing direction of the current animation
mapCell	The map cell which the enemy is currently pathing toward
travelDist	The distance along the path the enemy has traveled since it was created
frameTextures	The array of frame textures used in the enemy's current animation
actionQueue	The list of move actions that dictates the path the enemy will take across the cells of the map
unitSprite	Manages the enemy's sprite and animations
Enemy Attributes	Tracks the enemy's various attributes, including health points, armor, movement speed, and so on
slows	A list of active slow effects applied by Tower attacks which reduce the movement speed of the enemy while active
poisonTicks	A list of instances of damage applied by Tower attack poisoning effects which periodically inflict damage to the enemy
attackedBy	A lot of Towers which are currently targeting the enemy unit, so the enemy can notify the Towers to clear their target when the enemy is destroyed
parentWave	A reference to the wave object that created the enemy, so the enemy can notify the wave when it is destroyed
impactSprites	A list of sprites which manage the impact effect of attacks targeting the enemy
Effect Sprites	Manage the status effect animations of attack effects currently active on the enemy
Methods:	**Description:**
init(wave, enemyType, hp, armor, speed, scale, bounty, score)	Initializes the enemy by setting its attributes, giving it a reference to the wave that created it, and initializing its sprite
BeginPathing(startCell)	Begin the enemy's walking animation, move the enemy to the start cell, and start the enemy's move loop
Move()	Move the enemy in short increments toward the parent cell of the enemy's current map cell until the parent cell is reached, loop until the end cell is reached
SetAnim(newAnim, repeatAnim, newFacingDir)	Change the enemy's current animation

ReachEnd()	Notify the game map that an enemy has reached its destination, then clear and remove the enemy unit
Kill(despawnDelay)	Trigger the enemy's death animation, notify the parent wave and Towers targeting this enemy that it has been destroyed, and remove the enemy
Despawn()	Remove the enemy from the game
GetAttacked(attacker)	Add the attacker Tower to the enemy's list of Towers that it is being attacked by
OutrangedAttacker(attacker)	Notify the enemy that is has left the attack range of a Tower that was attacking it and remove that Tower from the attacked by list
TakeDamage(amount, armorBypass)	Inflict damage to the enemy, reduced by the enemy's armor proportional to the given armor bypass, and check if the enemy has been destroyed
ProjectDamage(amount, armorBypass)	Notify the enemy that a projectile has been launched at it and the amount of damage it will inflict on impact, allowing Towers to not attack enemies that already have a lethal attacked launched at them
UpdateMovementSpeed()	Change the enemy's movement speed based on its base movement speed and any active slows on the enemy
ApplySlow(slow, duration)	Add a slowing status effect to the enemy, reducing its movement speed by the given amount for the given duration and drawing a status effect on the enemy
ApplyPoison(dmg, numTicks)	Add a poison status effect to the enemy, periodically inflicting a instance of damage equal to the given amount for the given number of times and drawing a status effect on the enemy
TickPoison()	Inflict the next instance of poison damage to the enemy from its list of poison ticks, and stop the poison tick loop once all ticks are exhausted, removing the status effect sprite

Object:	**Description:**
ButtonNode	Manages a button user interface element and notifies the game engine when pressed
Members:	**Description:**
Button Textures	Stores the image textures that are drawn by the sprites
isDisable	Whether the button can be interacted with by the user
hasAltButtonState	Whether the button should switch to an alternate texture and pressed function when pressed and vice versa
currState	Which state the button is in, only used if the button has an alternate state
Button Sprites	Manages the button's textures and draws them

Press Actions	Reference to functions which are invoked when the button is pressed based on its current state
Methods:	**Description:**
init(buttonTexture, buttonDepressedTexture, buttonDisabledTexture)	Initializes the button's textures and sprites, and enables user interaction for the button
init(buttonTexture, buttonDepressedTexture, buttonDisabledTexture, buttonTextureAlt, buttonDepressedTextureAlt, buttonDisabledTextureAlt)	Overloaded intializer for buttons with an alternate state
touchesBegan(touches, withEvent)	Handles when the user begins a touch within the button
touchesMoved(touches, withEvent)	Handles when the user moves a touch which began within the button
touchesEnded(touches, withEvent)	Handles when a touch ends which began within the button
Depress()	Change the button's texture to its depressed texture
Release()	Change the button's texture to its idle texture
Press()	Trigger the button's pressed action when it is successfully pressed
ToggleState()	Manually change the buttons state, such as when the game engine's state is canceled
ChangeTextures(buttonTexture, buttonDepressedTexture, buttonDisabledTexture)	Changes the textures of the button and updates its sprites, used in the select Gem type and rank button panel

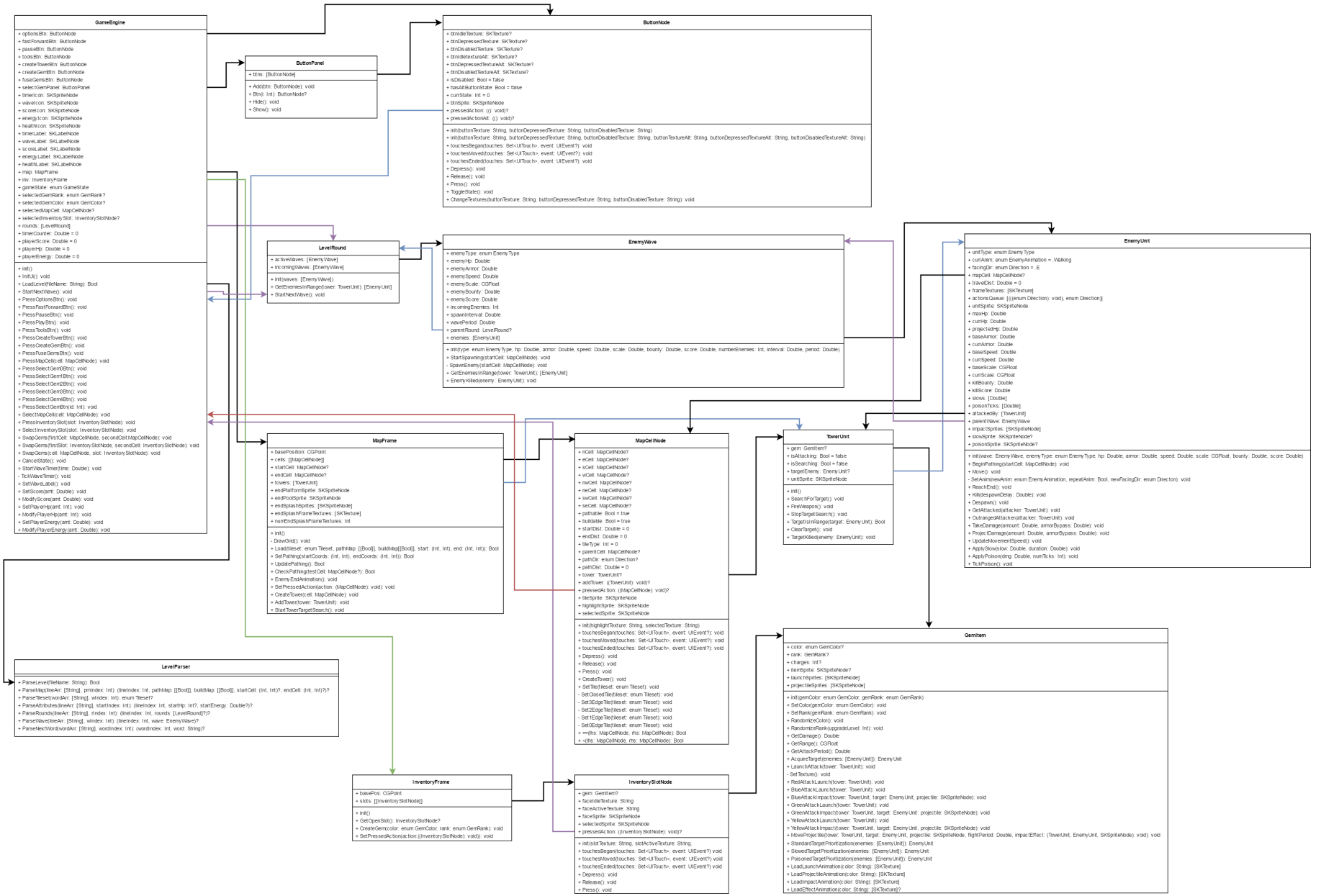

Figure 5: Class Diagram

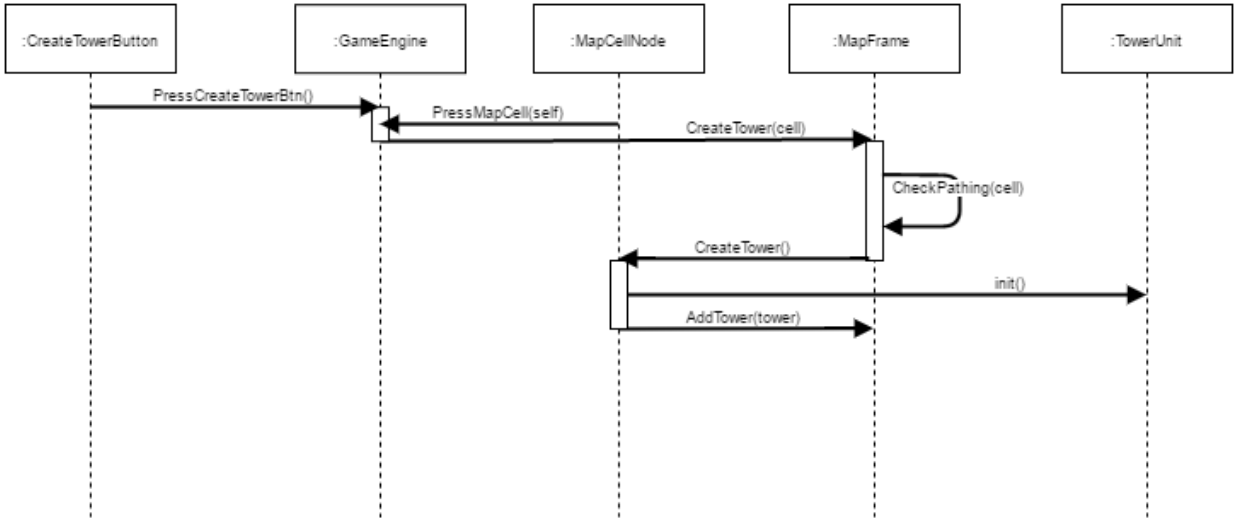

Figure 6: Sequence Diagram – Create Tower

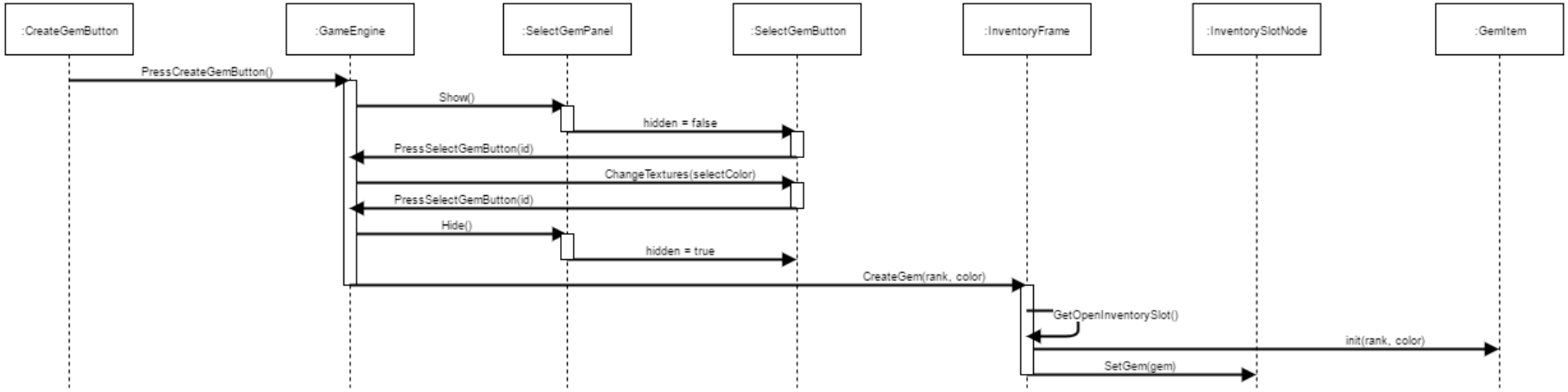

Figure 7: Sequence Diagram – Create Gem

Test and Integration

The testing of my application was done using Apple's Simulator application. The Xcode IDE uses the Simulator application as its default method of running applications targeted at another device. Given the nature of game development, and the ability to generate visual feedback, this made testing very easy. For any asset loading operations, I used debug messages to confirm whether they were successful, as the images they are loading cannot produce visual feedback until they are actually loaded.

Testing the level parser was the most difficult part of testing for my project. I wanted my parser to be case-insensitive, ignore blank lines and superfluous white space, and to be as insensitive to loading order as possible. I am satisfied with the results, and have achieved all of these goals. I used many different test level files to ensure that each level successfully loaded every attribute, regardless of the case of characters, any number of empty lines or lack thereof, any white space or lack thereof, and various load orders, as long as attributes that rely on another attribute are loaded after their dependencies. For example, rounds must be loaded in the order in which they occur in the level, and each round must be followed by all of its waves.

Overall, the testing of my application was heavily involved as a part of the development of the code and architecture of the application itself. Simple debug messages or visual inspection of the application's view while running was sufficient in testing.

Due to the limited scope of my project, integration only involves ensuring that all components are able to interface properly with one another. No external interfaces are currently used, but they will prove to be useful in future development of the application.

Installation Instructions

My project is currently only targeted to run on iPad Air 2 devices and has not yet been published. Therefore, in order to run the application, it must first be loaded in Xcode and can be launched from there. An appropriate device may be connected in order to allow the application to run on that device, or the application can be run using Apple's Simulator application.

Operating Instructions

My application functions as a simple Tower Defense game. The goal of the game is to destroy all enemies before they reach their destination, as indicated by the round platform located on one of the tiles on the map. The main game screen is broken up into several components.

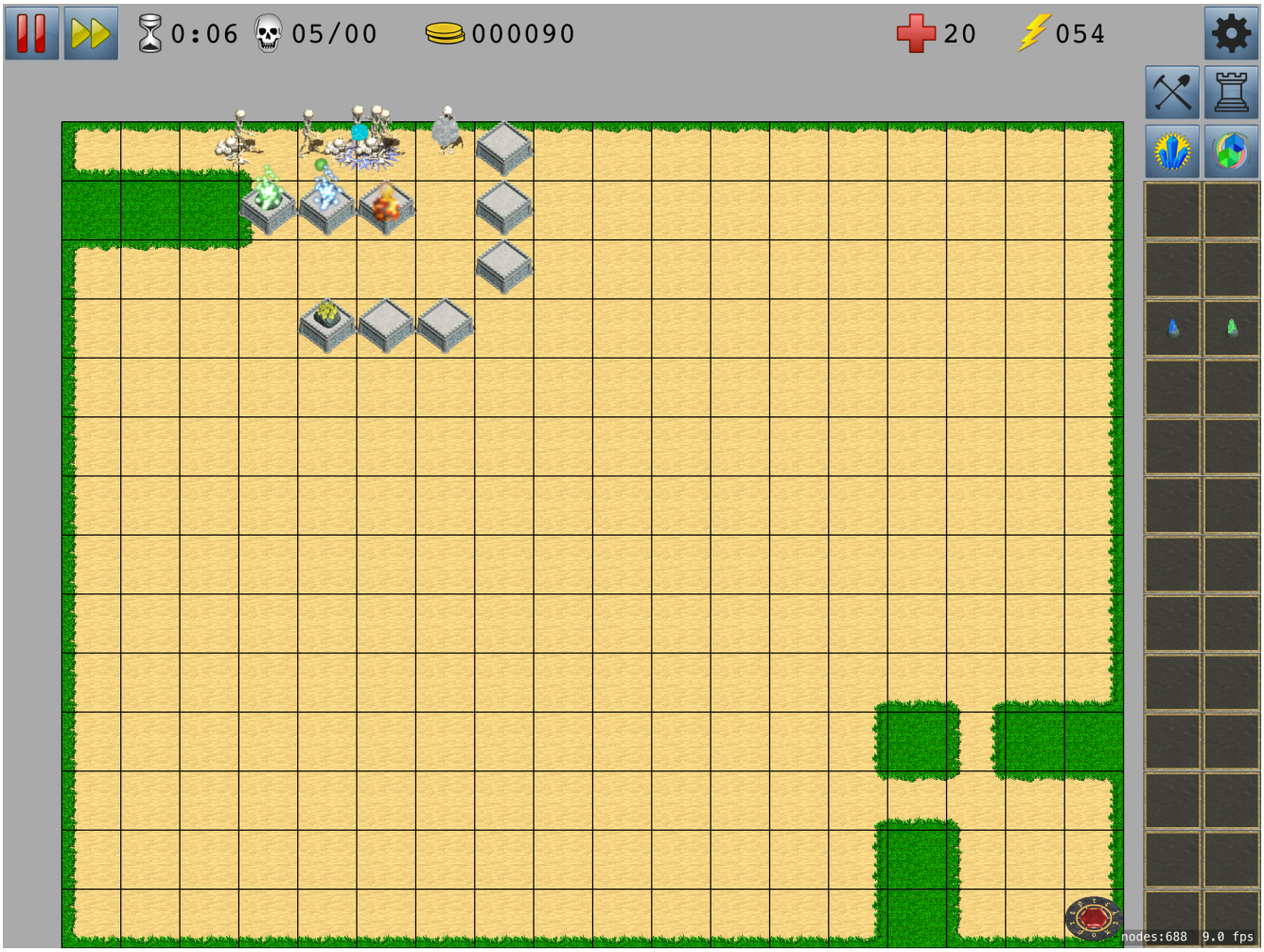

Figure 8: Game Play – A screen shot of my game being played.

Along the top of the screen are buttons, labels, and icons which show you the attributes of the game. The fast forward button in the top left will begin the next round. This will be covered more later. To the right of the fast forward button are the labels and icons. From left to right, there is the timer icon and label, which indicates the amount of time remaining before the next wave will automatically start. Next is the wave icon and label, which shows the number of enemies currently active on the map and the number of enemies that are in the process of being created by all of the current waves. To the right of that we have the score label and icon, which shows the player's score for that level which is gained by destroying enemies, completing rounds, and starting waves early. To the right more, there is the player health points label and icon, which shows how many hit points the player has remaining. If this value reaches zero, the player loses. The player loses health points whenever an enemy reaches its destination. Lastly, there is the energy icon and label, which shows how many resources the player has available to them to spend on creating Towers, creating Gems, upgrading tools, and fusing Gems into

more powerful versions. The player is able to gain energy by destroying enemies and completing rounds. Unlike score, energy is consumed by the player, and without sufficient energy, the player cannot perform many tasks.

The next major part of the main game screen is the map frame. This area is broken up into a grid of squares, which each represent a cell of the map. Enemies can move horizontally or vertically from one tile to another. Enemies will always follow the shortest path from the tile where they were created to their destination. The player may also construct a Tower on these cells, which will prevent enemies from being able to traverse that tile. However, the player must always leave a path from the enemies' starting tile to the end tile. Additionally, enemies may only traverse open tiles, as indicated by sandy ground on the default sand tileset. Towers on the map will automatically attack enemies that are within that Tower's attack range if that Tower is equipped with a Gem. This brings us to the next interface component.

The inventory panel is located along the right edge of the screen. Like the map frame, it is broken up into a grid of squares representing inventory slots which may contain a Gem. Once a Gem is created, it will be placed in the first open inventory slot. A Gem can be selected from the inventory panel by selecting it inventory slot. Then, another inventory slot or map cell containing a Tower can be selected, and the Gems contained in those two cells will switch places, or simply move the first Gem if the second cell does not contain a Gem. Doing this with a Tower will equip that Tower with the Gem, allowing it to attack enemies based on the Gem's attack attributes. The different types of Gems have different attack behaviors, and high ranks of Gems have more powerful attacks.

Lastly are the ability buttons near the top right of the screen. There are the Upgrade Tools button, the Create Tower button, the Create Gem button, and the Fuse Gems button. The Upgrade Tools button will allow the player to invest energy into unlocking the ability to create more powerful Gems. It is currently not functional. Without using this ability, the player will only be able to create basic Gems. Each upgrade allows the player to create a higher rank of Gems. The Create Tower button will switch the game into a create Tower state. While active, this state will allow the player to create a Tower on the map by selecting a map cell where they want to create a Tower. Each Tower created will cost the player a certain amount of energy, and Towers will not be able to be created if the player does not have sufficient energy saved up. Next is the Create Gem button. Pressing this button will bring up the Create Gem button panel, which will be overlaid on the top slots of the inventory panel. The first selection will allow the player to select which rank of Gem they wish to create. Then, the button panel's button will change to the corresponding rank and allow the player to select which type of Gem they wish to create. Higher ranks of Gems cost more energy to create, and each type of Gem cost the same amount of energy to create. However, the player may choose the white Gem type at the bottom to produce a Gem of a random type at a reduced cost. The last button is the Fuse Gems button. Pressing this button will allow the player to combine two Gems into one more powerful Gem. It is currently not functional.

Playing the game is simple. The player should start by creating some Towers on the map, then create a Gem for each Tower. The Gems can then be moved into each of the Towers to prepare them to attack. Once the player has set up an initial defense, they should begin the first round. The fast forward button will begin the first wave of the first round and a timer will begin indicating how long until the next wave will begin. Each wave will create several enemies, each separated by a short delay. If the fast forward button is pressed again, the next wave will be started immediately, causing multiple waves to create enemies at the same time. This dramatically increases the difficulty of the level, but will award bonus score to the player. After an enemy is created, it will begin moving toward the end tile, show as a round platform on the map. Enemies will always take the shortest, most direct path, which the player should exploit by placing Towers in strategic locations, and arming the most well-placed Towers with their most powerful Gems. As the player's Towers attack nearby enemies, the enemies

will eventually be destroyed by the attacks. Each enemy destroyed will grant the player an increase to their score and energy. Once every wave of a round has had all of its enemies destroyed or reach their destination, the player can use their energy to create more Towers or Gems. However, if too many enemies reach their destination, reducing the player's health points to zero, the player loses. If the player manages to survive every round of the level, the player wins and the next level may be loaded, resetting the map and the player's attributes, allowing the player to continue playing.

Recommendations for Enhancements

There are still a great deal of features that I would like to add to my application. While it is in a roughly playable state, it could benefit a great deal from the addition of many more enhancements and some overall polish. While I'm satisfied with the visual aesthetic of the game, an upgrade to the quality and variety of the textures would be very effective. Additionally, there are still many game mechanics which I was not able to implement over the course of this project.

One of the most time-consuming parts of this project was acquiring the graphical assets and modifying them to suit my application. The textures used in my project are all low resolution, but maintain a style that I feel is appropriate for the game. On the other hand, higher resolution textures would be better, and the limited selection of textures makes some of the visual cues of the game more difficult to read. For example, adding a visual effect to yellow Gems that would indicate the number of attack charges they have remaining would be an effective visual cue. A variety in enemy types would also raise the quality of the game overall. One of the most sorely missed features here is the lack of sound. Given the difficulty I had with acquiring the textures for the game, I did not manage to find such usable assets for sound effects.

There is also a great deal of room for improvement in the mechanics of the game itself. Some features, such as tools upgrade level and fusing Gems, were not able to be implemented within the confines of the project. While they don't directly contribute to whether the game is playable or not, they would add an additional layer of strategy to the game, increasing the potential difficulty of the game and making it more suitable for my target audience. Then there are smaller features such as having health point indicators above enemies to show how close enemies are to being destroyed, and damage indicators showing how much damage each Tower attack dealt to an enemy on impact. These would add more visual cues to allow players to assess the effectiveness of their strategy.

One of the more difficult problems to address is the performance of the application. While it is able to maintain an acceptable framerate and loading times, these are still slower than I would expect. It is difficult to assess where the slow performance is primarily coming from. However, an optimization of the interfaces between SpriteKit objects would be the easiest place to gain a boost in performance.

In the end while my game is playable, it still leaves a great deal of room for enhancements and improvements. I'm happy with the outcome of the architecture and design of the game engine itself, as it will allow me to continue to make these improvements over time thanks to its flexibility.

Bibliography

"Free Images – Pixabay." Free Images – Pixabay. Pixabay, n.d. Web. 18 May 2016. <https://pixabay.com>.

Game In A Bottle. "GemCraft." GemCraft. Armor Games, n.d. Web. 10 Oct. 2013. <http://armorgames.com/play/1716/gemcraft>.

Holko, Peter. "Gem Tower Defense." Gem Tower Defense. N.p., n.d. Web. 10 Oct. 2013. <http://www.gemtowerdefense.com/>.

Ironhide Game Studio. "Kingdom Rush: Media." Kingdom Rush. Ironhide Game Studio, n. d. Web. 10 Oct. 2013. <http://www.kingdomrush.com/play.php>.

Lester Patrick. "A* Pathfinding for Beginners." A* Pathfinding for Beginners. Policy Almanac, 18 July 2005. Web. 05 Dec. 2013. <http://www.policyalmanac.org/games/aStartTutorial.htm>.

Prokein, Reiner. "Reiner's Tilesets." Reiners Tilesets. N.p., n.d. Web. 18 May 2016. <http://www.reinerstilesets.de>.

Red Blob Games. "Pathfinding for Tower Defense from Red Blob Games." Pathfinding for Tower Defense. Red Blob Games, 06 Feb. 2014. Web. 18 May 2016. <http://www.redblobgames.com/pathfinding/tower-defense/>.