6-2015

ANTICS: A CROSS-PLATFORM MOBILE GAME

Gerren D. Willis
California State University - San Bernardino

ANTICS: A CROSS-PLATFORM MOBILE GAME

A Project

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Gerren Willis

June 2015

ANTICS: A CROSS-PLATFORM MOBILE GAME

A Project

Presented to the

Faculty of

California State University,

San Bernardino

by

Gerren Willis

June 2015

Approved by:

David Turner, Advisor, Computer Science Date

Kerstin Voigt

Ernesto Gomez

ABSTRACT

Recent technologies have made it increasingly easier for independent developers to build and deploy gaming applications for mobile devices. The focus of this master's project is to investigate one such set of technologies, which include Adobe AIR, Starling and Feathers, that allow games to be written in ActionScript and run on both Android and iOS devices. For this purpose, I chose to use these technologies to design and implement the game ANTics proposed by Whim Independent Studios, a start up game studio in San Bernardino. ANTics is written in ActionScript and runs on Adobe AIR. It can be deployed on a variety of different operating systems including: Android, iOS, OS X and Windows. I will discuss in this project report the game design requirements and how I chose to implement them. This will include system architecture and core functionality such as user interface, game screens, asset management, AI, particle systems, and other gameplay mechanics. I will also discuss the development tools that were used to create the software components of the game.

ACKNOWLEDGEMENTS

I would first like to thank my committee members Dr. Voigt and Dr. Gomez for supporting me on this project. I would also like to thank my graduate coordinator Dr. Mendoza for keeping me on track during my years as a graduate student. I would like to give special thanks to my project advisor Dr. Turner who has been by my side and helped me since my first year as an undergraduate. Without his assistance, I would have not gained the experience necessary for completing a project of this magnitude. I would also like to thank all of my team members at Whim Independent Studios. Especially the co-founders Grover Wimberley, Mathew Estrada, and Paul Vela for giving me the opportunity to work at their company. Finally, I would like to thank my family for their encouragement and support.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Background

For this master's project I developed a mobile game called ANTics. I was a member of a small development team at an indie company called Whim Independent Studios, which was established by Grover Wimberly, a former student at California State University San Bernardino. I was the primary programmer responsible for the system architecture and implementation of the game requirements. Other members were responsible for creating the art and music assets.

1.2 Purpose

The purpose of this master's project is two-fold. The primary purpose is to understand how to develop games for mobile devices using the latest tools and technologies and to improve my skill-set as a gameplay programmer. The secondary purpose is to help out a start up indie company by developing a successful mobile game that can be marketed. The game is also intended to be extensible so that other similar games can be created using the same architecture.

1.3 Scope

Developing a game, even a simple one, is a long and comprehensive undertaking that incorporates skill-sets from a variety of different fields. This report will only discuss the development of the software aspect of the game. It will also discuss the game design only to help the reader to understand the software requirements that were needed to be implemented. Other aspects of game development such as art, music, management, and marketing will not be discussed. Furthermore, over the course of development, ANTics went through several refactoring and game design changes. This report will only cover the game design concepts and software architecture of the current version of ANTics.

1.4 Development Tools

This project was created using various game development tools. The programming tools that were used include: the Adobe AIR cross-platform system, the FlashDevelop IDE, and the Starling and Feathers ActionScript libraries. The source code was completely written in ActionScript 3.0. There were also many other art and design tools that were used to create the assets for the project. These tools are outside the scope of this report and will not be discussed.

1.5 Definitions, Acronyms, and Abbreviations

The definitions, acronyms, and abbreviations used in the document are described in this section.

- ActionScript: Adobe's object oriented programming language for Flash applications.
- ADL: Adobe Debug Launcher.
- AIR: Adobe Integrated Runtime.
- AI: Artificial Intelligence.
- apk: Android file format.

- assets: Resources that are loaded in the application. such as image and audio files.
- class: A custom data structure that defines the properties and methods of the object it instantiates.
- collision rectangle: A rectangle that surrounds a display object that is used to determine collision calculation.
- display objects: Objects that are displayed visually in a Flash application.
- display list: A tree structure that manages all display objects in a Flash application.
- event: Some interaction that occurs in the application. Some examples of events are mouse click, touch, a new frame that occurs in the timeline, and completion of a load operation.
- event listener: A function that gets executed by the Flash runtime environment in response to a specific event.
- Feathers: An ActionScript library that contains a variety of helpful classes for developing GUI components in a gaming application.
- Image: A Starling class that has a texture mapped on a Quad.
- frame: A unit of time in the Flash timeline.
- gameplay: The rules and mechanics of a computer game.
- GPU: Graphics Processing Unit
- GUI: Graphical User Interface.
- HP: Health Points.
- HUD: Heads Up Display.

- IDE: Integrated Development Environment.
- ipa: iOS file format.
- Label: A Feathers class that is used to render a string of text.
- method: A function that belongs to a class.
- MovieClip: A Starling class that describes an animation as a collection of Images.
- object: An instance of a class.
- ProgressBar: A Feathers class that is a display object that represents a meter that can increase and decrease in value. This meter is depicted as an orange colored bar.
- property: A variable that belongs to a class.
- Quad: A Starling class that represents a rectangle with a color.
- SDK: Software Development Kit.
- skin: A texture that gets applied on GUI components.
- Sprite: A Starling class that is used to contain display objects.
- sprite: A texture of a character or object in a game.
- sprite sheet: A single image asset of a collection of sprites. Typically contains frames of animations for sprites.
- stage: The root display object in the Flash display list.
- Starling: An ActionScript library that contains a variety of helpful classes for developing gaming application.
- Stage3D: Adobe's GPU rendering pipeline.
- swf: Flash file format.

- texture: A 2D image asset usually in a PNG or JPEG file.
- texture atlas: Synonymous to sprite sheet and typically very large in size.
- TextureAtlas: A Starling class that can access textures that are stored in one big image which typically contains frames of animations.
- timeline: A structure that manages animations in a Flash application.
- tower defense game: A type of game where the goal is to prevent enemies from attacking certain objects on a map or game area.
- Tween: A Starling class used to animate certain properties on a display object.
- XML: Extensible Markup Language.

2. OVERVIEW OF DEVELOPMENT TOOLS

2.1 Adobe Air

ANTics is a Flash application that runs on top of Adobe AIR. Adobe AIR is a runtime environment that enables developers to package the same code into native applications for multiple operating systems, including Android and iOS [1]. With Adobe AIR, a developer does not have to worry about writing in several languages for devices with different operating systems. This relieves the developer from learning multiple native languages and also guarantee that an app will have the same look and feel on every operating system. In the past, users would have had to install Adobe AIR separately on their system in order to run AIR based applications. However, now Adobe has what is called a captive runtime, which embeds AIR directly into the application. This allows users to run the application as is, without having to install any extra components.

2.2 FlashDevelop

The ANTics project was developed in an IDE called FlashDevelop. FlashDevelop provides the Flash SDK and compiler as well as powerful debugging tools. According to [2]. ,"... it was created in 2005 by passionate Flash developers, for Flash developers. It is the product of many contributors which created what is today the best open source Flash development environment". FlashDevelop is very user friendly and similar to Visual Studio. It also provides ADL(Adobe Debug Launcher) which is a fast way to run and debug an application without having to test it directly on a target device. Furthermore, the ADL can emulate different screen dimensions, which is especially useful for mobile development. FlashDevelop can package an application as an apk(Android file format), an ipa(iOS file format), or as a Windows executable file.

2.3 ActionScript and Flash Applications

ActionScript is the programming language used to develop Flash applications. ActionScript is object oriented and syntactically similar to Java. It is also event-driven and deals with event handling and callbacks the same way as in JavaScript. Every ActionScript application also has a hierarchy of display objects called a display list. This is shown in figure 2.1.

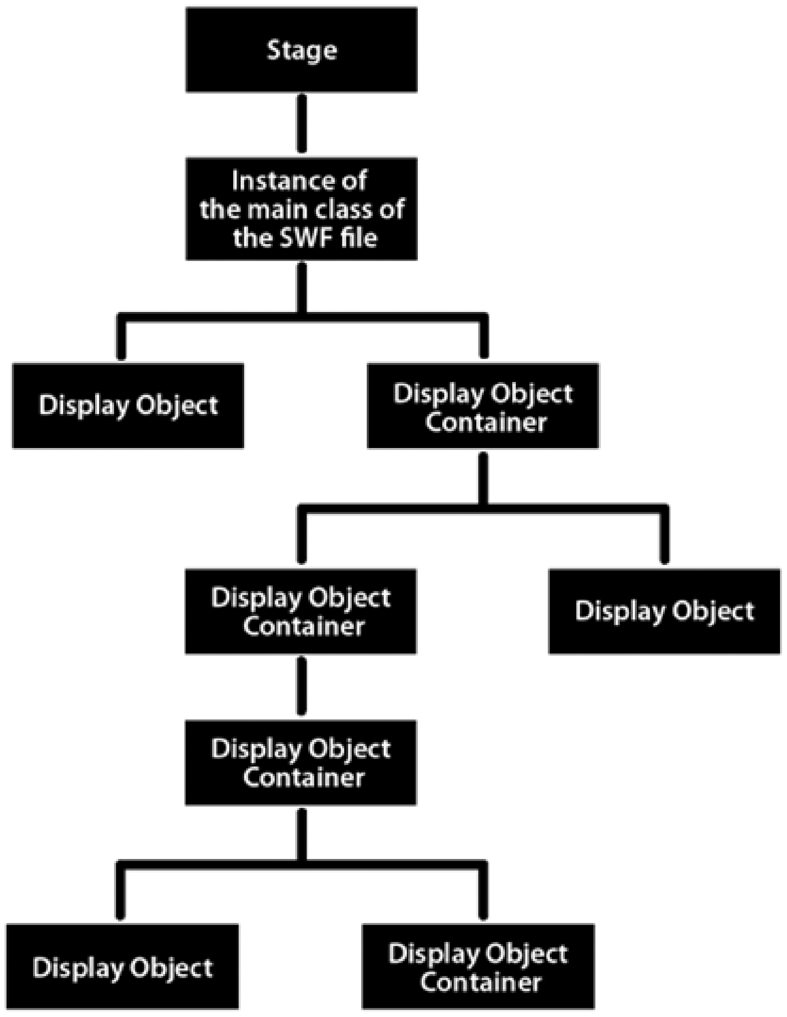


Fig. 2.1: Flash Display List Hierarchy [6]

The Flash display list contains all the visible elements in the application. Each application will have a single stage that acts as the base container of the display objects. Every swf(Flash file format) has a main class that AIR instantiates and adds to the stage immediately at startup. A display object is any element that is visible on the screen and serves as a base class that is extended by numerous other classes, such as sprites, movie clips, vector shapes, text fields, buttons, to name a few. Display object container is also a subclass of display object that has the added functionality of containing children of other display objects [6]. Traversing and managing the display list is straightforward and intuitive.

2.4 Starling and Feathers

Starling is an open source ActionScript library that mirrors the conventional Flash display list architecture. However, the display objects in Starling are rendered directly by the GPU via Stage3D (Adobe's GPU rendering pipeline). This hardware acceleration enables for faster rendering performance [3]. Starling also provides many other features that are essential for 2D game development, such as texture atlases for sprite sheet animation, a tweening library, and a particle system, to name a few. Starling is also an underlying framework for a user interface library called Feathers. According to [5], "Feathers puts it all together in one package: blazing fast GPU powered graphics, an impressive number of skinning options, and an extensible component architecture... to create a smooth and responsive experience." Every GUI component such as buttons, labels, and icons in ANTics comes from the Feather's library.

3. GAME DESIGN

3.1 Summary

ANTics is a casual tower defense game where the player must defend three stacks of candy from a hoard of different types of bugs. Bugs will randomly generate and enter from three sides of the screen and will try to approach and consume one of the three piles of candy on the playing field. The player must defend their candy by either tapping the bugs with their finger, or by using one of the five different power-ups. Power-ups are items that aid the player in different ways, these items include: a flyswatter, a magnifying glass, bug spray, sugar cubes, and firecrackers. If the bugs manage to eat all the piles of candy then the game is over.

There are two modes the game can be played in, survival mode and boss mode. The goal of survival mode is for the player to keep their candy safe for as long as possible. There score is determined by how many bugs the player squishes and how long the player was able to survive. In boss mode, the goal is for the player to defeat three unique and powerful bugs. The player must fight these bosses one after the next. If the player manages to defeat the last boss the player wins the game. The score for boss mode is determined by how many bugs the player squishes and how quickly the player completes the game.

3.2 Game Mechanics

3.2.1 Screens

The game has a simple touch interface with GUI buttons that transition between multiple game screens. This is shown in the screen flow diagram in figure 3.1.

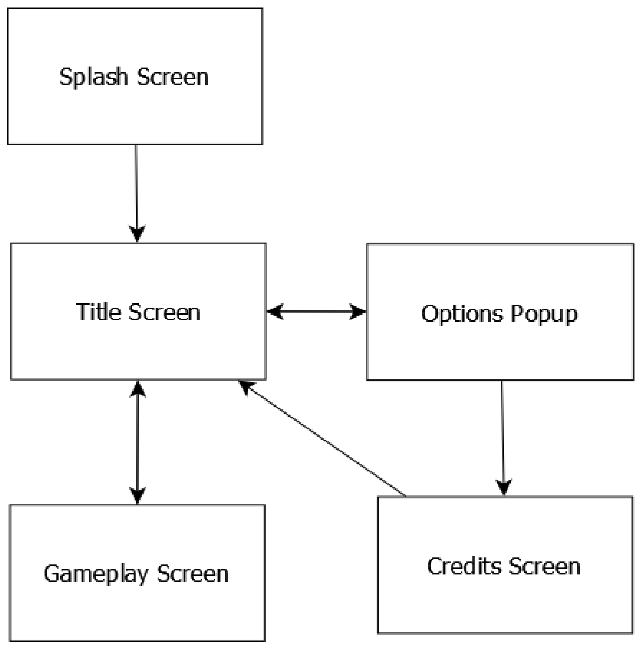
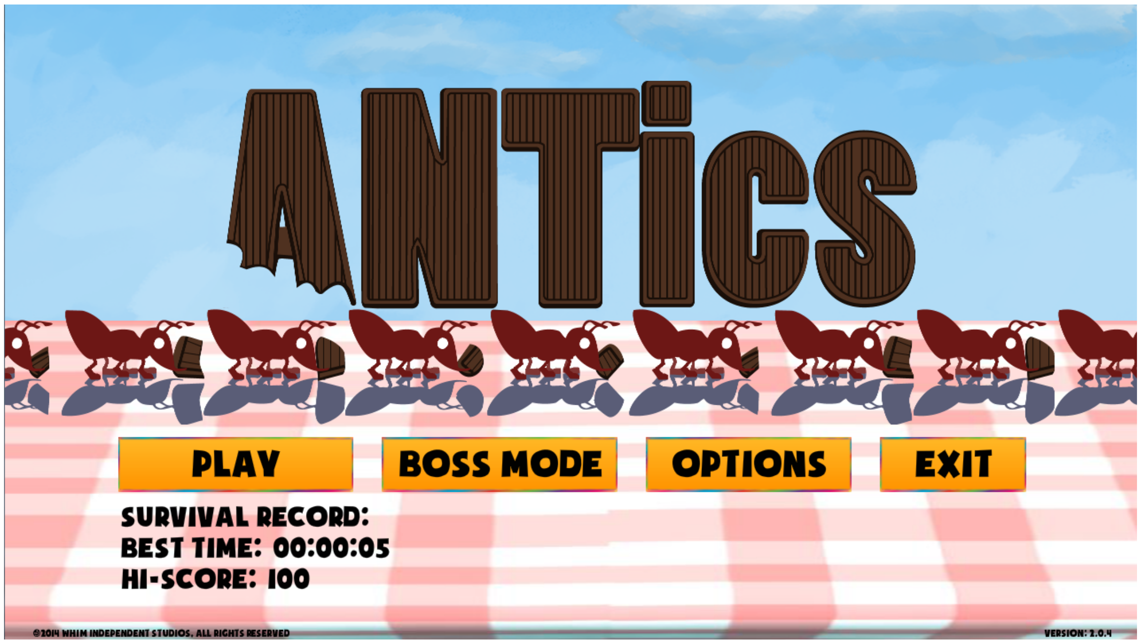


Fig. 3.1: Screen Flow Diagram

The first screen that appears when the application opens is the splash screen. This screen displays a logo of the game for a short time while some assets are being loaded. It will then transition to the title screen. The title screen acts as an interface to access the different components of the game. It has a button that will open the options screen, two buttons that will navigate to the gameplay Screen, and a button that closes the application.

This is shown in figure 3.2.



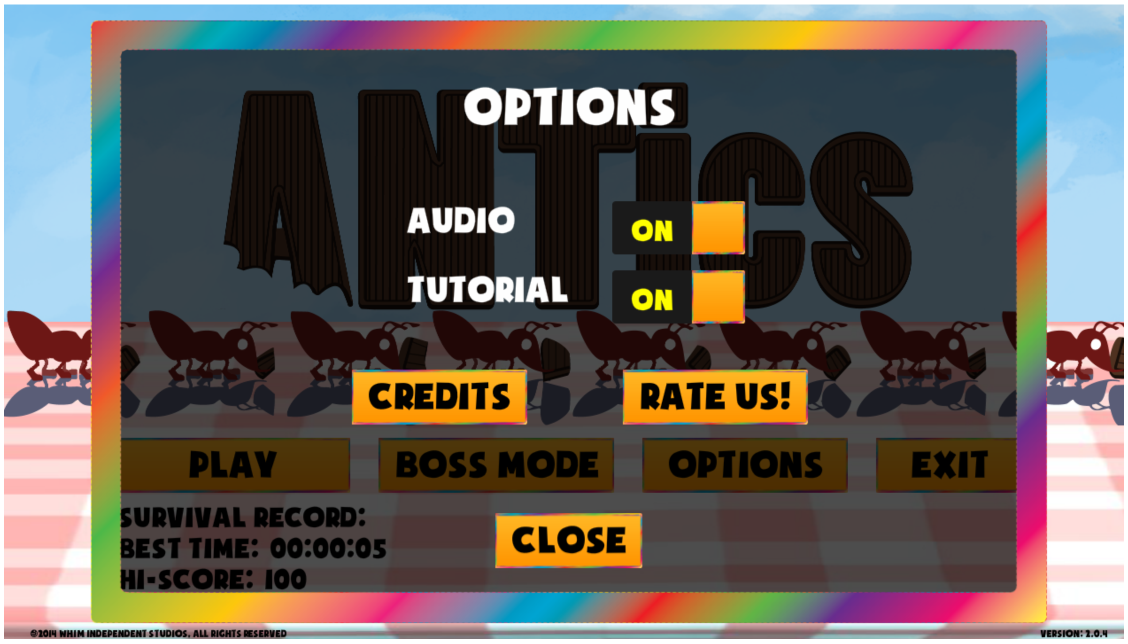
SURVIVAL RECORD:
BEST TIME: 00:00:05
HI-SCORE: 100

©2014 WHIM INDEPENDENT STUDIOS, ALL RIGHTS RESERVED

VERSION: 2.0.4

Fig. 3.2: Title Screen Screenshot

The options screen acts as an overlay on top of the title screen. This is shown in figure 3.3.



AUDIO ON
TUTORIAL ON

SURVIVAL RECORD:
BEST TIME: 00:00:05
HI-SCORE: 100

©2014 WHIM INDEPENDENT STUDIOS, ALL RIGHTS RESERVED

VERSION: 2.0.4

Fig. 3.3: Options Screenshot

The options screen is where the user can change the game settings such as turning the sound and tutorials on and off. It also has buttons that will navigate the user to the credits screen and a button that will take the user out of the application and into the Google Play store to rate the game.

The gameplay screen runs different gameplay logic and loads different assets depending on which mode it is in. The mode is determined by which button on the title screen the user activated to transition to the gameplay screen. The gameplay screen will transition back to the title screen when the player wins in boss mode and losses in both boss and survival mode. Figure 3.4 shows a screenshot of the gameplay screen.

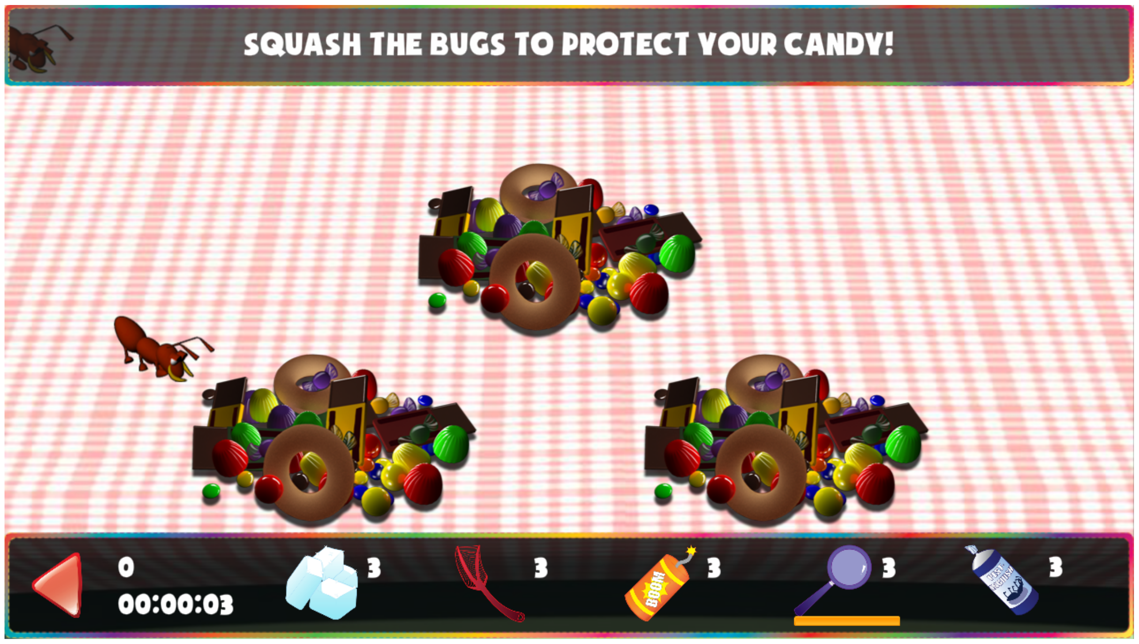


Fig. 3.4: Gameplay Screen Screenshot

The gameplay screen consist of the playing field and the HUD(Heads Up Display). The playing field consist of a background image of a picnic table, piles of candy, and the bugs. The player interacts with the playing field by either tapping bugs with their fingers or by utilizing power-up abilities. The HUD is what gets displayed over the playing field. At the top of the screen, the HUD is used to display helpful messages to the user. At the bottom of the screen, the HUD is used to display the current status of the power-ups. The user can interact with this part of the HUD by tapping a power-up icon to activate it.

3.2.2 Power-ups

Power-ups are items that add extra abilities that the player can utilize to help defeat the bugs. There are five different power-ups, each representing a real world bug exterminating tool. Below is a list of the power-ups and their abilities.

- Sugar Cube: Sugar cubes are dragged and dropped on to the playing field. Bugs will temporarily be drawn to the sugar cube instead of the candy.

- Swatter: Once activated, the user touches the screen and an image of a fly swatter will appear at the touch point. It will squish bugs that are inside its area of collision.

- Firecracker: Fire crackers are dragged and dropped on the playing field. They will have a two second delay and then explode with a particle effect, killing bugs within its collision field.

- Magnifying Glass: Once activated, the user can drag his finger across the playing field leaving a trail of particles that look like fire. Bugs that are caught in its wake will be instantly killed.

- Bug Spray: Once activated, the entire playing field will be covered with a particle effect resembling a toxic gas. All bugs on the screen will slow down for two seconds and then die.

3.2.3 Candy

Candies are the objects that the player must defend in order to stay alive. The player has a maximum of three candies. Bugs will be attracted to these candies and will attempt to devour them. Figure 3.5 is a series of screenshots that shows every state of the Candy.

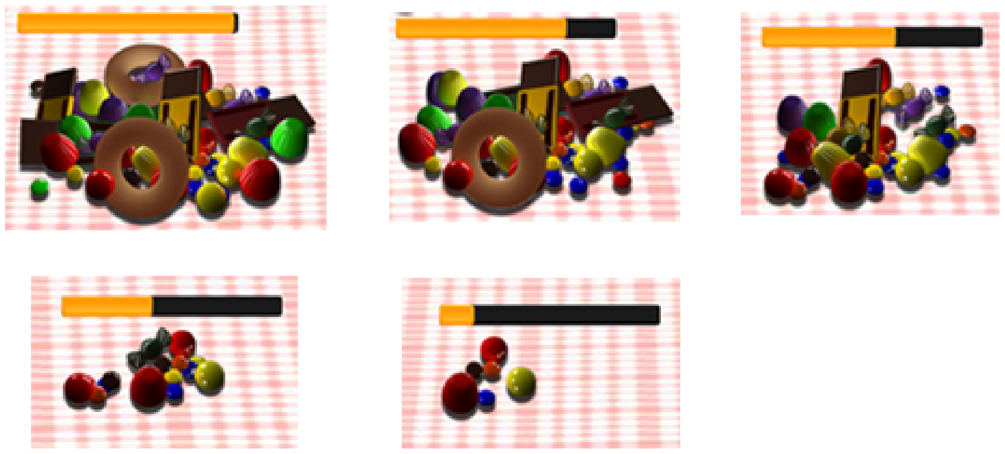


Fig. 3.5: Candy State Screenshots

Each candy has a health bar that gets displayed over it to indicate to the user the state of its current HP(Health Points). Depending on the HP status, the image of the candy will change to an image that shows a pile with less candy. Once the HP reaches zero, the candy is considered devoured and its image will no longer be displayed. The player loses the game when all three stacks of candy get devoured.

3.2.4 Bugs

Bugs are considered to be the enemies of the game and the obstacles that the player must overcome. Most bugs can be destroyed by the player by either taping them with their finger or by using one of the power-ups. There are a variety of bugs in the game each with different behaviors. Below is a list of all the bugs and their behaviors.

- Ants: Ants immediately go for the closest candy pile at moderate speed. They will continue to eat the candy until they are killed.
- Grasshoppers: Grasshoppers will wander around on the game field for three to five seconds before they go for a candy pile. Once they find a candy pile, they will deliver considerable damage and will then exit the playing field. They also

take two taps to squish.

- Hornets: Hornets will wander around on the game field for five to eight seconds and then will exit. They cannot be killed, and if the user taps them they will not be able to tap another bug for three seconds.

- Slugs: Slugs immediately go for the closest candy pile at a slow speed. They take three squishes to kill. Each tap will shrink them in size and will increase their speed, making them harder to tap.

- Ladybugs: Ladybugs are the fastest out of all the bugs. They wander around the game field for five to eight seconds and then will exit. If killed, they will reward the player with a power-up item or by replenishing the HP of a candy pile.

- Ant Queen: The ant queen will wander around the field and summon special type of ants that will carry pieces of the candy back to it, which will increase its health. The lower its health, the more ants it will spawn.

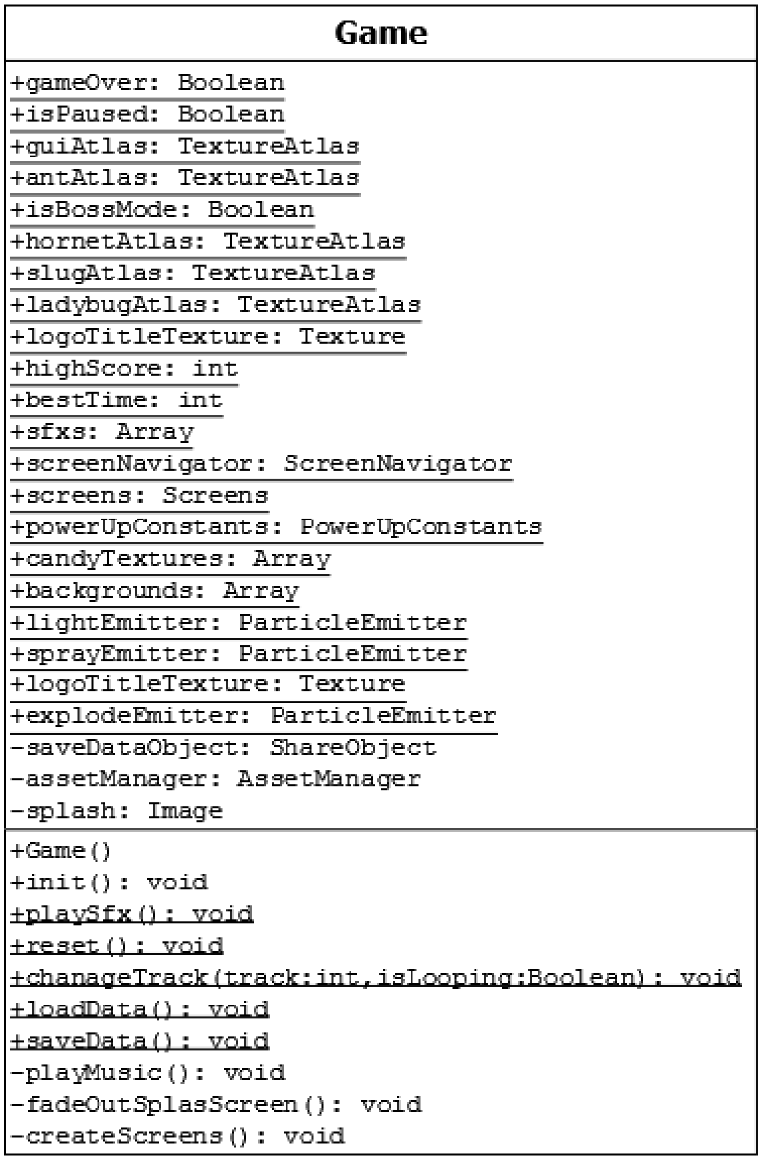
- Snail: The snail wanders around the game field with a hard shell that must be cracked open first before it can be damaged. Once the shell is cracked, it will increase its speed making it harder for the user to tap it. When it is tapped without its shell, it will spawn slugs from its body that will go for the candy.

- Cyber Hornet: The cyber hornet will wander around the field for a short time before shooting the candy with guns attached to its wings. After shooting a candy pile it will again wander for a short time before finding another candy pile to damage. It also will put up a shield every three to five seconds that makes it immune to all attacks.

4. SOFTWARE ARCHITECTURE AND DESIGN

4.1 Game and AssetManager Classes

The Game class is the first class that gets instantiated and is the first child added to the stage. All subsequent display objects in the application will be added under the Game display object. The Game class's primary tasks are to create the initial application variables, read and write to the file system, pause and resume the state of the game, load and display the splash screen, instantiate the AssetManager class, create the screen navigator, and navigate to the first screen of the application. It's properties and methods are mainly static and are accessible to all classes in the application. Most of these static properties consist of Texture, TextureAtlas, ParticleEmmiter, and Sound objects, which hold all of the art and sound assets. Other static properties store data that represents the player's progress, which will be later saved to the file system, such as the player's high score and best time. The static methods consist of different functions, such as handling music and sound effects, saving and loading the player's progress data, and a reset method that will set all of the player's progress data to its initial state. Figures 4.1, 4.2, and 4.3 shows UML class diagrams of the Game and AssetManager classes.



```
Game
+gameOver: Boolean
+isPaused: Boolean
+guiAtlas: TextureAtlas
+antAtlas: TextureAtlas
+isBossMode: Boolean
+hornetAtlas: TextureAtlas
+slugAtlas: TextureAtlas
+ladybugAtlas: TextureAtlas
+logoTitleTexture: Texture
+highScore: int
+bestTime: int
+sfxs: Array
+screenNavigator: ScreenNavigator
+screens: Screens
+powerUpConstants: PowerUpConstants
+candyTextures: Array
+backgrounds: Array
+lightEmitter: ParticleEmitter
+sprayEmitter: ParticleEmitter
+logoTitleTexture: Texture
+explodeEmitter: ParticleEmitter
-saveDataObject: ShareObject
-assetManager: AssetManager
-splash: Image
+Game()
+init(): void
+playSfx(): void
+reset(): void
+chanageTrack(track:int,isLooping:Boolean): void
+loadData(): void
+saveData(): void
-playMusic(): void
-fadeOutSplasScreen(): void
-createScreens(): void
```

Fig. 4.1: UML Diagram of the Game class

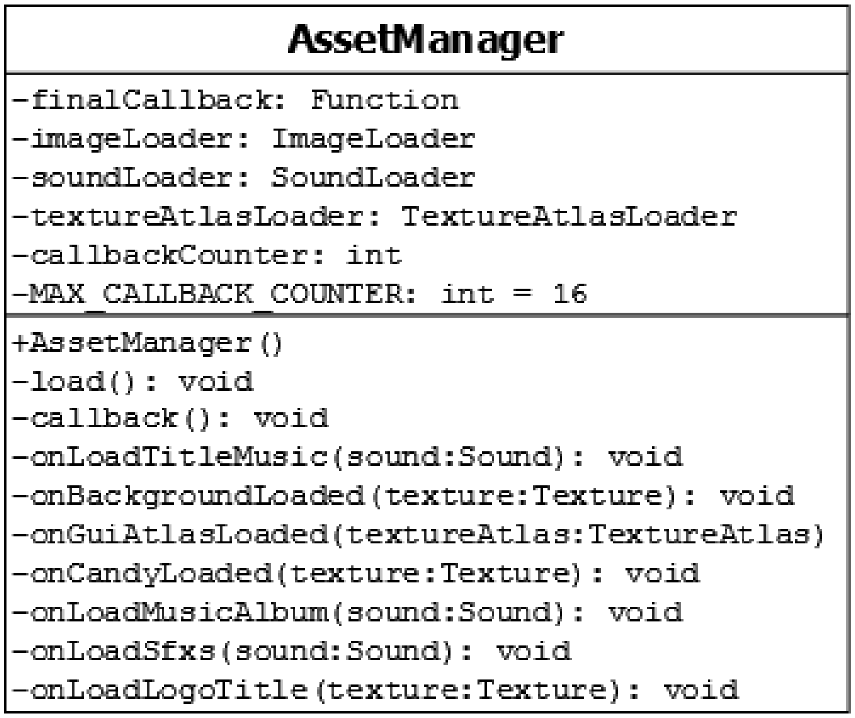

AssetManager
-finalCallback: Function
-imageLoader: ImageLoader
-soundLoader: SoundLoader
-textureAtlasLoader: TextureAtlasLoader
-callbackCounter: int
-MAX_CALLBACK_COUNTER: int = 16
+AssetManager()
-load(): void
-callback(): void
-onLoadTitleMusic(sound:Sound): void
-onBackgroundLoaded(texture:Texture): void
-onGuiAtlasLoaded(textureAtlas:TextureAtlas)
-onCandyLoaded(texture:Texture): void
-onLoadMusicAlbum(sound:Sound): void
-onLoadSfxs(sound:Sound): void
-onLoadLogoTitle(texture:Texture): void

Fig. 4.2: UML Diagram of the AssetManager class

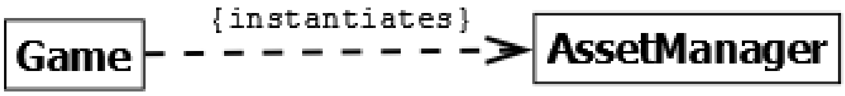
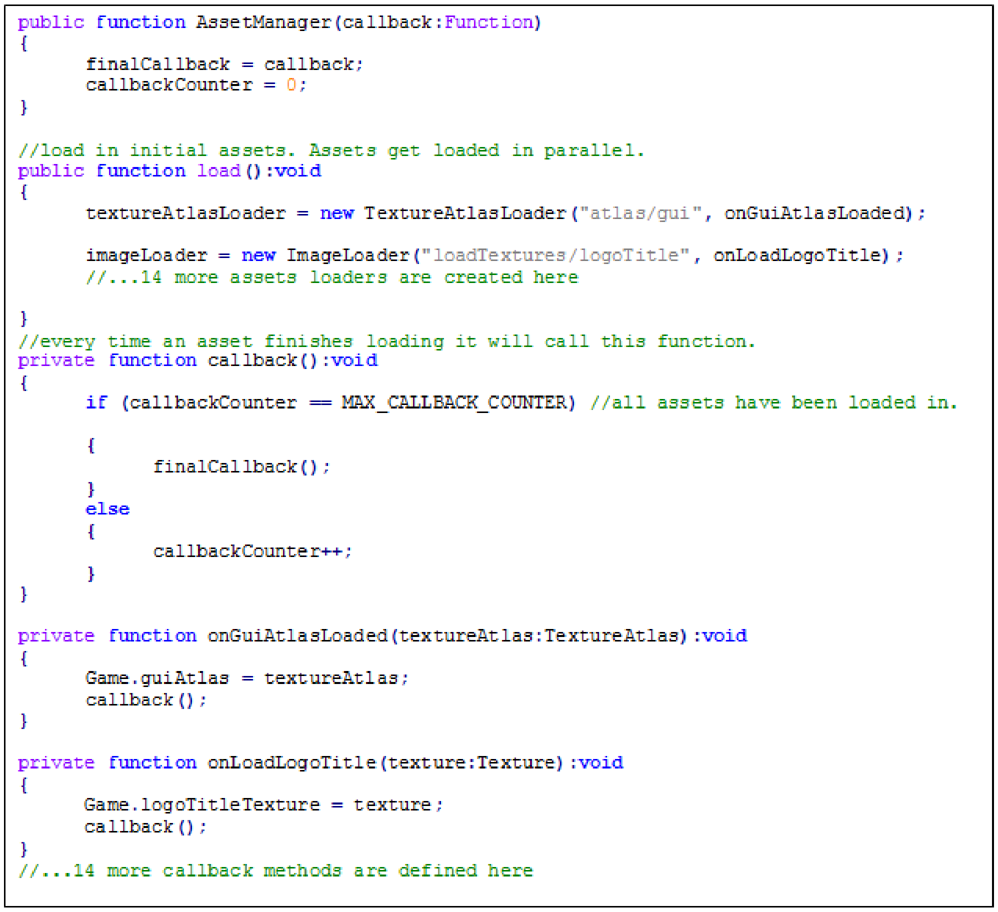


Fig. 4.3: UML Diagram of the Game and AssetManager relationship

Once the Game class is instantiated, the first thing it does is listen for when it has been added to the stage and then calls its init method. The init method will initialize its properties, add event listeners for when the application losses and regains focus, load the player's progress data, and load the splash screen. The splash screen then gets displayed, which shows the user an image of the game logo. While the splash screen is being displayed, the Game class instantiates the AssetManager class, which loads the rest of the initial game assets. Thus, the splash screen serves as a loading screen. The assets consist of the music, textures for the title screen, the particle systems data, and the texture atlases for the bugs and GUI elements. All of these assets get loaded in parallel and are tracked by a callback counter. This callback counter gets incremented by a callback method that each loader invokes once it finishes loading its content. Once the callback counter is equal to the number of assets that are designated to be loaded, the callback method will then call finalCallback, which is a callback method that was passed to its constructor by the Game class. Figure 4.4 shows code from the AssetManager class that depicts this process:



```
public function AssetManager(callback:Function)
{
	finalCallback = callback;
	callbackCounter = 0;
}

//load in initial assets. Assets get loaded in parallel.
public function load():void
{
	textureAtlasLoader = new TextureAtlasLoader("atlas/gui", onGuiAtlasLoaded);

	imageLoader = new ImageLoader("loadTextures/logoTitle", onLoadLogoTitle);
	//...14 more assets loaders are created here

}
//every time an asset finishes loading it will call this function.
private function callback():void
{
	if (callbackCounter == MAX_CALLBACK_COUNTER) //all assets have been loaded in.

	{
		finalCallback();
	}
	else
	{
		callbackCounter++;
	}
}

private function onGuiAtlasLoaded(textureAtlas:TextureAtlas):void
{
	Game.guiAtlas = textureAtlas;
	callback();
}

private function onLoadLogoTitle(texture:Texture):void
{
	Game.logoTitleTexture = texture;
	callback();
}
//...14 more callback methods are defined here
```

Fig. 4.4: AssetManager code

The finalCallBack method invokes the Game's fadeOutSplashScreen method. This method will call the createScreen method once the tween operation to fade out the splash screen is complete. The createScreen method sets the Feather's theme, instantiates the screen navigator, and transitions to the title screen. Figure 4.5 shows code of the Game's fadeOutSplashScreen and createScreen methods.

```
private function fadeOutSplashScreen():void

{
	var tween:Tween = new Tween(splash, 1);
	tween.fadeTo(0);
	Starling.juggler.add(tween);
	tween.onComplete = createScreens;
	progressBar.removeFromParent(true);

}

//creates all screens for the game and the screen navigator.
private function createScreens():void
{
	chosenMusic = MusicConstants.TITLE_MUSIC;
	//skins all gui elements based on a feathers theme
	var theme:MetalWorkMobileThemeExtension = new MetalWorkMobileThemeExtension(stage,
	                                                                             false);
	removeChild(splash);
	splash.dispose()      splash = null;
	screens = new Screens();

	screenNavigator = new ScreenNavigator();
	addChild(screenNavigator);

	//sets the transition effect for the screens.
	transition = new ScreenSlidingStackTransitionManager(screenNavigator);

	transition.duration = DURATION;

	screenNavigator.addScreen(screens.TITLE_SCREEN,
	                          new ScreenNavigatorItem(TitleScreen));

	screenNavigator.addScreen(screens.GAMEPLAY_SCREEN,
	                          new ScreenNavigatorItem(GameplayScreen));

	screenNavigator.addScreen(screens.CREDITS, new ScreenNavigatorItem(CreditsScreen));

	isOnSplash = false;
	//show the first screen for our game
	screenNavigator.showScreen(screens.TITLE_SCREEN);
}
```

Fig. 4.5: fadeOutSplashScreen and createScreen code

A theme is a component in Feathers that is used to skin every GUI element in the application. "A Feathers theme is a class that packages up the skinning code for multiple UI components in one location. Skins are registered globally, and when any Feathers component is instantiated, it will be automatically skinned"[5]. I have extended the default theme called MetalWorksMobileTheme, which Feathers provides with a subclass that I call MetalWorkMobileThemeExtention. This allows for additional functionality to the theme for further customization of the UI components. ScreenNavigator is a class provided by the Feathers Library. "A view stack-like container that supports navigation between screens through events" [8]. Every game screen class in this application extends the Feather's Screen class and gets added to the screen navigator. When I need to transition to a particular screen, I simply call the showScreen method on screen navigator and pass a constant string that is associated with the target screen. The showScreen method will dispose of its current screen, if one exists, instantiate the target screen, add it to the display list, and then transition to it.

4.2 TitleScreen Class

The first screen that the screen navigator adds to the display list and transitions to is the title screen. Figure 4.6 shows a UML class diagram of the TitleScreen and Option classes.

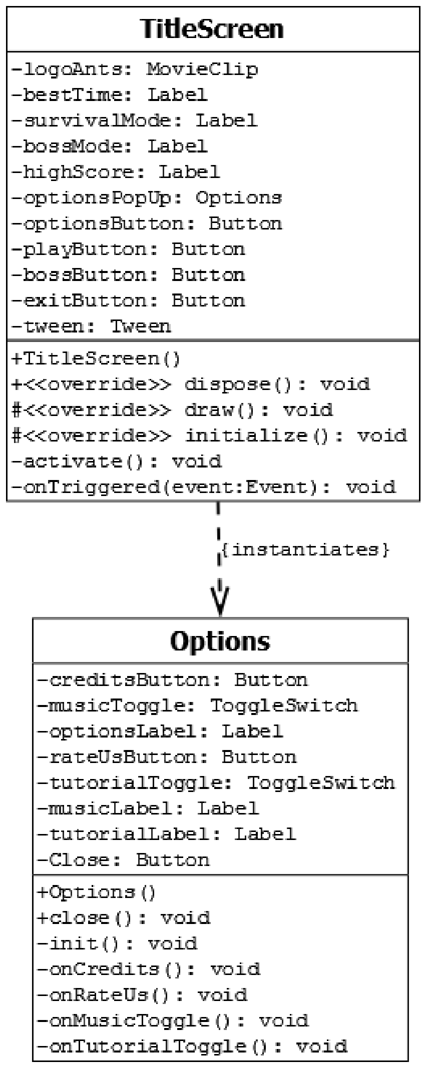
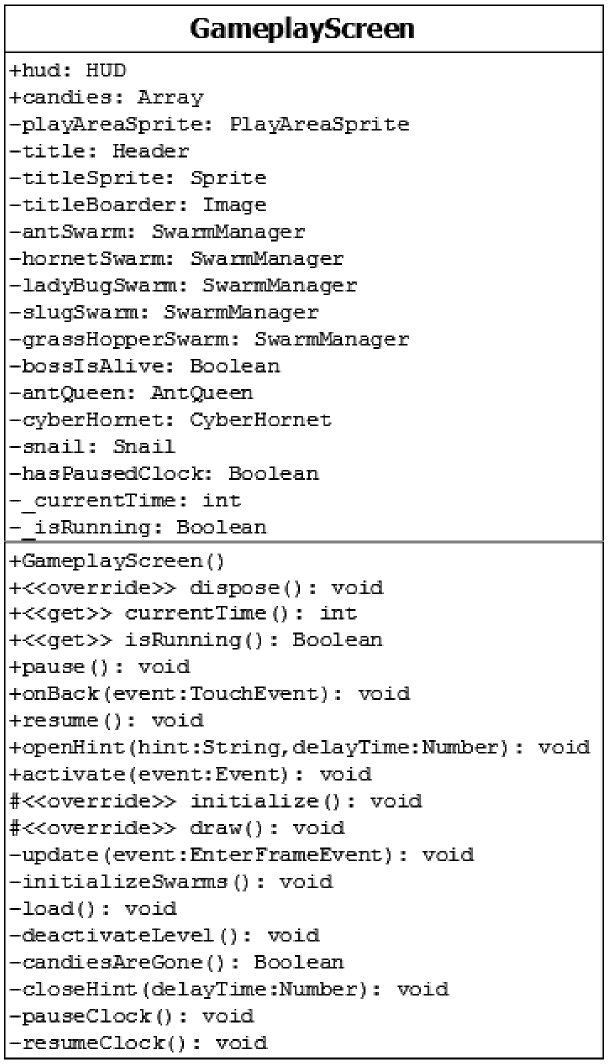


Fig. 4.6: UML Diagram of TitleScreen and Option classes

Once the title screen is added to the display list, the screen navigator calls its initialize method first, then it invokes its draw method, and finally it renders its display objects. I override the draw method to set the position and scale of the background images and menu buttons and add them to the display list. The menu buttons are initially positioned off the screen and will then get tweened into view. Once the tweening operation is complete, the activate method is invoked. Activate will add all the event listeners for each of the menu buttons. The event handlers for the play and boss mode buttons transition to the gameplay screen and will set a flag in the Game class that tells the gameplay screen if it should run in survival mode or boss mode. The options button will display the options popup screen. This options popup screen is an Options object that is not of a Feathers Screen class and does not get added to the screen navigator. Instead, Options gets instantiated by the TitleScreen class and gets added as a child to the TitleScreen display object. The Options class provides simple settings that the user can change, such as toggling the music and tutorials on or off, as well as a button that navigates to the credits screen. It also has a button that take the user out of the application and into the market place where they can rate the application. Lastly, the title screen has an exit button that will simply close out the application.

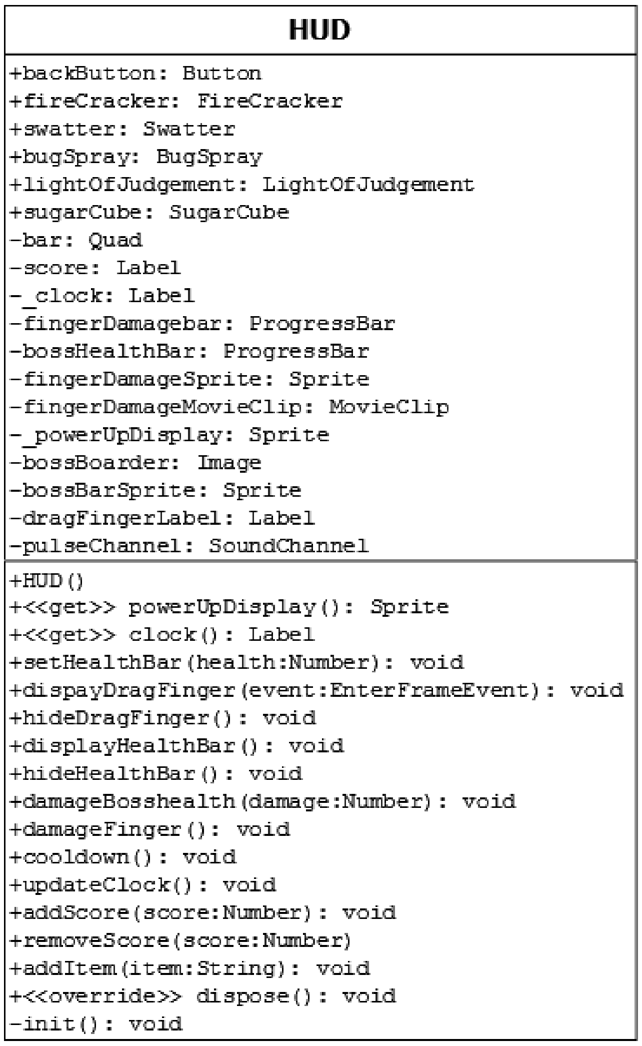
4.3 GameplayScreen Class

The gameplay screen facilitates the core game mechanics. It initializes and updates the bugs, candies, and HUD. Figures 4.7, 4.8, 4.9, and 4.10 are UML class diagram of the GameplayScreen, HUD, and PlayAreaSprite classes.



```
GameplayScreen
+hud: HUD
+candies: Array
-playAreaSprite: PlayAreaSprite
-title: Header
-titleSprite: Sprite
-titleBoarder: Image
-antSwarm: SwarmManager
-hornetSwarm: SwarmManager
-ladyBugSwarm: SwarmManager
-slugSwarm: SwarmManager
-grassHopperSwarm: SwarmManager
-bossIsAlive: Boolean
-antQueen: AntQueen
-cyberHornet: CyberHornet
-snail: Snail
-hasPausedClock: Boolean
-_currentTime: int
-_isRunning: Boolean
+GameplayScreen()
+<<override>> dispose(): void
+<<get>> currentTime(): int
+<<get>> isRunning(): Boolean
+pause(): void
+onBack(event:TouchEvent): void
+resume(): void
+openHint(hint:String,delayTime:Number): void
+activate(event:Event): void
#<<override>> initialize(): void
#<<override>> draw(): void
-update(event:EnterFrameEvent): void
-initializeSwarms(): void
-load(): void
-deactivateLevel(): void
-candiesAreGone(): Boolean
-closeHint(delayTime:Number): void
-pauseClock(): void
-resumeClock(): void
```

Fig. 4.7: UML Diagram of GameplayScreen class



```
HUD
+backButton: Button
+fireCracker: FireCracker
+swatter: Swatter
+bugSpray: BugSpray
+lightOfJudgement: LightOfJudgement
+sugarCube: SugarCube
-bar: Quad
-score: Label
-_clock: Label
-fingerDamagebar: ProgressBar
-bossHealthBar: ProgressBar
-fingerDamageSprite: Sprite
-fingerDamageMovieClip: MovieClip
-_powerUpDisplay: Sprite
-bossBoarder: Image
-bossBarSprite: Sprite
-dragFingerLabel: Label
-pulseChannel: SoundChannel
+HUD()
+<<get>> powerUpDisplay(): Sprite
+<<get>> clock(): Label
+setHealthBar(health:Number): void
+dispayDragFinger(event:EnterFrameEvent): void
+hideDragFinger(): void
+displayHealthBar(): void
+hideHealthBar(): void
+damageBosshealth(damage:Number): void
+damageFinger(): void
+cooldown(): void
+updateClock(): void
+addScore(score:Number): void
+removeScore(score:Number)
+addItem(item:String): void
+<<override>> dispose(): void
-init(): void
```

Fig. 4.8: UML Diagram of HUD class

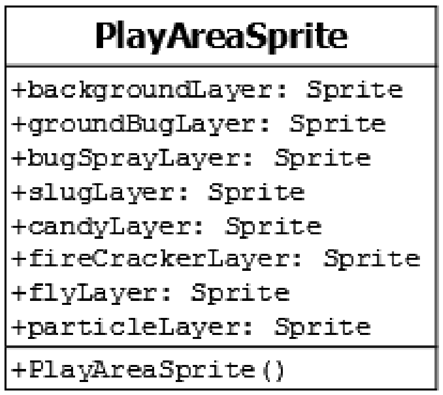


Fig. 4.9: UML Diagram of PlayAreaSprite class

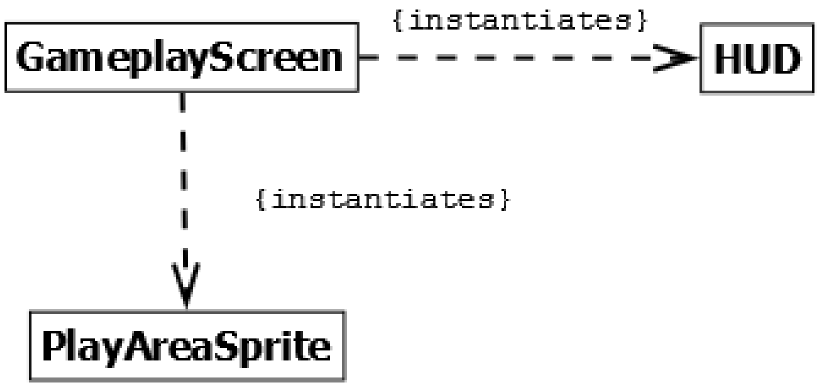


Fig. 4.10: UML Diagram of Gameplay, HUD, and PlayAreaSprite relationship

There are two modes the gameplay screen runs in: survival mode and boss mode. This is determined by the isBossMode flag in the Game class. The first thing the gameplay screen does is check if this flag was set. This will determine which music should be played, which bugs should be initially created, and which gameplay logic should be running. The first display object, which the gameplay screen adds as a child to its display object, is an instance of PlayAreaSprite. PlayAreaSprite extends Sprite and acts as a container for all other Sprite objects that deal with gameplay and are rendered on the screen. These gameplay Sprite objects include the background, candies, bugs, and power-ups. The play area sprite is composed of multiple instances of Sprites that act as layers. These layers insure the correct draw order of the gameplay Sprite objects. For example, the fly layer sprite object is drawn on top of the ground layer sprite object. Therefore, I add the bugs that fly to the fly layer and the bugs that crawl to the ground layer. This guarantees that the flying bugs always get drawn on top of the crawling bugs. The last display object the gameplay screen adds to the display list is the HUD.

4.4 HUD Class

The HUD is composed of four main components: titleBar, powerUpDisplay, fingerDamageSprite, and a bossBarSprite. The title bar is an instance of the Sprite class and contains a image object and a Feather's header object. The image object acts as a skin to improve the appearance of the title bar. The header object displays a label on top of a colored background. The purpose of the title bar is to display messages to the user as he is playing the game. One example is the "Get Ready!" message, which serves as a load screen when the user enters the gameplay screen. It remains displayed until all required assets finish loading. Other examples include: a series of tutorial messages that explain how to play the game, a "Game Over!" message when the player loses survival mode or boss mode, and finally a "You Win" message when the player wins in boss mode. The power-up display is an instance of a Sprite that contains five instances of the PowerUp class as its children. Other children it contains are labels that display clock and player score information, as well as a Feather's Button instance that allows the user to back out of the gameplay screen. The power-up display has a Quad object and a Image object that serve as simple skins. The main purpose of the power-up display is to act as a user interface for the player to activate the power-up functionality. The finger damage sprite is an instance of Sprite that contains a MovieClip object and a Feather's ProgressBar object as its children. The finger damage sprite is only visible when the user touches a Hornet object. When it is visible, the movie clip runs an animation of a pulsating finger, and the progress bar is displayed with its value gradually decreasing. This indicates to the player that he will not be able to squish another bug until the cooldown has finished. The boss bar sprite is an instance of Sprite that contains an Image object and a Feather's ProgressBar object as its children. The image displays the texture of the boss bar and the progress bar represents the HP of the boss. As the boss receives damage, the progress bar's value will decrease. The boss bar sprite is only visible in boss mode and when a Boss object appears on the stage.

4.5 SwarmManager and SwarmPool Classes

The gameplay screen creates, initializes, and updates five instances of the SwarmManager class, one for each type of bug(excluding the boss bugs). The SwarmManager class is responsible for managing a pool of bugs of the same type. This pool object is an instance of the SwarmPool class. Figure 4.11 is a UML class diagram of the SwarmManager and SwarmPool classes.

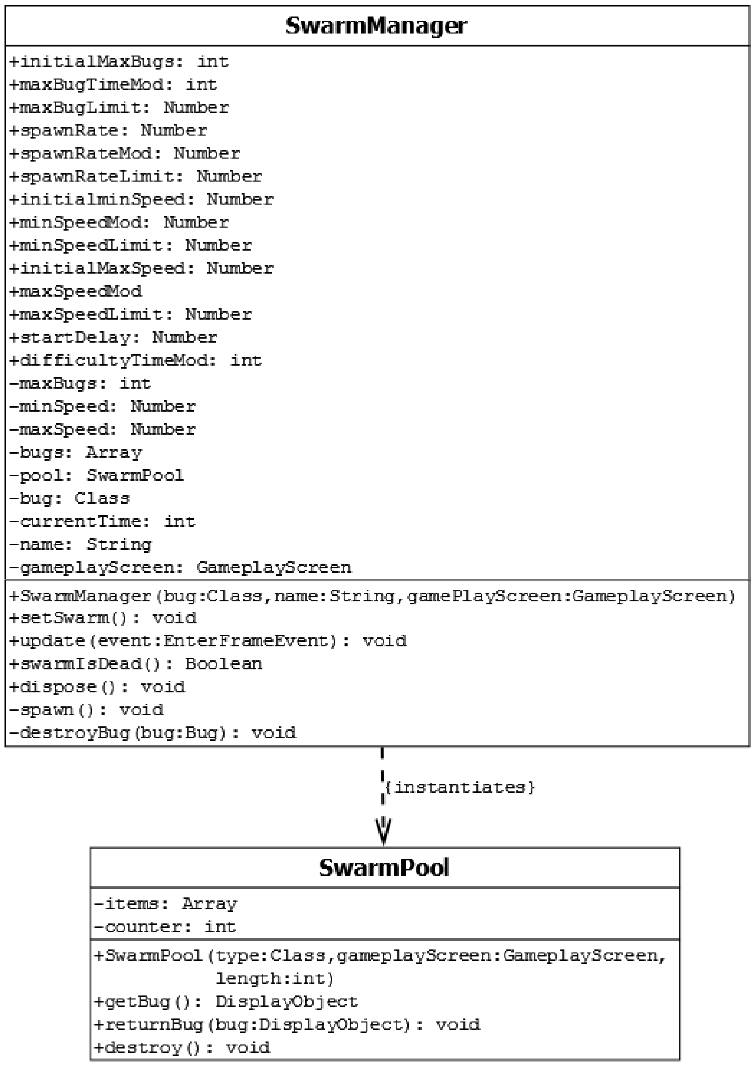


Fig. 4.11: UML Diagram of SwarmManager and SwarmPool classes

The SwarmPool class utilizes an optimization technique called object pool pattern.

> The object pool pattern is a software creational design pattern that uses a set of initialized objects kept ready to use a 'pool' rather than allocating and destroying them on demand. A client of the pool will request an object from the pool and perform operations on the returned object [7].

The swarm pool creates a static array of a given length that holds instances of bugs of the same type. The swarm manager will call the getBug method of its pool object every time it needs to get a bug out of the pool. This will remove the bug from the array in the pool object, and push it onto an array that is managed by the swarm manager. This is performed in the swarm manager's spawn method shown in figure 4.12.

```
private function spawn():void
{
	var bug:Bug = pool.getBug() as Bug;
	bugs.push(bug);
	bug.initialize(minSpeed, maxSpeed);
}
```

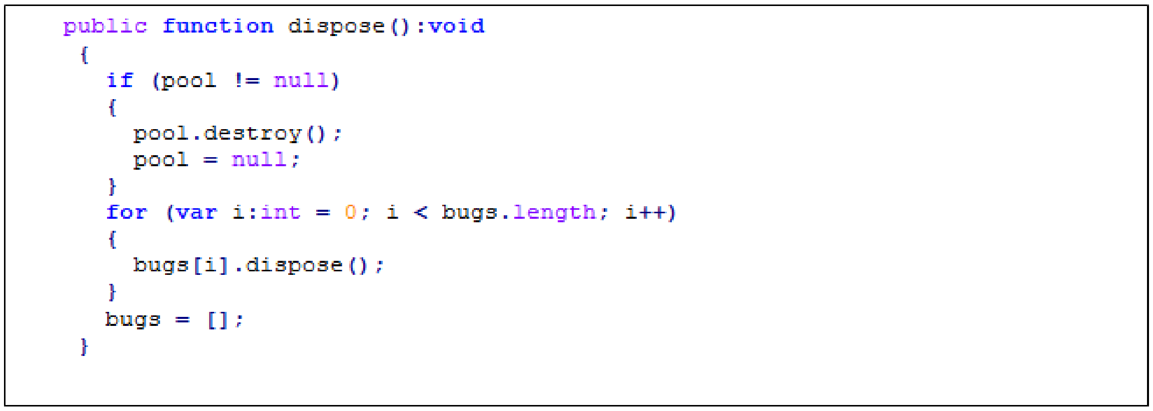
Fig. 4.12: spawn method code

Similarly, every time the swarm manager needs to destroy a bug it will splice it out of its array and return it to the pool. This is performed in the swarm manager's destroyBug method as shown in Figure 4.13.

```
private function destroyBug(bug:Bug):void
{
	for (var i:int = 0; i < bugs.length; i++)
	{
		if (bug == bugs[i])
		{
			bugs.splice(i, 1);
			pool.returnBug(bug);
		}
	}
}
```

Fig. 4.13: destroyBug method code

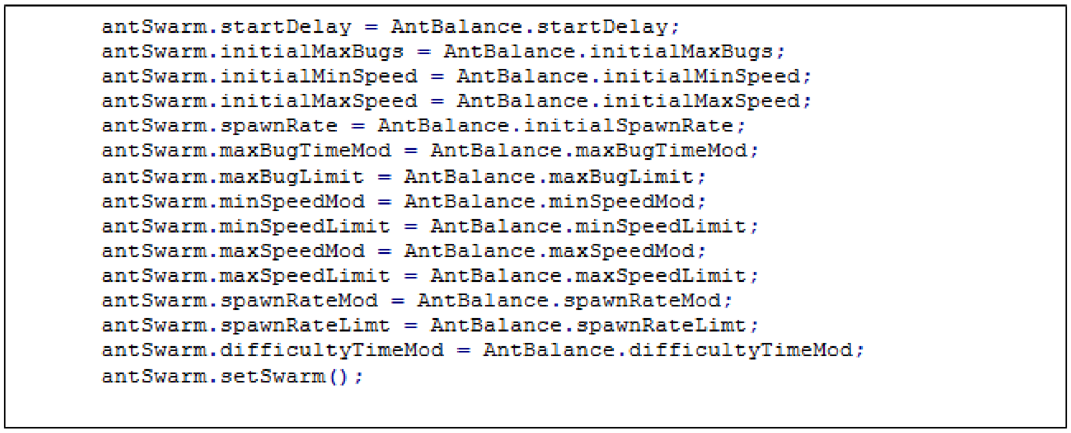
When the swarm manager is no longer needed, it's dispose method is called. This will destroy the bugs array and the pool object. The code for this method is shown in Figure 4.14.



```
public function dispose():void
 {
   if (pool != null)
   {
     pool.destroy();
     pool = null;
   }
   for (var i:int = 0; i < bugs.length; i++)
   {
     bugs[i].dispose();
   }
   bugs = [];
 }
```

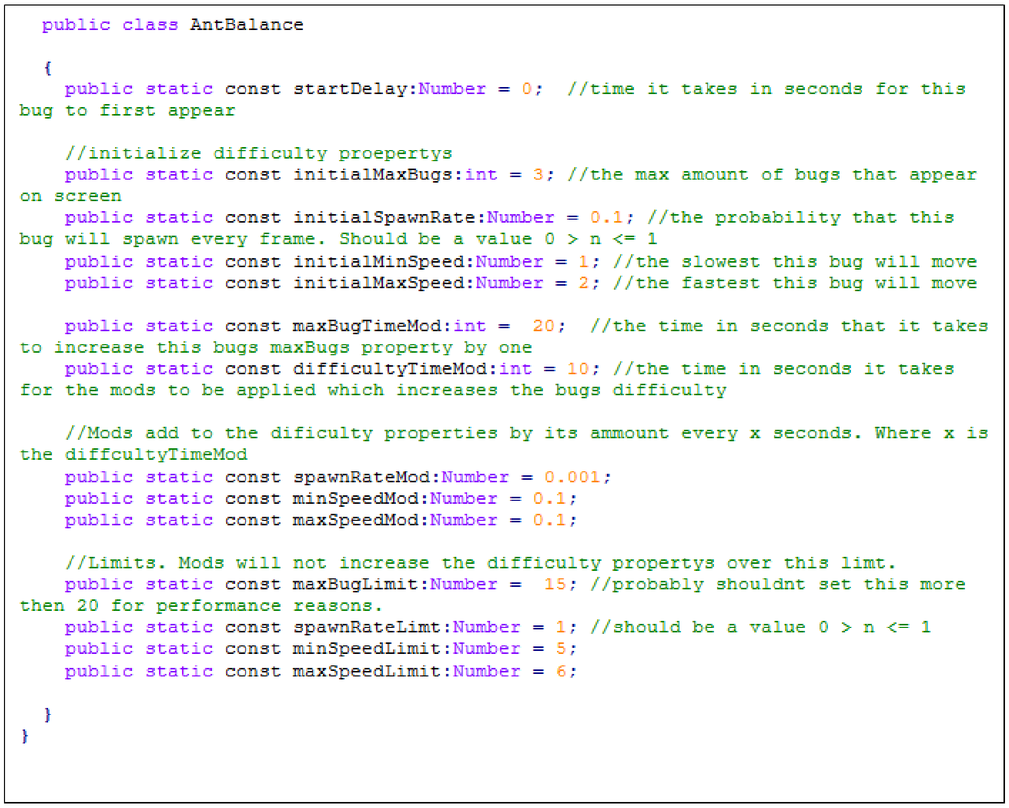
Fig. 4.14: dispose method code

The gameplay screen's initializeSwarms method will initialize all of the properties of the swarm manager, such as maxBugLimit, minSpeed, maxSpeed, spawnRate, and many more. These properties balance the difficulty of each bug, which in turn determines the difficulty of the game. Each bug type has a balance class that stores this balancing data that the initializeSwarms method pulls from. For example, Figure 4.15 shows partial code from the initializeSwarms method that initializes a SwarmManager instance for the ant type. Figure 4.16 shows the code for the AntBalance class.



```
    antSwarm.startDelay = AntBalance.startDelay;
    antSwarm.initialMaxBugs = AntBalance.initialMaxBugs;
    antSwarm.initialMinSpeed = AntBalance.initialMinSpeed;
    antSwarm.initialMaxSpeed = AntBalance.initialMaxSpeed;
    antSwarm.spawnRate = AntBalance.initialSpawnRate;
    antSwarm.maxBugTimeMod = AntBalance.maxBugTimeMod;
    antSwarm.maxBugLimit = AntBalance.maxBugLimit;
    antSwarm.minSpeedMod = AntBalance.minSpeedMod;
    antSwarm.minSpeedLimit = AntBalance.minSpeedLimit;
    antSwarm.maxSpeedMod = AntBalance.maxSpeedMod;
    antSwarm.maxSpeedLimit = AntBalance.maxSpeedLimit;
    antSwarm.spawnRateMod = AntBalance.spawnRateMod;
    antSwarm.spawnRateLimt = AntBalance.spawnRateLimt;
    antSwarm.difficultyTimeMod = AntBalance.difficultyTimeMod;
    antSwarm.setSwarm();
```

Fig. 4.15: initializeSwarms method code



```
  public class AntBalance

  {
    public static const startDelay:Number = 0;  //time it takes in seconds for this
bug to first appear

    //initialize difficulty proepertys
    public static const initialMaxBugs:int = 3; //the max amount of bugs that appear
on screen
    public static const initialSpawnRate:Number = 0.1; //the probability that this
bug will spawn every frame. Should be a value 0 > n <= 1
    public static const initialMinSpeed:Number = 1; //the slowest this bug will move
    public static const initialMaxSpeed:Number = 2; //the fastest this bug will move

    public static const maxBugTimeMod:int =  20;  //the time in seconds that it takes
to increase this bugs maxBugs property by one
    public static const difficultyTimeMod:int = 10; //the time in seconds it takes
for the mods to be applied which increases the bugs difficulty

    //Mods add to the dificulty properties by its ammount every x seconds. Where x is
the diffcultyTimeMod
    public static const spawnRateMod:Number = 0.001;
    public static const minSpeedMod:Number = 0.1;
    public static const maxSpeedMod:Number = 0.1;

    //Limits. Mods will not increase the difficulty propertys over this limt.
    public static const maxBugLimit:Number =  15; //probably shouldnt set this more
then 20 for performance reasons.
    public static const spawnRateLimt:Number = 1; //should be a value 0 > n <= 1
    public static const minSpeedLimit:Number = 5;
    public static const maxSpeedLimit:Number = 6;

  }
}
```

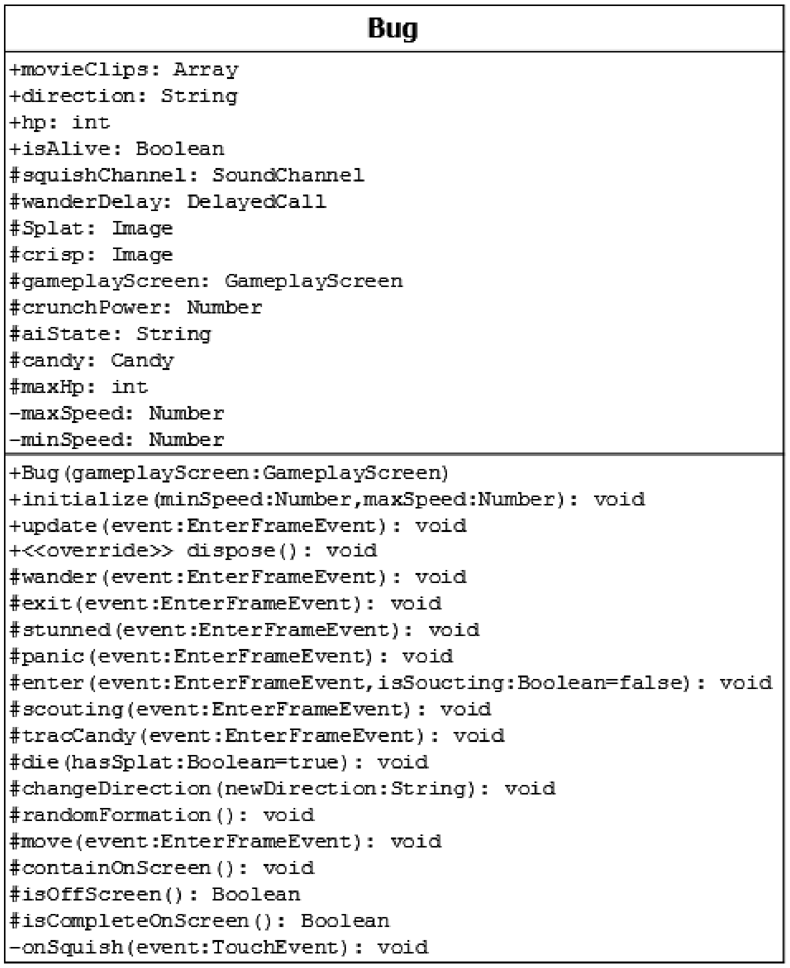
Fig. 4.16: AntBalance class source code

This makes balancing convenient because one can simply adjust the values stored in the AntBalance class to make the ants in the game easier or more difficult.

After the SwarmManager instances get initialized, the gameplay screen's update method will call update on each SwarmManager object. The swarm manager's update method keeps track of the current time in seconds so that it can increase the difficulty of the next bug spawned after a certain amount of seconds. This amount of seconds is determined by a modification value called difficultyTimeMod. The difficulty of each bug increases by adding a modification value to the bugs minSpeed, maxSpeed, and spawnRate. These modification values are set in the gameplay screen's initializeSwarms method as mentioned above. There is also another time modification value called maxBugTime, which determines when to increase the maximum amount of bugs that can be spawned. Each of these balancing properties also have a limit value that prevents adding the modifications at a certain point.

4.6 Bug Classes

There are nine different types of bugs in the game that all inherit from a generic Bug class. Figure 4.17 shows a UML class diagram of the Bug parent class and Figure 4.18 depicts the inheritance.



```
Bug
+movieClips: Array
+direction: String
+hp: int
+isAlive: Boolean
#squishChannel: SoundChannel
#wanderDelay: DelayedCall
#Splat: Image
#crisp: Image
#gameplayScreen: GameplayScreen
#crunchPower: Number
#aiState: String
#candy: Candy
#maxHp: int
-maxSpeed: Number
-minSpeed: Number
+Bug(gameplayScreen:GameplayScreen)
+initialize(minSpeed:Number,maxSpeed:Number): void
+update(event:EnterFrameEvent): void
+<<override>> dispose(): void
#wander(event:EnterFrameEvent): void
#exit(event:EnterFrameEvent): void
#stunned(event:EnterFrameEvent): void
#panic(event:EnterFrameEvent): void
#enter(event:EnterFrameEvent,isSoucting:Boolean=false): void
#scouting(event:EnterFrameEvent): void
#tracCandy(event:EnterFrameEvent): void
#die(hasSplat:Boolean=true): void
#changeDirection(newDirection:String): void
#randomFormation(): void
#move(event:EnterFrameEvent): void
#containOnScreen(): void
#isOffScreen(): Boolean
#isCompleteOnScreen(): Boolean
-onSquish(event:TouchEvent): void
```

Fig. 4.17: UML Diagram of the Bug class

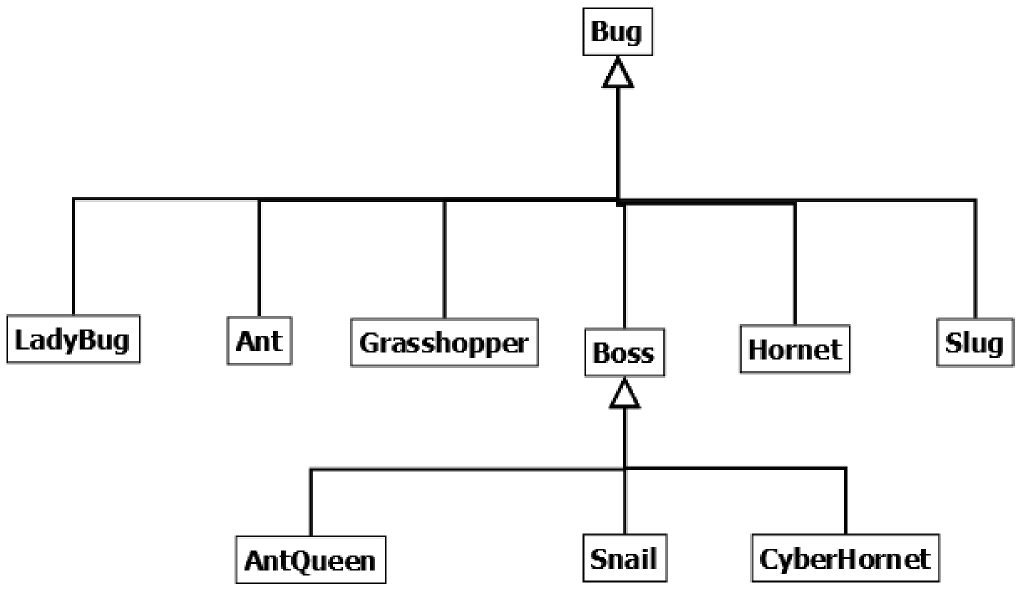


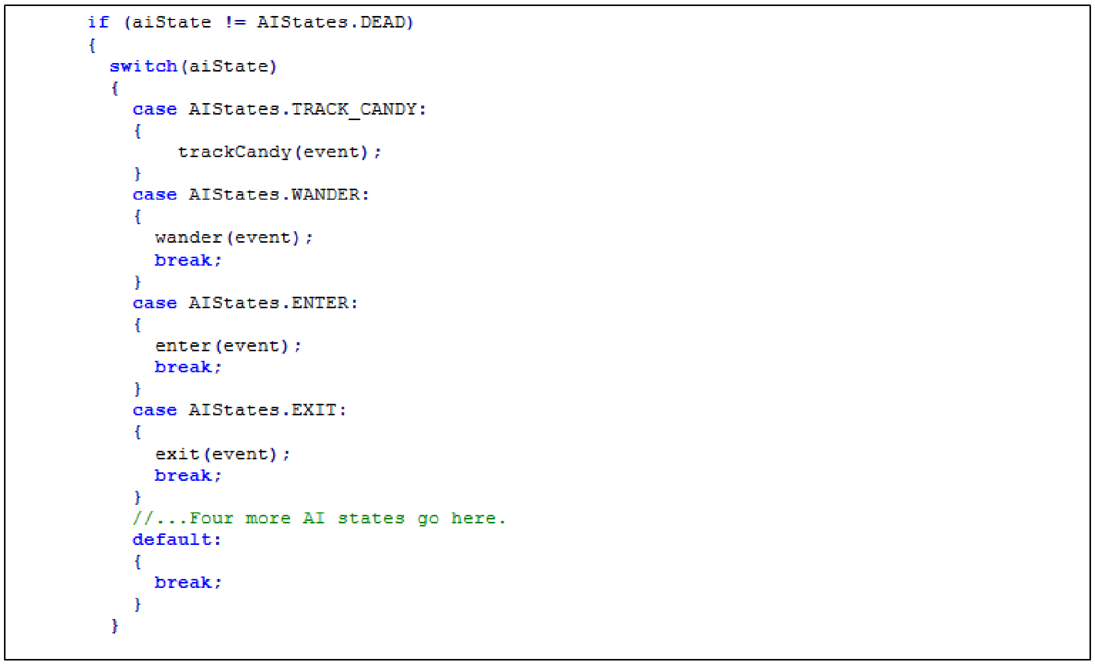
Fig. 4.18: UML Diagram of the Bug classes inheritance

The Bug class contains properties that are common across all bug types. These properties include, but are not limited to, array of movie clips for the animations, images that are displayed when the bug is killed, properties that keep track of its current direction and AI state, and balancing properties, such as minSpeed, maxSpeed, hp, maxHp, and crunchPower. The crunch power determines how much HP the bug removes from a candy that it is eating.

The Bug's constructor, which gets overridden by the subclasses, will initialize the properties that do not change during the bug's lifecycle. This includes, but is not limited to, movie clips, images, max HP, initial AI state, and crunch power. On the other hand, the initialize method will reinitialize the variables that do change during the bug's lifecycle. This includes, but is not limited to, min speed, max speed, AI state, direction, and the isAlive flag. Because I am using a pooling design, I have to use this initialize method to reinitialize the same bug object in memory each time it is being brought out from the pool. The initialize method is invoked in the SwarmManager's spawn method(see Figure 4.15 above).

The Bug's onSquish method is an event handler that gets called when the bug is touched by the player's finger. The onSquish method will decrement the bug's HP if the player was not damaged by a hornet. It will then call the die method if the HP is zero. The die method will set the isAlive flag to true, display the appropriate death image, and remove the bug from the display list.

The Bug's update method updates the state of the bug based on its current AI state. The AI state will get assigned a constant value from the AIState class. The bug's functionality will change depending on its current AI state. Figure 4.19 shows partial code from the update method that depicts this process.



```
if (aiState != AIStates.DEAD)
{
  switch(aiState)
  {
    case AIStates.TRACK_CANDY:
    {
        trackCandy(event);
    }
    case AIStates.WANDER:
    {
      wander(event);
      break;
    }
    case AIStates.ENTER:
    {
      enter(event);
      break;
    }
    case AIStates.EXIT:
    {
      exit(event);
      break;
    }
    //...Four more AI states go here.
    default:
    {
      break;
    }
  }
```

Fig. 4.19: Bug's update method code

Simply put, based on its current AI state, the corresponding method will be called. These methods can be overridden by the Bug's subclasses to provide unique behavior. Furthermore, each of the Bug's subclasses transition between a subset of these AI states differently from each other. To illustrate this point, below is a description of the AI transitioning process of two Bug subclasses: the Ant class, and Hornet class.

The Ant class initializes its AI state to “Enter”, which calls the enter method. The Ant class overrides the default behavior of the enter method and sets its isScouting flag from false to true. This tells the enter method that once it is done bringing the bug into the screen it must transition its AI state to “Scouting”, instead of transitioning to “Wandering”, which is the default behavior. The scouting method then gets called on the next update. This method will have the ant move randomly around the screen until its collision rectangle intersects with a Candy object’s collision rectangle. Then the ant’s AI state will transition to “Track Candy”, and the trackCandy method will get called on the next update. The trackCandy method will move the ant closer to the candy it collided with and will then add its crunchPower to the Candy object’s crunchPower(see section 4.7). The ant will stay in this state until the candy is gone and will transition back to the “Scouting” AI state. The ant will continue to transition in this manner until it is killed.

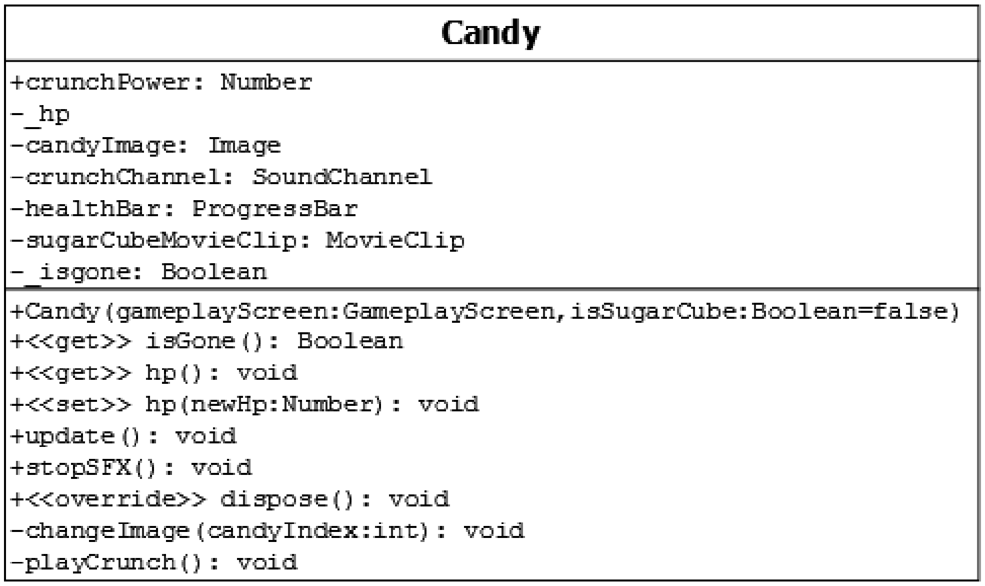
The Hornet’s AI behavior is distinctly different. Like the Ant class, it initialize its AI state to “Enter”, however, it will keep the enter method’s default behavior of transitioning to “Wander”. It overrides the wander method, which changes its default functionality from moving randomly indefinitely, to moving randomly until a given amount of time has passed. Once this time is up, the wander method will transition the AI state to “Exit”. The exit method simply moves the hornet off the screen and then calls the die method.

Similar to updating the state of the bug based on its AI states, the update method also changes the behavior of the bug depending on the power-up that is currently activated. The first thing the update method checks for is immunity. Immunity means that the bug is not affected by the power-up and therefore will not change its behavior. For the sugar cube power-up, when a sugar cube gets added to the play area sprite, the update method will assign the sugar cube as the Candy object to be tracked. For the firecracker power-up, the update method will call the bug's kill method only if the FireCrackerSprite object it collides with has its isExploding flag set to true. For more details on power-ups(see section 4.8).

The Boss type bugs operate much the same way as any other bugs. They have unique AI behaviors and will override certain methods to add extra functionality. However, there are a few distinct differences. First, their HP is displayed as a health bar on the HUD(see section 4.4 for more details). Second, they are not managed by the swarm manager because there are only three boss instances that ever get created, one for each type. Lastly, they are only instantiated if the player is playing in boss mode, and only one of the Boss instances is rendered on the field at a time. The Boss's die method notifies the gameplay screen when it is dead and then the gameplay screen will bring out the next boss.

4.7 Candy Class

The gameplay screen instantiates an array of Candy objects and adds them to the playAreaSprite's candy layer. The GameplayScreen's update method updates each Candy object and also check to see if all of the candies are gone. If this is the case, the gameplay screen will display "Game Over" on the HUD, and will then navigate to the title screen. Figure 4.20 shows a UML class diagram of the Candy class.



```
Candy
+crunchPower: Number
-_hp
-candyImage: Image
-crunchChannel: SoundChannel
-healthBar: ProgressBar
-sugarCubeMovieClip: MovieClip
-_isgone: Boolean
+Candy(gameplayScreen:GameplayScreen,isSugarCube:Boolean=false)
+<<get>> isGone(): Boolean
+<<get>> hp(): void
+<<set>> hp(newHp:Number): void
+update(): void
+stopSFX(): void
+<<override>> dispose(): void
-changeImage(candyIndex:int): void
-playCrunch(): void
```

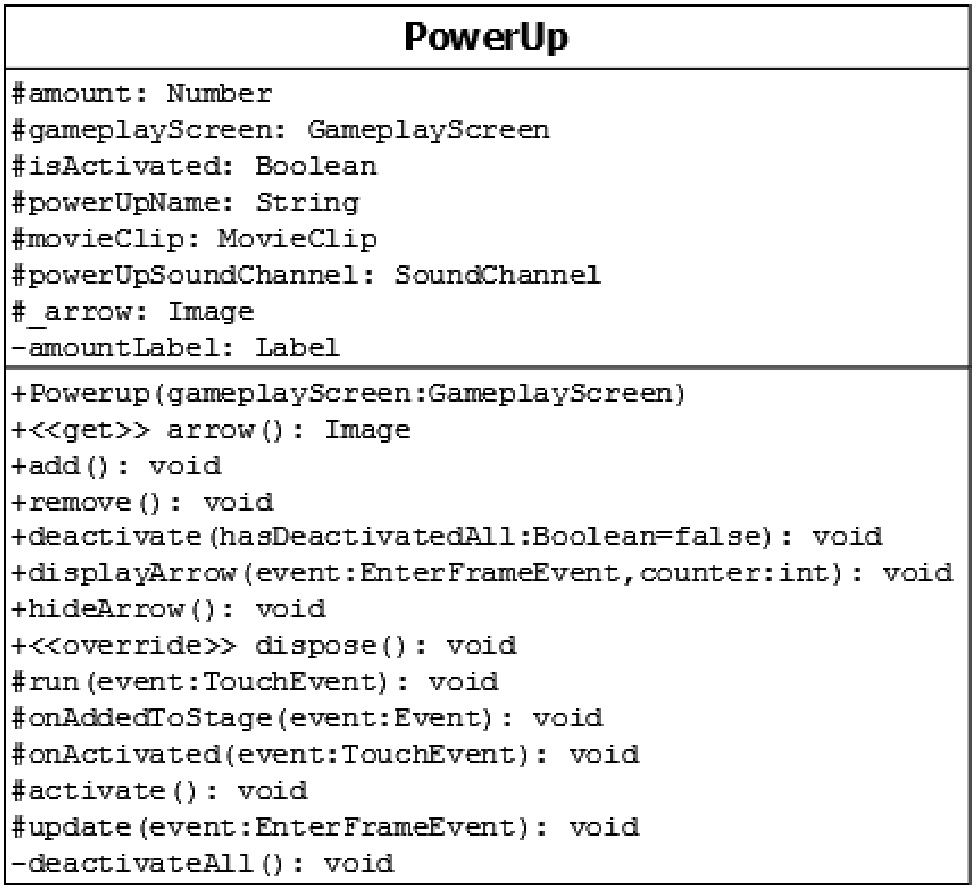
Fig. 4.20: UML of the Candy class

The basic properties of the Candy class mainly consists of a candyImage, a healthBar, hp, crunchPower, and a crunchChannel. The candy image is used to render one of the five candy textures. During its update, a new texture will get assigned to the candy image depending on the current state of the candies HP. The health bar is a ProgressBar object and is used to display the state of the candies HP to the user. The state of the HP is decremented by the crunch power at every update. The crunch channel is a Flash SoundChannel object. The SoundChannel class is a part of the Flash library and is used to control the sound in an application. The crunch channel is used to play a crunching sound effect when it is being eaten by a bug.

The Candy class also acts as a sugar cube power-up. The Candy's constructor takes a Boolean value to determine if the Candy should be displayed as a candy or a sugar cube. The difference is that if displayed as a sugar cube, the Candy class uses a movie clip instead of an image to display its textures. This is because the art for the sugar cube has an animation and the art for the candy does not.

4.8 Power-up Classes

There are five types of power-up classes in the game: SugarCube, BugSpray, Swatter, MagnifyingGlass, and FireCracker. These classes all inherit from the PowerUp Class. Figure 4.21 shows a UML class diagram of the PowerUp parent class and Figure 4.22 depicts the inheritance.



```
PowerUp
#amount: Number
#gameplayScreen: GameplayScreen
#isActivated: Boolean
#powerUpName: String
#movieClip: MovieClip
#powerUpSoundChannel: SoundChannel
#_arrow: Image
-amountLabel: Label
+Powerup(gameplayScreen:GameplayScreen)
+<<get>> arrow(): Image
+add(): void
+remove(): void
+deactivate(hasDeactivatedAll:Boolean=false): void
+displayArrow(event:EnterFrameEvent,counter:int): void
+hideArrow(): void
+<<override>> dispose(): void
#run(event:TouchEvent): void
#onAddedToStage(event:Event): void
#onActivated(event:TouchEvent): void
#activate(): void
#update(event:EnterFrameEvent): void
-deactivateAll(): void
```

Fig. 4.21: UML Diagram of the PowerUp classes

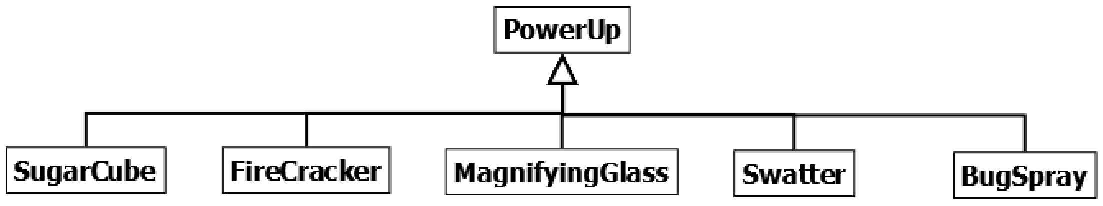
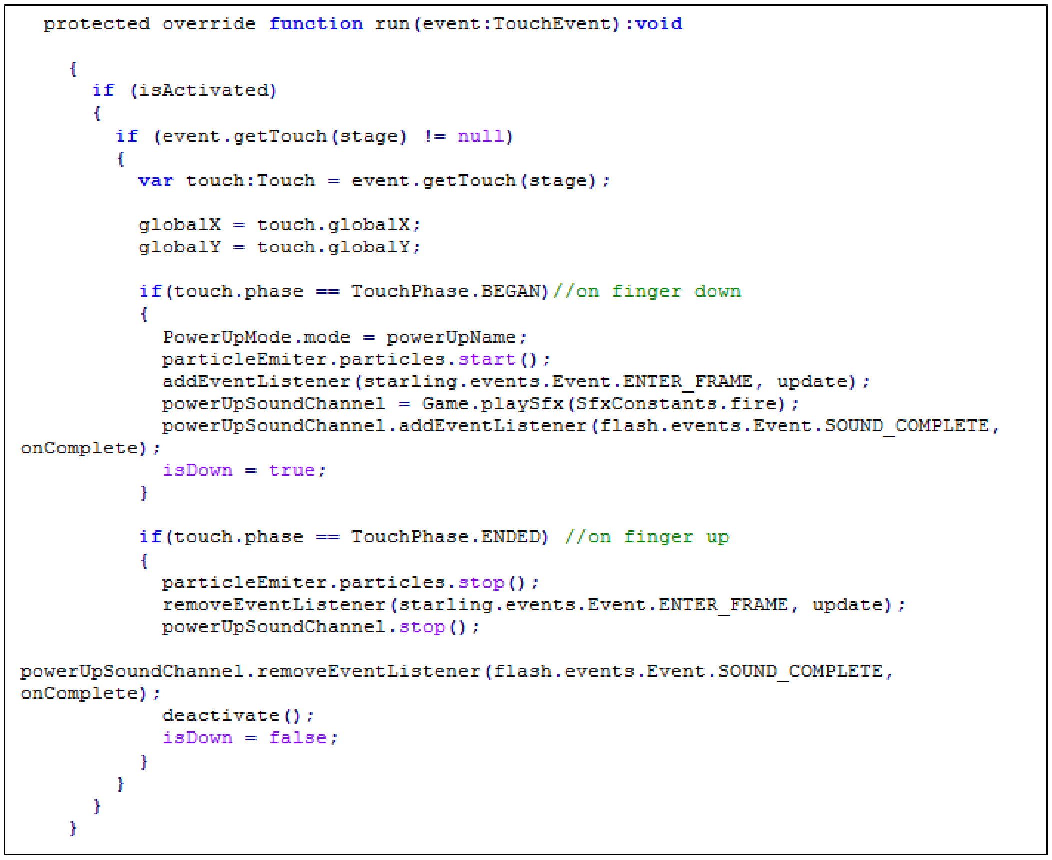


Fig. 4.22: UML Diagram of the PowerUp classes inheritance

The parent class PowerUp properties mainly consists of a movieClip, an amount, an amountLabel, a powerUpSoundChannel, and an isActivated flag. The movie clip is used to display an animated icon of the power-up to the user, which gets drawn on the HUD. The amount variable keeps track of how many power-ups the user currently has available. The amount label is used to display the amount to the user. The power-up sound channel plays the sound effect associated with the power-up type. The isActivated flag keeps track of when the current power-up is activated. The power-up will only run its functionality when this flag is set to true.

The ActionScript language does not have support for abstract functions or classes. However, I treat two functions in the PowerUp base class as if they were abstract. These functions are run, and update. The implementation of both of these functions are only done in the subclasses. Run is an event handler that gets called when the play area sprite is touched. Every implementation of the run method by the subclasses will first check that the isActivated flag is set to true before it performs its run logic. This insures that only the current activated power-up is functioning on the play area sprite. Update is an enter frame event handler that gets called every frame. Each subclass adds this event listener to its display object. Each subclass implements the run and update method differently. Figure 4.23 shows the code for the MagnifyingGlass's run method.

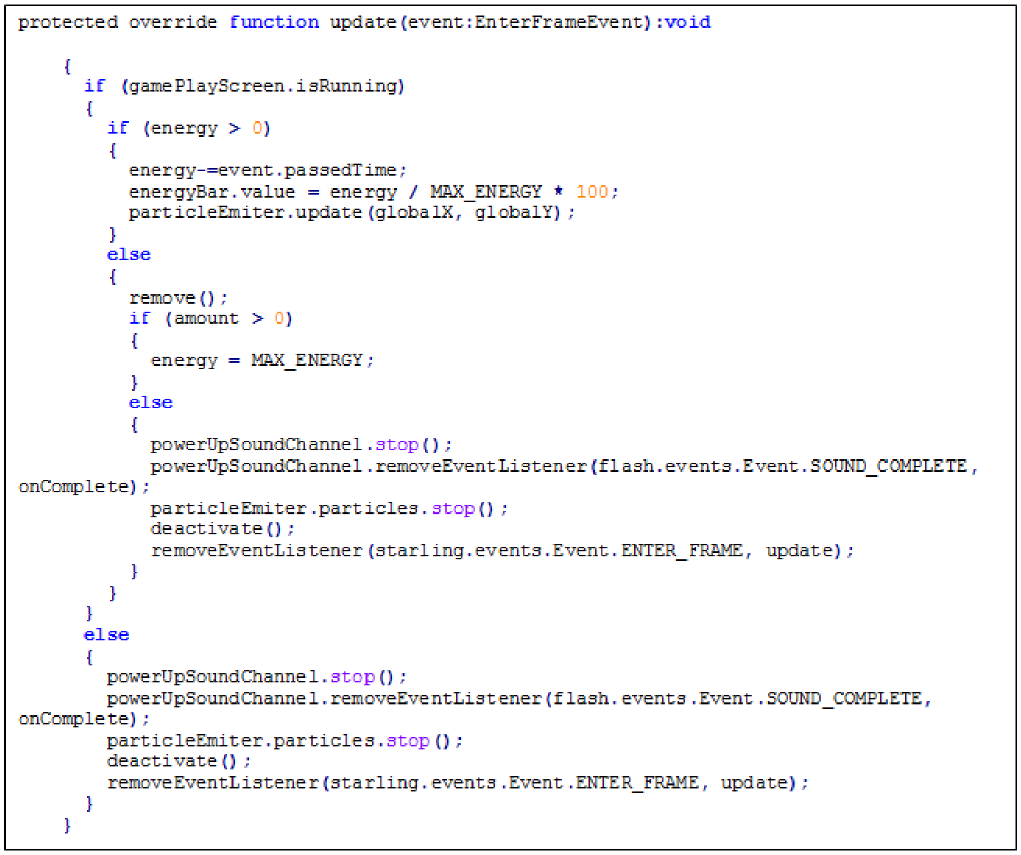


```
  protected override function run(event:TouchEvent):void

    {
      if (isActivated)
      {
        if (event.getTouch(stage) != null)
        {
          var touch:Touch = event.getTouch(stage);

          globalX = touch.globalX;
          globalY = touch.globalY;

          if(touch.phase == TouchPhase.BEGAN)//on finger down
          {
            PowerUpMode.mode = powerUpName;
            particleEmiter.particles.start();
            addEventListener(starling.events.Event.ENTER_FRAME, update);
            powerUpSoundChannel = Game.playSfx(SfxConstants.fire);
            powerUpSoundChannel.addEventListener(flash.events.Event.SOUND_COMPLETE,
onComplete);
            isDown = true;
          }

          if(touch.phase == TouchPhase.ENDED) //on finger up
          {
            particleEmiter.particles.stop();
            removeEventListener(starling.events.Event.ENTER_FRAME, update);
            powerUpSoundChannel.stop();

powerUpSoundChannel.removeEventListener(flash.events.Event.SOUND_COMPLETE,
onComplete);
            deactivate();
            isDown = false;
          }
        }
      }
    }
```

Fig. 4.23: MagnifyingGlass's run method code

The run method keeps track of the player's finger touching down and its position. When the finger is down, the static string property mode found in the PowerUpMode class gets set to the name of the power-up that is currently running. This static property gets read by the bug class to let it know how to react when a power-up is activated. The run method also starts the MagnifyingGlass's particle emitter and loops the power-up sound channel. An enter frame event listener gets added, which calls the update method every frame. When the finger is lifted up, the particle emitter and power up sound channel stops, the event listeners are removed, and the power-up gets deactivated.

Figure 4.24 shows the code for the MagnifyingGlass's update method.



```
protected override function update(event:EnterFrameEvent):void

    {
      if (gamePlayScreen.isRunning)
      {
        if (energy > 0)
        {
          energy-=event.passedTime;
          energyBar.value = energy / MAX_ENERGY * 100;
          particleEmiter.update(globalX, globalY);
        }
        else
        {
          remove();
          if (amount > 0)
          {
            energy = MAX_ENERGY;
          }
          else
          {
            powerUpSoundChannel.stop();
            powerUpSoundChannel.removeEventListener(flash.events.Event.SOUND_COMPLETE,
onComplete);
            particleEmiter.particles.stop();
            deactivate();
            removeEventListener(starling.events.Event.ENTER_FRAME, update);
          }
        }
      }
      else
      {
        powerUpSoundChannel.stop();
        powerUpSoundChannel.removeEventListener(flash.events.Event.SOUND_COMPLETE,
onComplete);
        particleEmiter.particles.stop();
        deactivate();
        removeEventListener(starling.events.Event.ENTER_FRAME, update);
      }
    }
```

Fig. 4.24: MagnifyingGlass's update method code

The update method updates the state of the energy property, which keeps track of how long the user can keep the magnifying glass active. When the energy reaches zero the amount property will be decremented. The energy will get reinitialized if the amount remains greater than zero. If the amount is zero, then the magnifying glass will deactivate. When the energy is greater than zero, update will call the particleEmitter's update and give it the current touch positions. This allows the particle effect to follow the finger as the user drags it. Update will also deactivate the magnifying glass if the gameplay screen has entered the game over state.

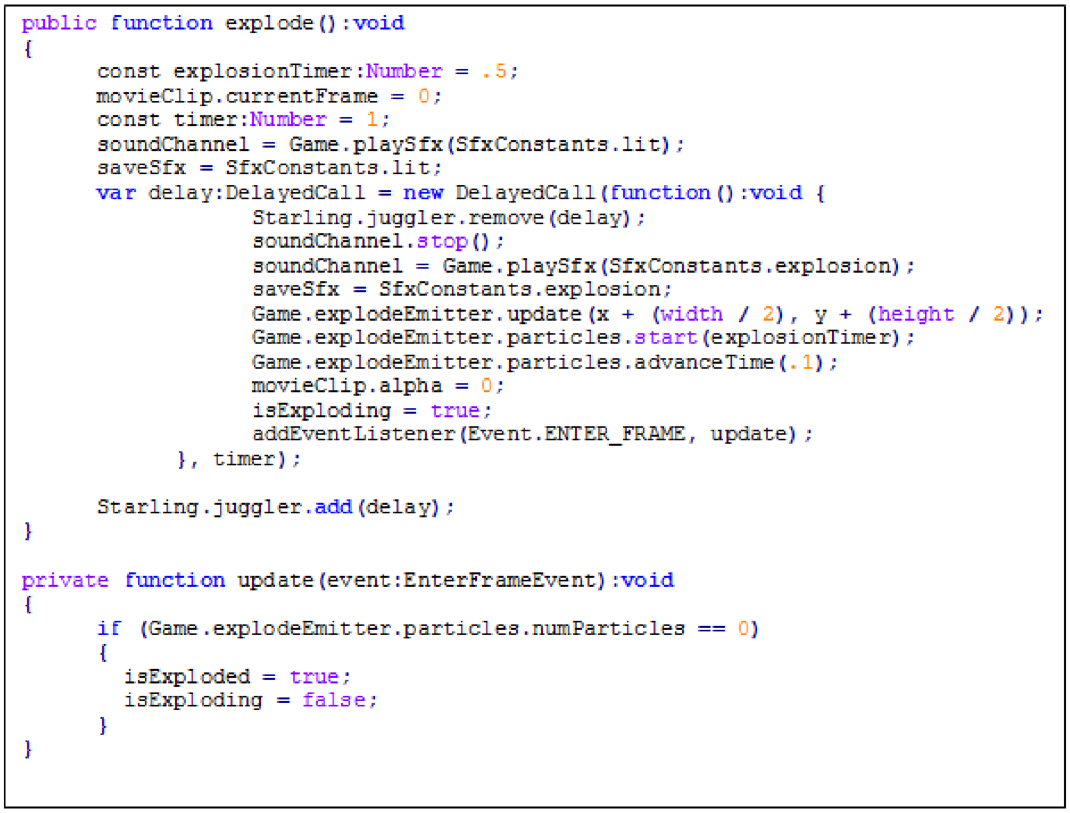
The Swatter's run method sets the position of its quad and swatter properties based on the touch position and makes them visible when the user's finger is down.

The quad property is a Quad object and is used as a collision rectangle for the swatter. The swatter is a image that displays the swatter's texture. The run method also adds a delay call event that waits a fraction of a second before deactivating the power-up. The Swatter class does not implement update method.

The SugarCube's run method creates a new Candy object, sets its position, adds it to the playAreaSprite's candy layer, and pushes it onto an array when the user's finger is down. The update method calls update on the first candy in the array. When update sees that the candy isGone flag is set to true, it will remove it from the array, remove it from the play area sprite, and finally dispose of it.

The FireCracker's run method listens for a touch event by the user, pushes a new FireCrackerSprite object onto an array, sets its position, adds it to the playArea-Sprite's firecracker layer, and calls its explode method. The FireCracker's update method will dispose of the FireCrackerSprite objects if their isExploded flag is set.

The FireCrackerSprite class is mainly composed of a movie clip for the animation, a quad for the collision rectangle, and a particle emitter for the particle effects. Figure 4.25 shows the code for the FireCrackerSprite's explode and update method.

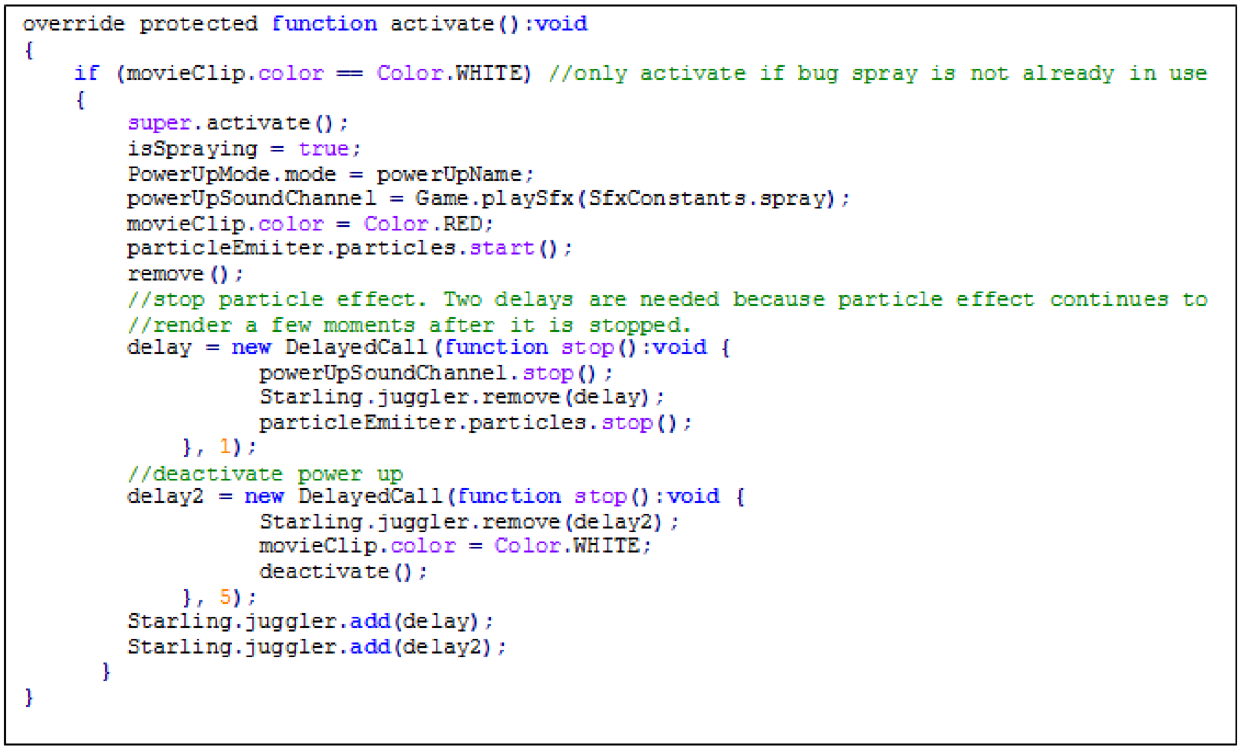


```
public function explode():void
{
        const explosionTimer:Number = .5;
        movieClip.currentFrame = 0;
        const timer:Number = 1;
        soundChannel = Game.playSfx(SfxConstants.lit);
        saveSfx = SfxConstants.lit;
        var delay:DelayedCall = new DelayedCall(function():void {
                        Starling.juggler.remove(delay);
                        soundChannel.stop();
                        soundChannel = Game.playSfx(SfxConstants.explosion);
                        saveSfx = SfxConstants.explosion;
                        Game.explodeEmitter.update(x + (width / 2), y + (height / 2));
                        Game.explodeEmitter.particles.start(explosionTimer);
                        Game.explodeEmitter.particles.advanceTime(.1);
                        movieClip.alpha = 0;
                        isExploding = true;
                        addEventListener(Event.ENTER_FRAME, update);
                }, timer);

        Starling.juggler.add(delay);
}

private function update(event:EnterFrameEvent):void
{
        if (Game.explodeEmitter.particles.numParticles == 0)
        {
          isExploded = true;
          isExploding = false;
        }
}
```

Fig. 4.25: FireCrackerSprite's explode and update method code

The explode method has a delayed call event handlers that will wait one second before executing its callback. The callback plays the explosion sound effect, updates the explode emitter, and adds an enter frame event listener that calls its update method every frame. The update method checks to see when there are no longer any particles in the emitter and will then set the isExploded flag to true. This indicates that the FireCrackerSprite is no longer in use and the FireCracker class can dispose of it.

The BugSpray is the only power-up that does not implement the run and update method. The functionality of the bug spray is all done through its activate method. When the player touches the bug spray's movie clip on the HUD the power-up is activated and is immediately in use. Figure 4.26 shows source code for the BugSpray's activate method.



```
override protected function activate():void
{
    if (movieClip.color == Color.WHITE) //only activate if bug spray is not already in use
    {
        super.activate();
        isSpraying = true;
        PowerUpMode.mode = powerUpName;
        powerUpSoundChannel = Game.playSfx(SfxConstants.spray);
        movieClip.color = Color.RED;
        particleEmiiter.particles.start();
        remove();
        //stop particle effect. Two delays are needed because particle effect continues to
        //render a few moments after it is stopped.
        delay = new DelayedCall(function stop():void {
                    powerUpSoundChannel.stop();
                    Starling.juggler.remove(delay);
                    particleEmiiter.particles.stop();
            }, 1);
        //deactivate power up
        delay2 = new DelayedCall(function stop():void {
                    Starling.juggler.remove(delay2);
                    movieClip.color = Color.WHITE;
                    deactivate();
            }, 5);
        Starling.juggler.add(delay);
        Starling.juggler.add(delay2);
      }
}
```

Fig. 4.26: BugSpray's activate method code

The activate method starts the particle emitter, plays the sound effect, and changes the movie clip color to red, which indicate to the player that the bug spray is currently in use and cannot be activated again until it finishes. There are also two delay call event handlers, the first one stops the particle emitter and the second one will deactivate the power-up.

4.9 DropItem Class

The DropItem class is used to add more power-ups to the HUD or to replenish a candy's HP. Figure 4.27 is a UML class diagram of the DropItem class.

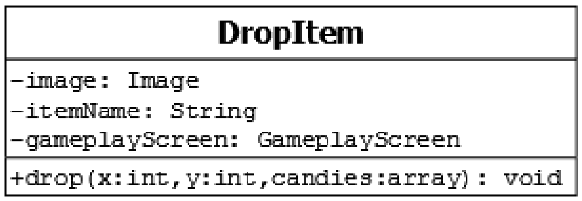

DropItem
-image: Image
-itemName: String
-gameplayScreen: GameplayScreen
+drop(x:int,y:int,candies:array): void

Fig. 4.27: UML diagram of the DropItem class

The player accumulates power-ups and replenishes candy by killing ladybugs. The LadyBug class instantiates a new DropItem object every time it is initialized. The DropItem class assigns a string with a randomly chosen name of a power-up or candy. This string determines two actions: which type of power-up to add to the HUD, or if it should replenish a candy. These actions happen when its drop method is called. The drop method gets called inside the LadyBug's die method.

4.10 ParticleEmitter, PDparticleSystem, and PexLoader Classes

The ParticleEmitter class provides an interface for the particle system in the game. The particle system is an instances of Starling's PDparticleSystem class. The data for the particle system is loaded by the PexLoader class. Figure 4.28 is a UML class diagram of these three classes.

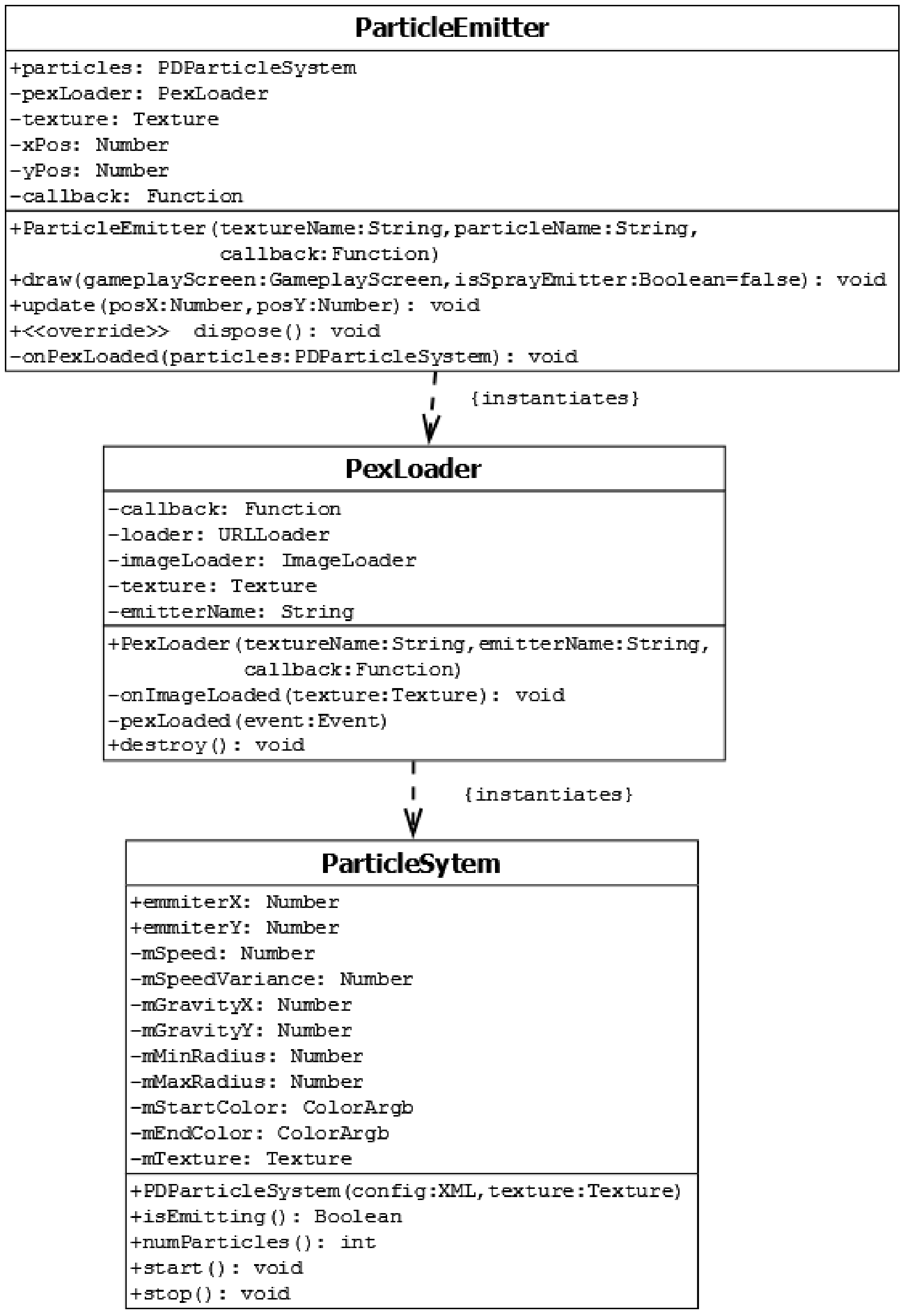
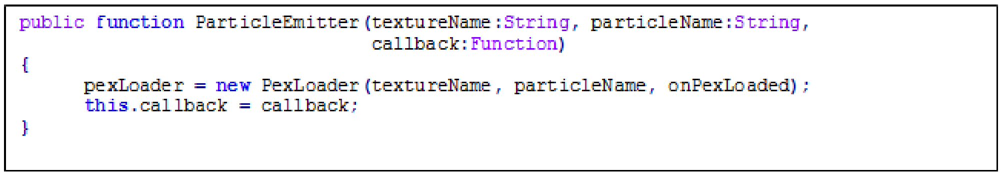
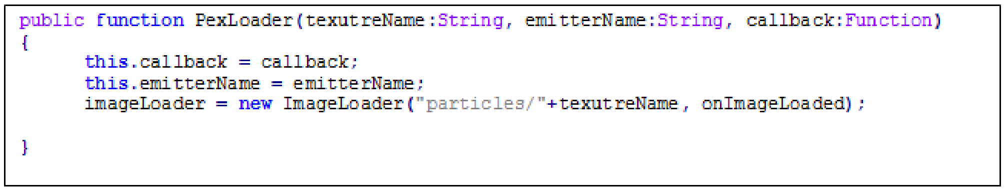


Fig. 4.28: UML Diagram of the ParticleEmitter, PDparticleSystem, and PexLoader classes

The ParticleEmitter class is comprised of a PexLoader object, a Texture object, a Starling's PDparticleSystem object called particles, and two variables to keep track of the x and y positions of the emitter. The ParticleEmitter's constructor loads the assets by creating an instance of PexLoader. Figure 4.29 shows the constructors for the ParticleEmitter and the PexLoader.



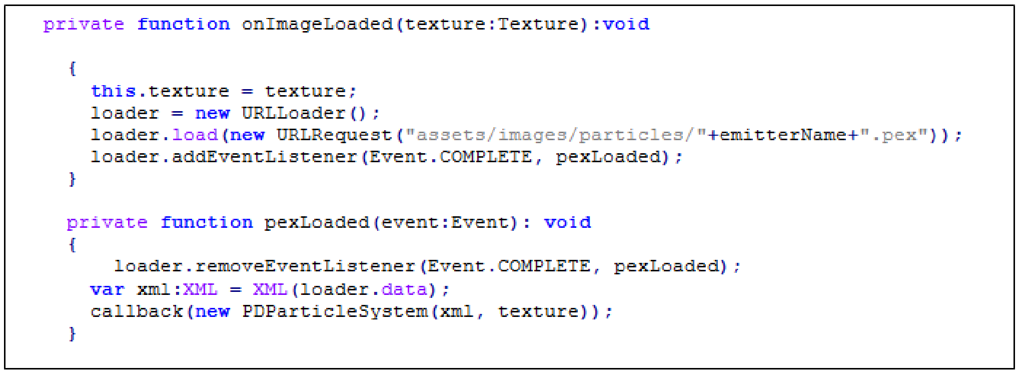
```
public function ParticleEmitter(textureName:String, particleName:String,
                                callback:Function)
{
        pexLoader = new PexLoader(textureName, particleName, onPexLoaded);
        this.callback = callback;
}
```



```
public function PexLoader(texutreName:String, emitterName:String, callback:Function)
{
        this.callback = callback;
        this.emitterName = emitterName;
        imageLoader = new ImageLoader("particles/"+texutreName, onImageLoaded);

}
```

Fig. 4.29: ParticleEmitter and PexLoader's constructors

The PexLoader loads the texture and XML file that defines the particle behavior. The image and XML data gets loaded in sequences. Figure 4.30 shows the code of the callbacks for each of these loaded assets.

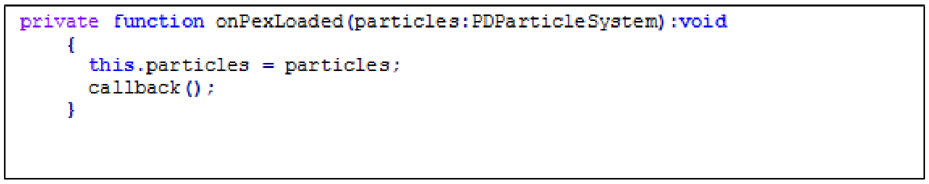


```
 private function onImageLoaded(texture:Texture):void

   {
     this.texture = texture;
     loader = new URLLoader();
     loader.load(new URLRequest("assets/images/particles/"+emitterName+".pex"));
     loader.addEventListener(Event.COMPLETE, pexLoaded);
   }

   private function pexLoaded(event:Event): void
   {
       loader.removeEventListener(Event.COMPLETE, pexLoaded);
     var xml:XML = XML(loader.data);
     callback(new PDParticleSystem(xml, texture));
   }
```

Fig. 4.30: PexLoader's callback methods

Once the final load is complete, the pexLoaded method creates a PDParticleSystem object, which takes the XML and texture data. A callback is then invoked that takes the instance of PDParticleSytem. This callback is the onPexLoaded method of the

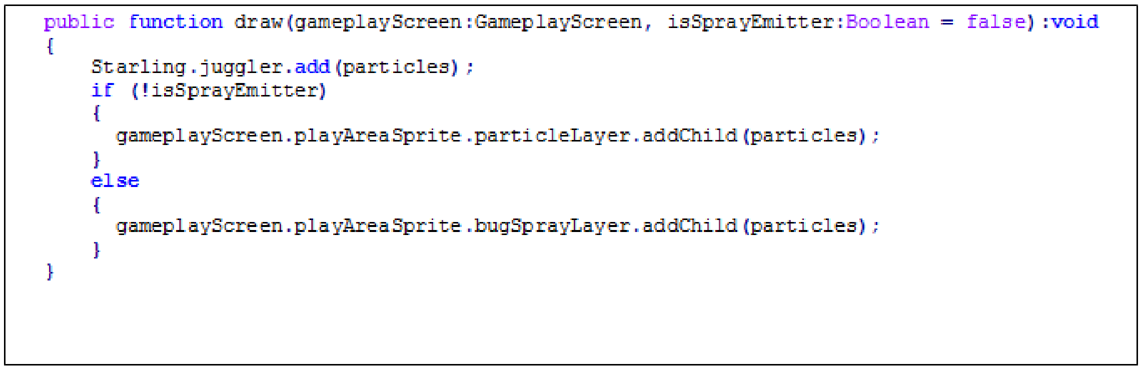
ParticleEmiter class, which is shown in Figure 4.31.



```
private function onPexLoaded(particles:PDParticleSystem):void
    {
      this.particles = particles;
      callback();
    }
```

Fig. 4.31: PexLoader's onPexLoaded method

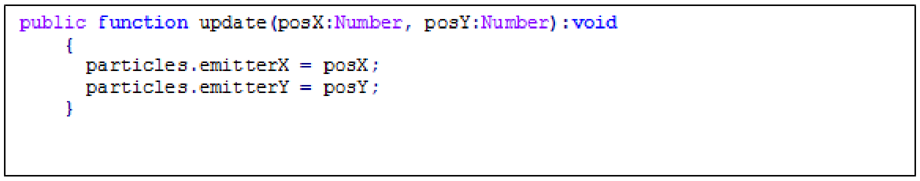
The onPexLoaded method gets the PDparticleSystem instance and assigns it to the particles property. It will then invoke its callback method that was passed in from the ParticleEmitter's constructor. The ParticleEmitter's draw method adds its particles property to the particle layer on the GameplayScreen's playAreaSprite. This method is shown in Figure 4.32.



```
public function draw(gameplayScreen:GameplayScreen, isSprayEmitter:Boolean = false):void
{
    Starling.juggler.add(particles);
    if (!isSprayEmitter)
    {
      gameplayScreen.playAreaSprite.particleLayer.addChild(particles);
    }
    else
    {
      gameplayScreen.playAreaSprite.bugSprayLayer.addChild(particles);
    }
}
```

Fig. 4.32: ParticleEmitter's draw method

This method will draw the particles on the screen in the appropriate draw order. The update method simply updates the position of the particles. Figure 4.33 shows the code for this method:



```
public function update(posX:Number, posY:Number):void
    {
      particles.emitterX = posX;
      particles.emitterY = posY;
    }
```

Fig. 4.33: ParticleEmitter's update method

The particles property is an instance of Starling's PDParticleSystems. The emitterX and emitterY properties keep track of the particles position. It also has a start and stop method, which is used to render the particles. These two methods get called directly on the particles property by other classes elsewhere in the application.

5. CONCLUSION AND FUTURE DIRECTION

5.1 Conclusion

It has been a long and challenging journey but we were able to complete a solid mobile game. Although there were many revisions during our development process, I believe the finished version of ANTics met and went beyond our requirements and expectations. I was able to develop a solid software system design that can be extended upon and reused to build similar tower defense games. The application contains fully functional components, such as an asset management system for loading 2D art and sound assets, a screen management system that can be easily extended, an AI system geared for tower defense type behaviors, and a customizable user interface. Furthermore, thanks to the flexibility of Adobe AIR, the codebase can run on multiple devices with different operating systems.

5.2 Future Direction

ANTics is a fully completed and published game. However, there are always extra features that could be added. Once we gain an audience for ANTics we will add additions that the community asks for. There also might be more development bugs that may have gone unnoticed during testing. I will continually provide patches with bug fixes if any should arise.

There are currently a paid and free versions of ANTics on the Google Play store. ANTics has also been accepted by Apple to be marketed on their app store. We have also decided to target Windows PC as well. We intend to try to get a Windows version on markets like Desura and Steam. Although finishing a completed app and getting it on various markets is a big accomplishment, it is still only half the battle. Currently ANTics has virtually no visibility on Google Play. Without proper advertising, ANTics will remain buried under the millions of other apps on the market. There is no reason to believe that the same won't be true when ANTics is published on the other distribution services. Our next task is clear, we must leave the realm of development and enter the realm of advertising.

REFERENCES

[1] Adobe AIR. [Online]. Viewed 2015 February 26. Available: http://www.adobe.com/products/air.html.

[2] FlashDevelop Open Source Code Editor. [Online]. Viewed 2015 March 13, 2015. Available: http://www.flashdevelop.org .

[3] Starling The Cross Platform Game Engine. [Online]. Viewed 2015 April 20. Available: http://gamua.com/starling.

[4] J. Tynjala. Feathers. [Online]. Viewed 2015 April 8. Available: http://feathersui.com.

[5] J. Tynjala. Feathers 2.1.1 Help. [Online]. Viewed 2015 April 8. Available: http://feathersui.com/help/themes.html.

[6] Adobe Flash Platform. [Online]. Viewed 2015 March 7. Available: http://help.adobe.com/en_US/as3/dev/index.html .

[7] Object pool pattern. [Online]. Viewed 2015 February 12. Available: http://en.wikipedia.org/wiki/Object_pool_pattern.

[8] Feathers 2.1.1 API Reference. [Online]. Viewed 2015 April 8. Available: http://feathersui.com/api-reference/.