Game Design Document (GDD)

Letter Storm
GG Productions

Version 3.0 Created: December 6th, 2014

Version 2.0 Created: November 6th, 2014

Version 1.0 Created: October 16th, 2014

Last Updated: December 6th, 2014

Table of Contents:

1: Executive Summary

1.1 About this Document

This document describes the game design of the Project: Letter Storm. This is a living document and can be changed at any time at the discretion of the developers.

1.2 Motivation

It is well known that if a child does not find learning software fun to use they simply will not use it. In this sense, software that advertises itself as educational yet is panned by its target audience is somewhat paradoxical, for you cannot teach someone if they are not paying attention. Thus it can be concluded that, for educational software to be successful, it must engage children over the long-term while simultaneously teaching them. Our team has set out to develop a product that meets these requirements: Letter Storm, a game that is both entertaining and educational.

1.3 High Concept

Letter Storm is an educational top-down shooter PC game that teaches you how to spell by challenging you to defeat monsters to collect letters, and using those letters to spell out and defeat bosses.

1.4 Core Gameplay

Letter Storm's core gameplay is that of a 2D shooter game. Players must proceed through the game without losing all of their given set of finite health. To defend themselves, players can shoot at monsters in the shape of alphabetical letters. Upon being defeated, these monsters turn into collectible items in which the player can collect into his or her inventory. The collected letters can then be used during a boss fight, in which the player must "shoot the letters" to "spell" and defeat the boss at the end of the stage. Other elements have been incorporated into our design. They include elements such as obstacles, power ups, and a running score. A user can also input a customizable word list in order to challenge a player in spelling specific words.

1.5 Genre

Letter Storm is categorized as a Shoot 'em Up, Top-down Shooter, and (Spelling) Educational PC platform game.

1.6 Target Audience

While Letter Storm can be used by anyone, the target audience is early elementary school students. The intended audience for Letter Storm are children, both boys and girls, between ages 4 to 8 for gameplay, and educators and parents looking to teach their students or children how to spell.

1.7 Team Members / Jobs / Contact Info

Team Member	**Role**	**Contact**
Nabil Lamriben	Project Leader / Design Leader / Environment & integration Leader	lamribenn@gmail.com
David Lustig	Backup Project Leader / Implementation Leader	davelustig@hotmail.com
James Raygor	QA Leader	jamraygo@bu.edu
Jeannie Trinh	Requirement Leader	jtrinh@bu.edu
Paul Pollack	Design Leader/ Configuration Leader	pdpman@bu.edu

2: Environment Overview

2.1 The Unity Engine

2.1.1 Why Unity?

Compared to 5 years ago, there are many game engines available that offer industry level functionality without licensing costs when used for non-commercial projects. The Unreal Engine, CryEngine, Blender Game Engine, and Unity Game Engine are just a few examples of such robust, powerful gaming development tool suites that are utilized by well-known gaming studios for their products.

When choosing the engine to use for Letter Storm, GG Projections evaluated the offerings based on two primary criteria: existing knowledge within the team, and learning curve for those not familiar with it. Various members of the team had a working knowledge of both the Unreal Engine and Unity, eliminating other contestants. Their experience with each engine lead them to conclude that, while the Unreal Engine is considered more powerful and robust than Unity, it is also significantly more complex to develop in. Also, programming in Unreal is done via C++, while Unity utilizes C#, which most of the team was already familiar with. Because of its reduced complexity, it was decided that Unity would be the easiest and most appropriate Engine to use for the development of Letter Storm.

2.1.2 Game Objects, Prefabs and Scenes

Within a Unity project, one must create a scene before anything else. You can create as many scenes as you like and each one is like a blank canvas where you can then position objects and add functionality. Although there is no forced correspondence, we have been using scenes to separate menus and levels in-game.

A scene is merely a two- or three-dimensional space where the developer can experiment with objects. Game objects are themselves the fundamental unit of development in Unity. For the sake of simplicity, we will merely say that Game objects are anything that appears in a scene. What makes each object unique is the components that may be attached to it. We will discuss components in the next section.

One important tool that Unity provides is the ability to turn a game object into a “prefab.” After creating a game object, one may opt to designate it as a prefab, which stores the blueprint necessary to instantiate that object at any time, but the object itself does not exist within the scene infinitely.

2.2 Components

Components are the properties of objects that dictate their appearance and behavior. Some components are more commonly found among different objects such as C# scripts and meshes, while some provide more fine-grained functionality and aren't used as often, such as colliders. Components modularize much of the common functionality found in games and make it trivial for a developer to implement. Some of the more important components are described in this section.

2.2.1 Transforms

Transforms are components that allow manipulation of an object's position, rotation and scale. Every object has a transform component, even the empty ones. This is useful for empty game objects like our spawn points, which, although we do not want the player to be aware of them, must occupy space around the edges of the screen so that enemies may be instantiated as child objects at their position.

2.2.2 C# Scripts

Any programmatic functionality associated with an object must be attached to that object as a component. In order to use the methods provided by Unity in one's own scripts, the class must inherit from MonoBehaviour. Unity provides an Update() method that is called once per frame, which is invaluable for accurately drawing objects to the screen. There are also Awake() and Start() methods that allow for some initialization and setup as soon as the scene exists and as soon as the object exists, respectively. These methods are a nice way to avoid race conditions. Since manager classes such as EnemyGenerator and Context should exist and be ready to use as soon as possible, putting their initialization code in Awake() makes sure that no object whose initialization that takes place in Start() will preemptively attempt to access or modify those variables before they are ready.

2.2.3 Colliders

Event handling plays an important role in any game and ours is no exception. One of the most common types of event triggers is a collision. That is to say, one object enters the space that another object occupies and something is expected to happen. The collider component is a simple way of alerting Unity that this object will, at some point, probably collide with another object and when that happens, the handler code exists to respond appropriately.

2.3 3D Modeling and Image editing

2.3.1 GIMP

The GNU Image Manipulation program, commonly known as GIMP, is powerful open source software for editing images that offers functionality similar to Photoshop. Gimp was used to

create and edit Letter Storm's image assets such as the logos, menu backgrounds, button images, character textures, land textures, and the like.

GIMP offers many robust features, including layers, custom brushes, image distorting filters, color balancing, and various pixel selection tools and options. While it is too complex of a tool to cover in detail here, if you wish to read more about it, please see its online documentation found at http://docs.gimp.org/2.8/en/

2.3.2 Blender

Blender is a free open source program that is used for modeling and animating 3 dimensional models. It features a wide variety of 3D modeling tools, image and video editing, and it even has a built-in game engine.
http://www.blender.org/foundation/history/

2.3.2.1 Blender features used in making Letter Storm

Modeling: Creating a three-dimensional object using vertices. A vertex is represented as a single point with x ,y and z coordinates relative to a center position. Clusters of 3 or more vertices represent a surface and a cluster of surfaces constitute a "mesh". Every game object seen within Letter Storm, with exclusion of all GUI objects, is created as a mesh in Blender.

UVM Wrapping: Projecting an image onto a surface. A mesh can be cut into sub groups, and each subgroup can have a different image projected onto it. This is used to create texture variety and enhances a mesh's visual richness. All colors and designs seen on visible game objects within Letter Storm are created using UVM wrapping.

Rigging: Rigging involves attaching a mesh to an object that will deform it when moved. Such an object is called a bone. Bones are represented by a vector with a base at the joint or pivot of the bone. There are two important properties of a bone:

1) Bones can record their orientation and change in orientation over time.
2) A sub mesh can be parented (attached) to a bone.

3: Game Overview

3.1 Overview

The main character, Albert, has discovered a lab overrun by Letter Monsters. Higher motivations drive him to attempt to rid the area of these creatures, collecting the letters associated with them along the way. During his journey, he will encounter large creatures that can only be defeated by spelling a word correctly using the letters collected. In this way,

the user will learn how to spell while enjoying the gameplay elements of a 2D scrolling shooter.

3.2 Story

It was a dark and stormy night, and two siblings, Albert and Annie, were caught out in it. As the wind howled and the trees creaked, they plodded through the nearly blinding rain toward a hope of refuge in the distance. The dark, rectangular shape on the horizon would have been foreboding under any other circumstance, but to them it symbolized shelter and warmth. With each passing minute, hopes were strengthened as the structure slowly grew larger and clearer.

Then, their moment finally came. After journeying for what seemed to be hours, the two found themselves before the large, unsecured iron gates of what appeared to be an old college building of sorts. It took only one look at the rusted iron bars and crumbling brick walls to see that time was no friend to this work of man. Yet, they were not deterred. Perhaps they could take temporary shelter inside until the storm passed, or at least such was the thought.

Slowly they worked their way through the gates and to the facility's oversized engraved doorway. Whatever fantasies of danger it would have stirred up under normal circumstances fell prey to the desperation of their plight, and so, without much thought, they eagerly pressed the two iron-clad doors open. Their entrance did not go unannounced, though, as the age-old hinges let out a moan strong enough to shake one's bones. And that is when the trouble started.

Looking into the dark interior, one could make out a large foyer walled in by deep hallways and a large, central staircase. The faint but growing noise similar to that of frantic footsteps emanated from the core of the structure. Then they saw him: A scholarly gentleman with his hair amuck and his clothes wrinkled and stained came running madly down the staircase right toward them. "Help! Help!" he yelled, gasping for breath between his words. "Something has gone terribly wrong. The sound of your entry startled me into a grave accident. My lab is now run amuck with creatures not from this world."

At the sound of such news, the two siblings began to feel fear well up within them for the first time that night. "It's alright to be afraid, but do not let fear stop you from doing what is right. I have here the book that the creatures came from and the pencil used to design them. If you use them correctly, you should be able to vanquish the creatures and I will be forever in your debt."

Hesitantly accepting their fate, Albert and Annie took up the tools and began their adventure into the LetterStorm.

3.3 Characters

Albert: Older brother of Annie, Albert is widely recognized as a geek. But don't let his square glasses fool you; his quick wit it more than a match for any challenge.

Annie: Younger sister of Albert, Annie is the most shy girl in her class. Her curiosity is her greatest strength, empowering her to overcome her shyness and be brave in the face of danger.

Charamons: Accidental creations of Professor P, these letter-shaped creatures now roam his lab and the surrounding hillsides. While most of them are docile, the six vowels are carnivorous and easily agitated. They can be defeated using the magic pencil and their pure form collected by the magic journal.

Professor P: Professor P is a highly experienced lab scientist devoted to the greater good. Unfortunately, his experiments sometimes get a little out of control…

Wordians: When a few charamons join together in the correct sequence, they can become a more powerful creature called a wordian. Wordians are rare but also difficult to defeat. They are most vulnerable to the pure forms of the letters composing them, when fired in the correct sequence.

3.4 Level Overview

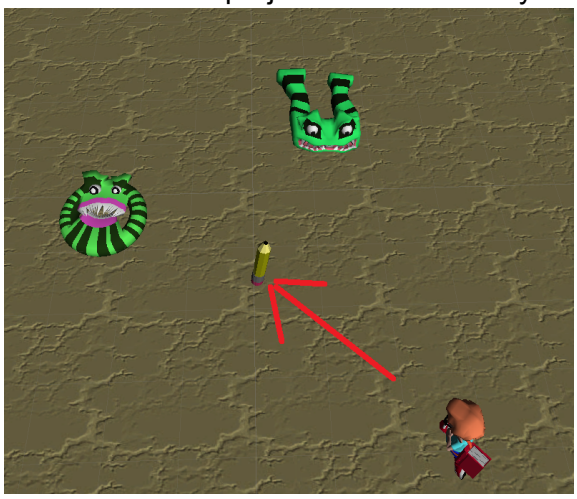
When the user starts the game, he selects a set of words he wants to learn how to spell, or a lesson. Each level is associated with one word in this lesson, with early levels in the game based on easy words and the latter levels on harder words.. The user's avatar starts out at the bottom of the screen facing upward and is automatically moved forward as the background scrolls down. Enemies are generated from the top and sides of the screen based off the characters in the level's word. All enemies attempt to damage the player, some via collision and others via weapons. The user can damage enemies using one or more of his weapons, with each enemy dropping a letter upon defeat. Once all the letters required for the word are collected, enemy generation stops, the boss arrives on-screen, and a word hint is displayed at the top of the screen. The user must defeat the boss by firing the collected letters at him in the right order to spell the level's word.

Since this game is a 2D scrolling shooter, gameplay environments do not play a significant role in the game itself. The game's story is set in and around a large lab building secluded within a lush, natural landscape. Earlier parts of the game will take place on a dirt road through a natural surrounding leading up to the lab. Later levels will take place within the lab building itself. It is difficult to make the environment progression more fine-grained than this, as the number of levels the user will play will be dependent upon which set of words he wants to learn, and he has the ability to change how many words are in any word set.

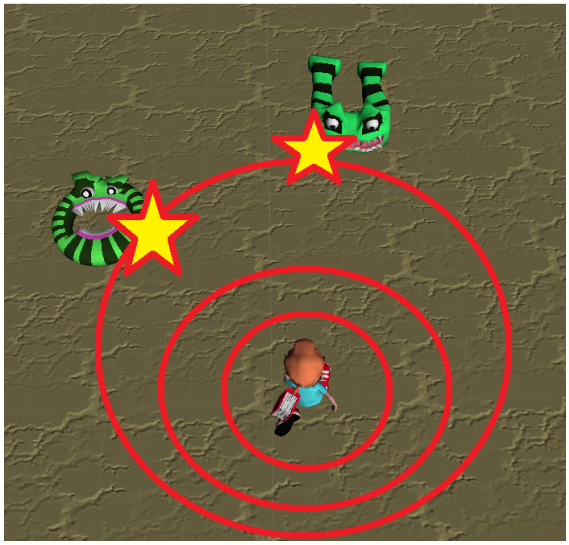
4: Gameplay - Paul/Nabil

4.1 Combat

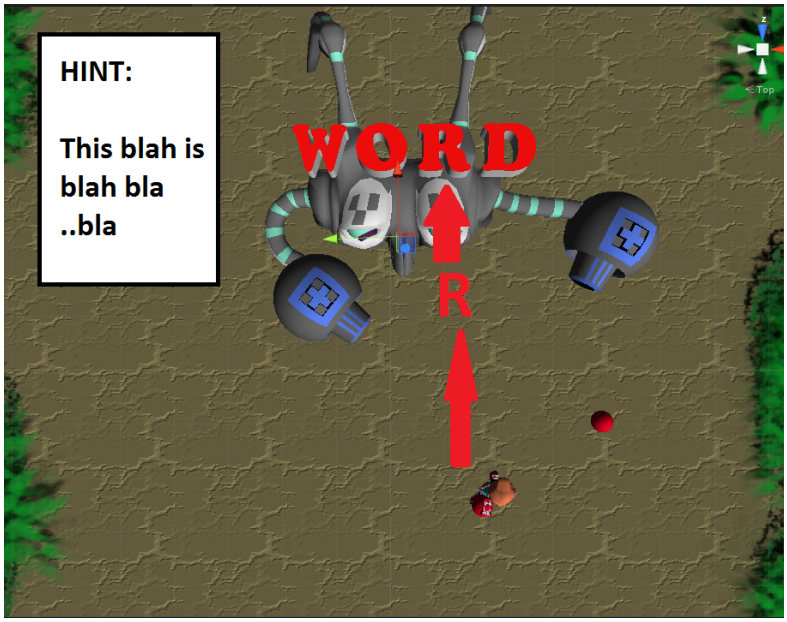
Projectile:
Albert can throw projectiles at an enemy in order to collect a letter drop.



Area damage:
Albert can use a different type of attack such as AOE.



Boss combat:
Albert will have to use the letters collected in the level in order to solve and spell out a word during the boss fight.



4.1.1 The Player Perspective

Each level pits the player against waves of letter monsters. The player is force-scrolled through the level, although within the player's field of vision he may move along a horizontal and vertical axis. The player may throw projectiles in order to defeat letter monsters. When the use-weapon key is pressed, a projectile is instantiated directly in front of the player's current position and travels in a straight line in the direction the player is currently facing until it either collides with an enemy or moves out of view, at which point it is destroyed.

4.1.2 The Enemy Perspective

There are two different types of letter monsters in Letter Storm -- Consonant Monsters and Vowel Monsters. If either Monster type moves below the bottom of the screen or a player's projectile collides with it, it is destroyed. If the player manages to defeat a Monster before it reaches the bottom of the screen, it will drop its associated letter for collection by the user.

Consonant Monsters are unintelligent enemies that move in variations of two basic patterns. Upon spawning, they may either choose to move along the vertical axis from the top of the screen to the bottom, or they may choose to move towards the coordinates the player was located in at the exact moment of its instantiation. Consonant Monsters do not have projectiles or attacks of any kind, and may only cause a player to lose a life if the enemy collides with him.

Vowel Monsters are the more threatening of the two basic enemies and pose a greater challenge to the player. Vowel Monsters may move sinusoidally, rather than follow a simple trajectory through the battlefield and then out of sight. In many cases, they move back and

forth on screen firing their own projectile at the player. If a player collides with either a Vowel Monster or the monster's projectile, he will lose a life.

4.1.3 Bosses

At the end of each level, when the player has collected sufficient letters to spell a word that has been predetermined, a boss will appear -- recognizable as such by its massive size and imposing arsenal. The boss moves in a simple back-and-forth pattern while attacking with projectiles and melee weapons. The player must bombard the boss with projectiles until it goes into a vulnerable state, at which point it will not move or attack and appear dazed. At this time, the player may use the letters he has collected to fire special projectiles in order to spell the boss word correctly. After the boss is hit with the correct letter in sequence, it is no longer in a vulnerable state, once again moving and attacking the player.

4.2 Enemy Management

The Enemy Generator is an object that keeps track of how many enemies currently exist and decides whether or not to spawn enemies, and in what quantity to do so. The generator has an array of Spawn Points, and to create an enemy, will simply instantiate either a Vowel Monster or Consonant Monster as the child of a Spawn Point. Movement is not handled by the generator, but rather by the enemy itself. The frequency at which enemies spawn and the number of which may exist at any given time are determined by the difficulty setting that the player has chosen.

The generator also stores the word associated with the boss for the level, and keeps track of how many of the necessary letters the player has collected before ceasing its normal spawning pattern and instead instantiating the boss.

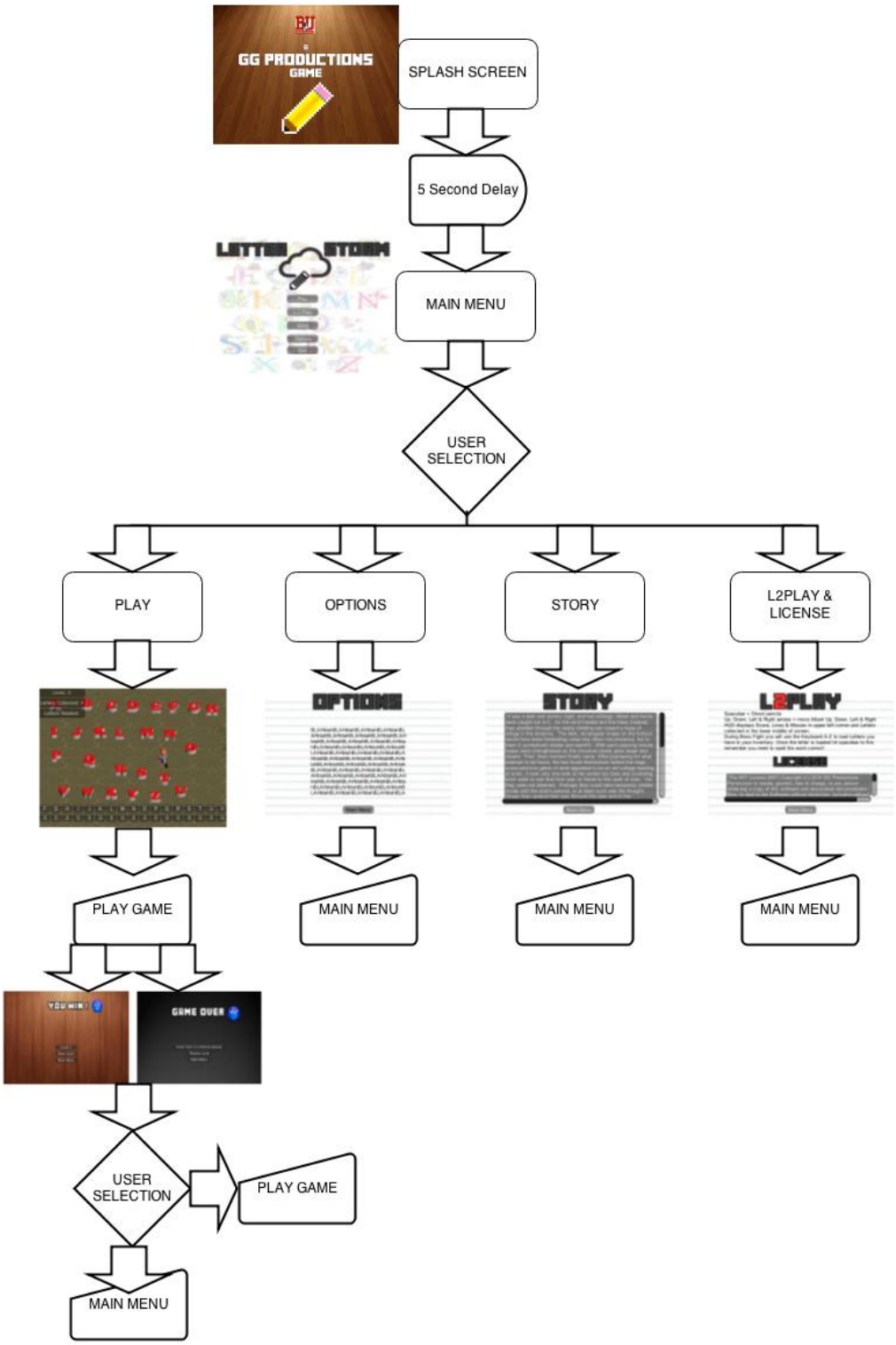
4.3 Collectibles

There are several types of collectibles that will appear as players defeat enemies. Enemies will drop the letters that they resemble, which players will be required to stock up on in order to face the boss. As mentioned previously, the boss will not spawn unless the necessary letters have been collected.

In addition to these, players will have the opportunity to collect powerups, which spawn infrequently and give the player a stronger means of attacking. Some powerups planned include split fire, a one-time-use area-of-effect attack, health, and the ability to slow enemies. The player may also be able to acquire trophies in the form of classic books, which do not alter gameplay in any way but encourage players with completionist tendencies to collect all of the "classics" and fill up their bookshelf.

5: Menu Layout - James

Menu Flow Chart



5.1 Color Scheme

For color scheme we have mainly stuck with warm background colors, however this could change farther down the line.

5.2 Game Title Screen

Currently we have a Splash screen advertising our group name which leads into the “Main Menu / Title Screen”.

5.3 Legal info

The following license will be included with the game, as well as listed on the Credits screen.

“The MIT License (MIT)

Copyright (c) 2014 GG Productions

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without

restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

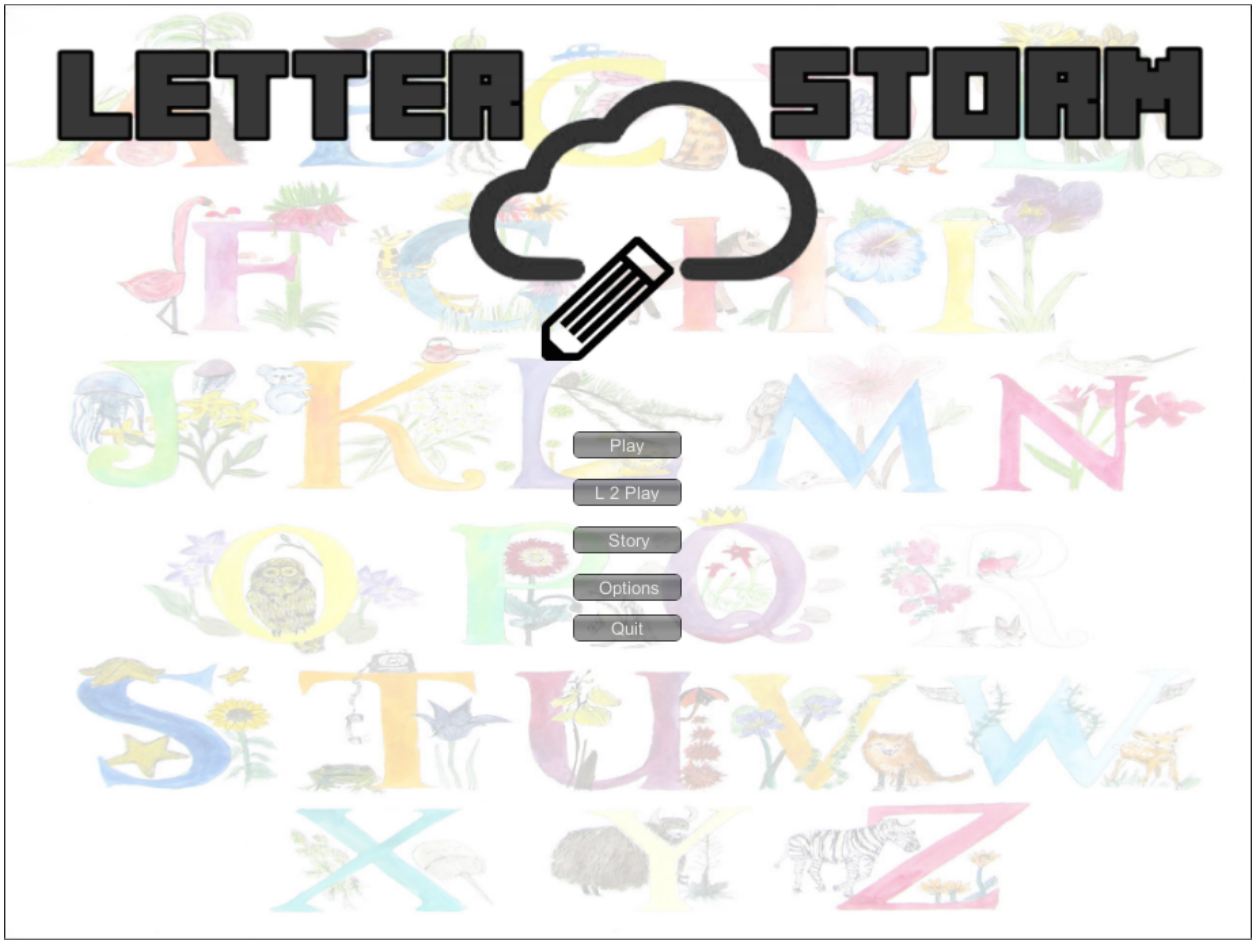
The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.”

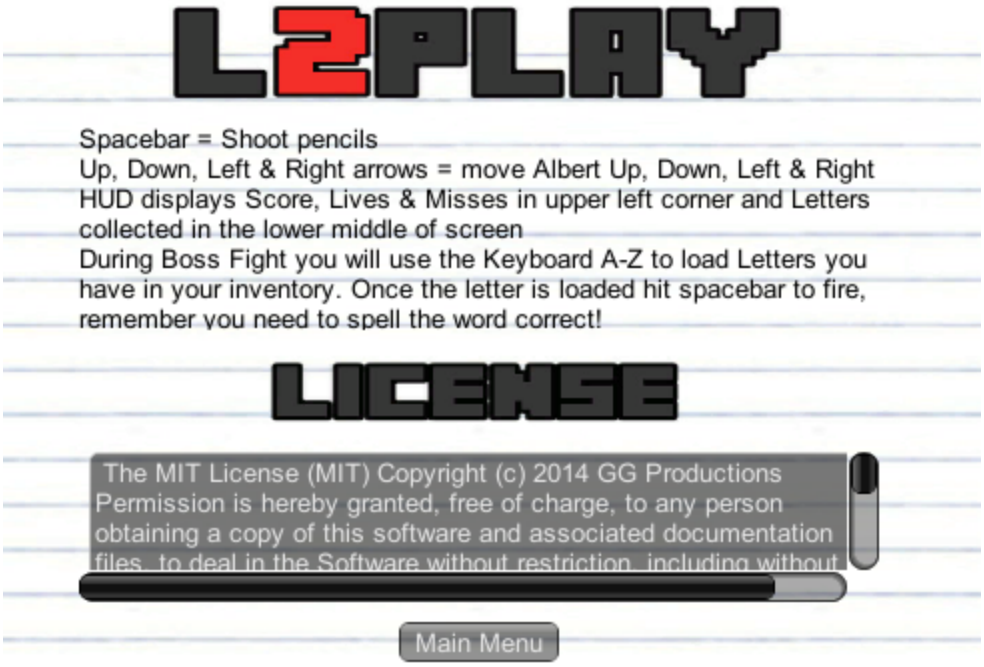
5.4 Menu Screen

The Main Menu screen has 5 major buttons: “Play”, “L2Play” (aka “Tutorial”), “Story”, “Lessons” (aka “Options”), and “Quit”. The “Play” button lets you select the enemy difficulty and which word set you would like to learn, and then start the game from level 1. “Tutorial”, or “L 2 Play” in the prototype below, presents the user with instructions on how to play when clicked. However “L2Play” also houses the license to play the game. “Lessons” is where players can customize what words users can learn throughout the game. “Story” links a page where users can read up on the game’s backstory. Finally, the “Quit” button allows the player to stop and exit the game.

The second iteration prototype of the Main Menu screen is pictured below.

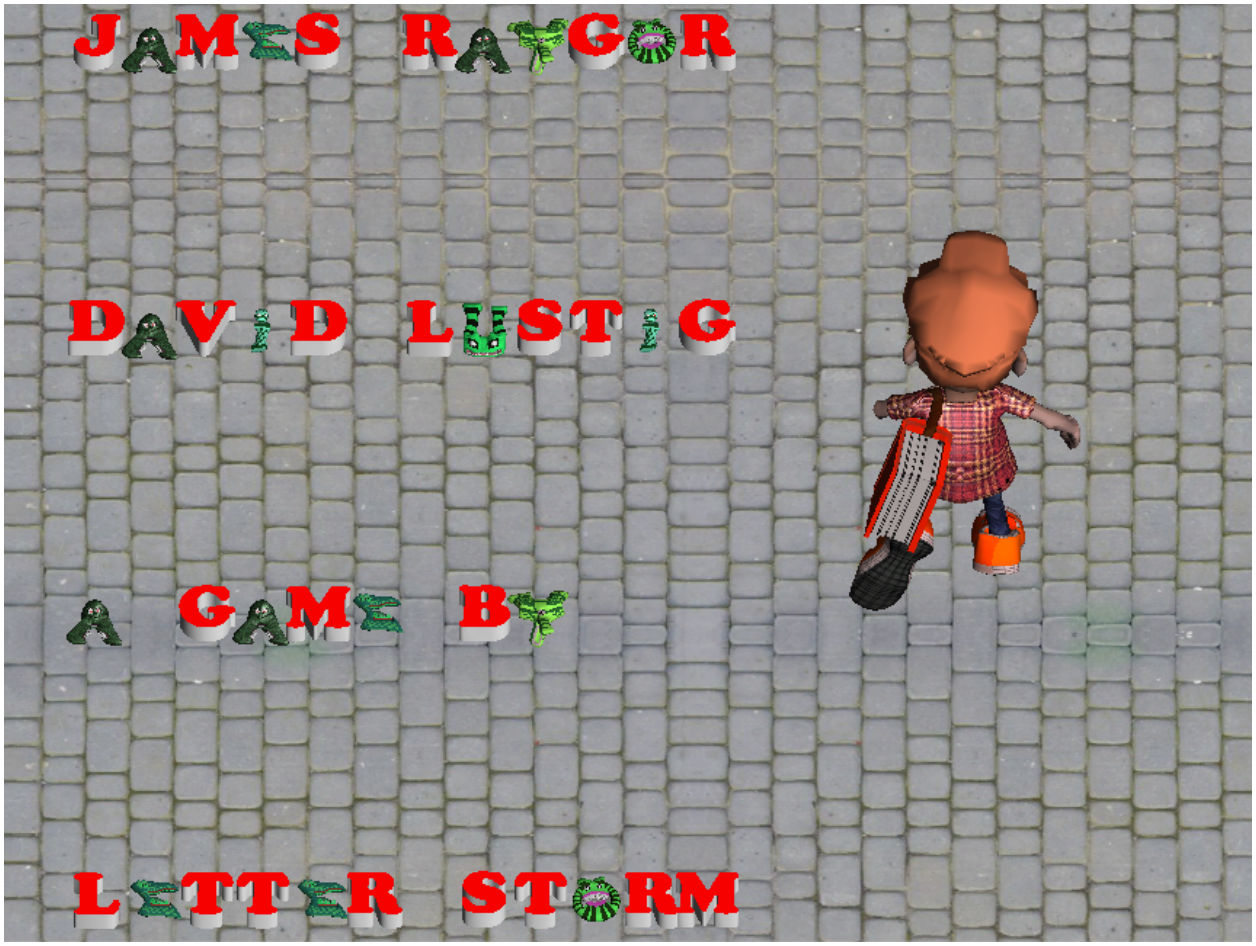

LETTER STORM
Play
L 2 Play
Story
Options
Quit

5.5 Tutorial/Instructions Screen



5.6 Credits Screen

A animated credits screen is displayed after the user has successfully spelled all words in a lesson (ie has beaten all levels associated with that lesson), but before the Win All screen is displayed. It cannot be accessed directly.

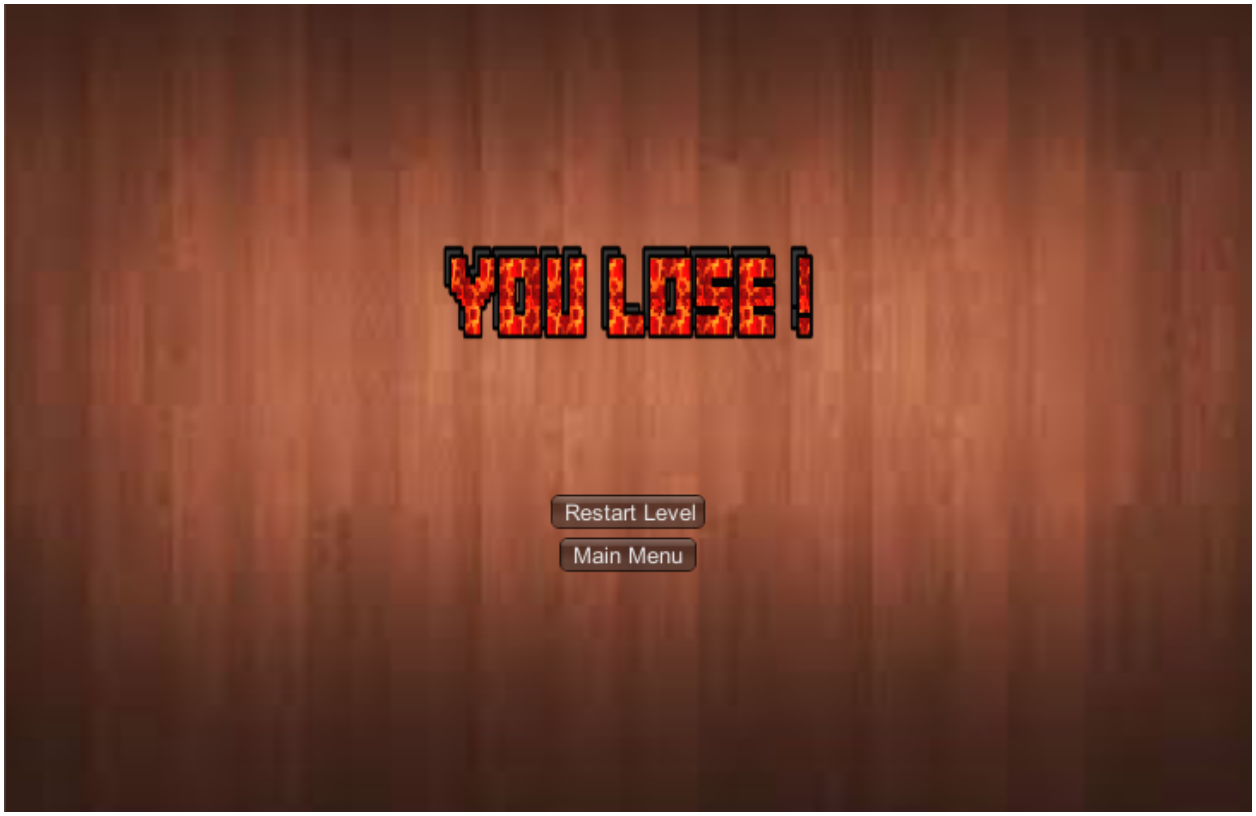


5.7 Game Win Screen

Upon successfully completing a level, the user will be greeted with the Game Win screen. It contains 2 buttons: “Next Level”, and “Main Menu”. “Next Level” loads the next level. “Main Menu” returns the player back to the Main Menu, at which point the player could customize gameplay or quit the Game.

5.8 Game Over Screen

Upon losing all his lives, the player will see the Game Over screen. It contains 2 buttons: “Retry” and “Main Menu”. The “Restart” button allows the user to start the level over. The“Main Menu” button functions identically to that on the Game Win screen.



5.9 High Score Layout

The possibility of score tracking has been discussed, but is considered a low-priority requirement. Should time permit, it might be motivational to the user experience to display a list of high scores on the “Game Over” screen or “You Win” screen, essentially showing your play statistics from the level in relation to other playthroughs.

5.10 Win All Screen

When the user defeats all levels for a given lesson, the Win All screen is displayed. It lists the player’s score, the chosen difficulty level, and all the words the player was tested on. A button is provided to allow the user to navigate back to the main menu.

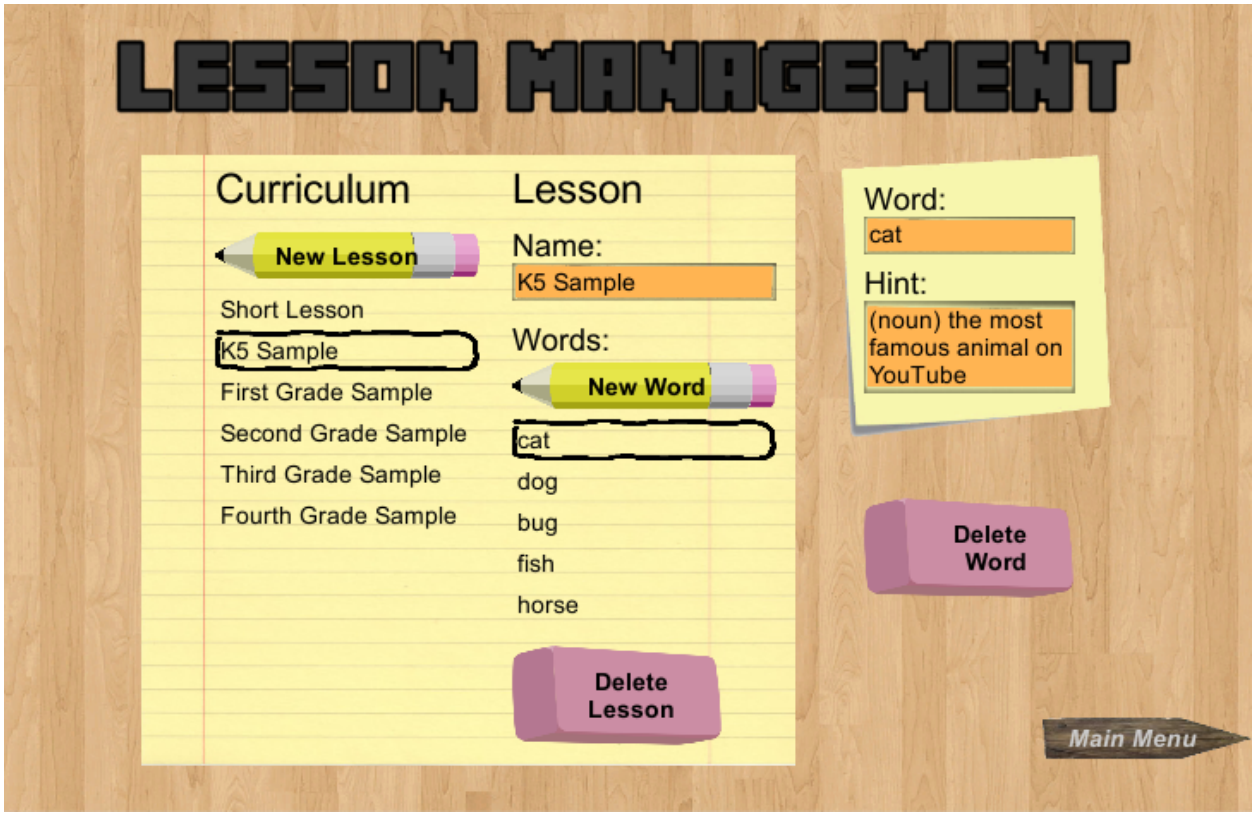
5.11 New Game Screen

The New Game screen is displayed when the user click the “Play” button on the Main Menu. It allows the user to select which set of words he want to learn and the difficulty of the enemies. Upon selection of these, he will be able to launch a new game starting at the first level from the screen.

It was originally hoped that the screen would also allow the user to choose which avatar he wishes to play with, but time constraints have prevented the implementation of multiple playable characters.

5.12 Lesson Management Screen

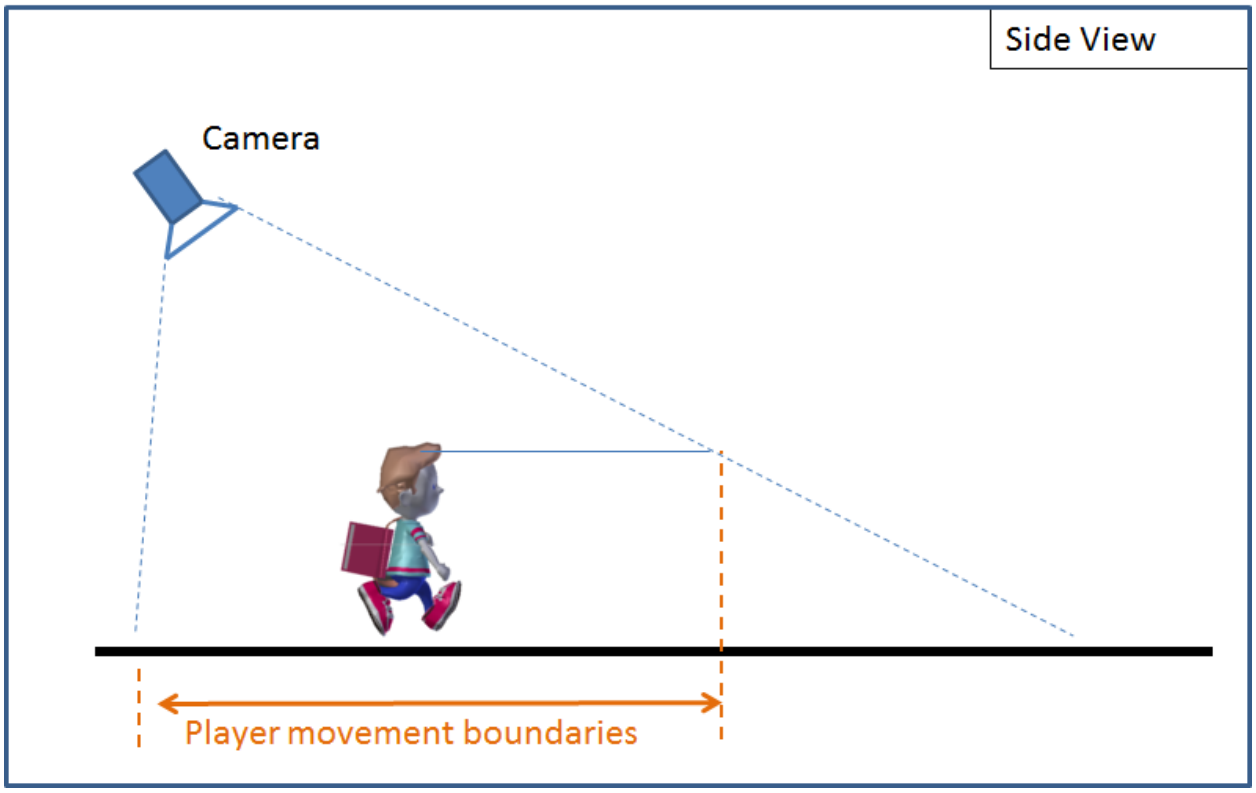
The Lesson Management Screen allows the user to create new word sets or edit existing ones. The user may also specify the hints displayed for each word. This Menu is accessed from the “Main Menu” using the “Lessons” button.


LESSON MANAGEMENT
Curriculum
New Lesson
Short Lesson
K5 Sample
First Grade Sample
Second Grade Sample
Third Grade Sample
Fourth Grade Sample
Lesson
Name:
K5 Sample
Words:
New Word
cat
dog
bug
fish
horse
Delete Lesson
Word:
cat
Hint:
(noun) the most famous animal on YouTube
Delete Word
Main Menu

6: Game Layout

6.1 Camera Setup

The camera remains stationary throughout the entire game. It is placed above the player, and it is tilted slightly to give a 3d effect.



6.2 Game Controls

6.2.1 Player Actions

The player has the following actions available to him:

- Move
  - Forward, into the scene
  - Backward, toward the camera
  - Strafe left with slight angle
  - Strafe right with slight angle
- Attack
  - Use equipped weapon
  - Fire letter at boss

        - Select letter to fire
        - Fire letter
    - Change weapon
        - Change to specific weapon
        - Cycle through weapons

6.2.2 Control Mapping

Upon release, multiple control schemes will be supported. The two most likely to be included are detailed below.

Keyboard-only

- Move
    - Up arrow: Move forward
    - Down arrow: Move backward
    - Left arrow: Move left
    - Right arrow: Move right
- Attack
    - Space: throw pencil, or after boss has spawned throw selected letter
    - A-Z: After boss has spawned, select letter
    - Change weapon
        - number key: change to specific weapon
        - tab: cycle through weapons

Mouse

- Move
    - Main character will follow the mouse cursor
- Attack
    - Left-click: use weapon
    - Fire letter at boss
        - Click the letter in the inventory: select letter to fire
        - Left-click: fire letter
    - Change weapon
        - scroll-wheel: cycle through weapons

6.3 Game Mode(s)

The game will have one playable mode with variable levels of difficulty. There are two factors that affect game difficulty: enemy difficulty and word length. When the user starts a new game, he will be able to select the enemy difficulty (easy, normal, hard) and the lesson to practice. The enemy difficulty affects such properties as the enemy health, enemy speed, and enemy density. It will optionally affect how many lives the player starts out with, and how often he is presented with power-ups. The lesson selected will indirectly affect how many enemies must be defeated before reaching the boss, how many letters must be used to defeat the boss, and how many levels must be completed to win the game.

6.4 Player Count

This game is being designed for playthrough by one player at a time, and thus is intended to be a single-player game.

6.5 Hours of Gameplay

Letter Storm will feature a finite set of levels that can be repeated multiple times. The levels will get harder as the player progresses. While the amount of time spent in the game will be directly related to how many words are in the lesson the user is learning, their complexity, and the chosen enemy difficulty, it is not expected that a level should take more than 10 minutes to complete.

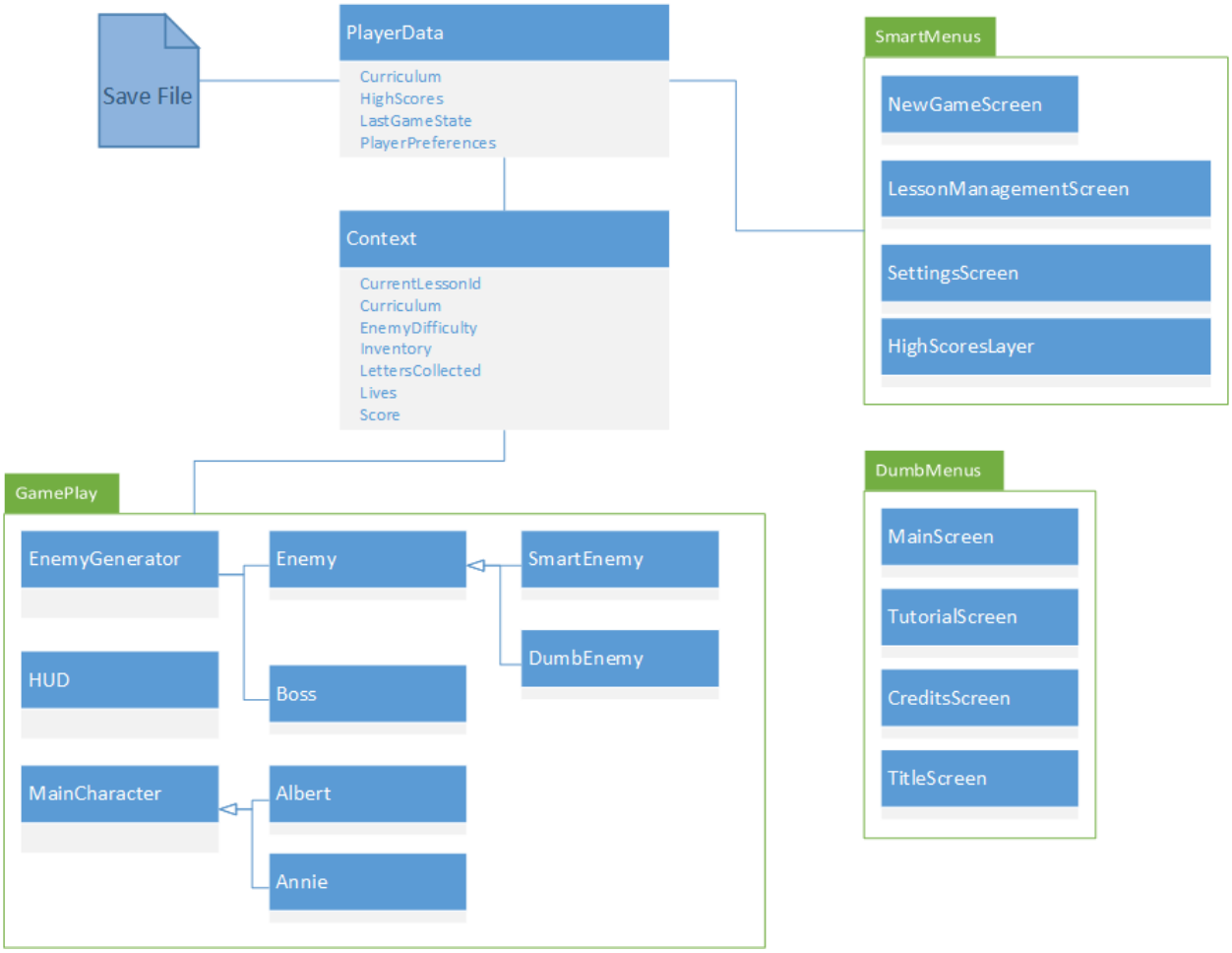
7: Code Design

7.1 Design Model Overview

This code base for Letter Storm is being developed with the MVC architecture in mind. Due to the variation of objects present in a complete game and their varying dependence upon data, though, some objects will simply follow either a VM pattern (no controller). Please see the sections below for more details.

7.2 Data Flow

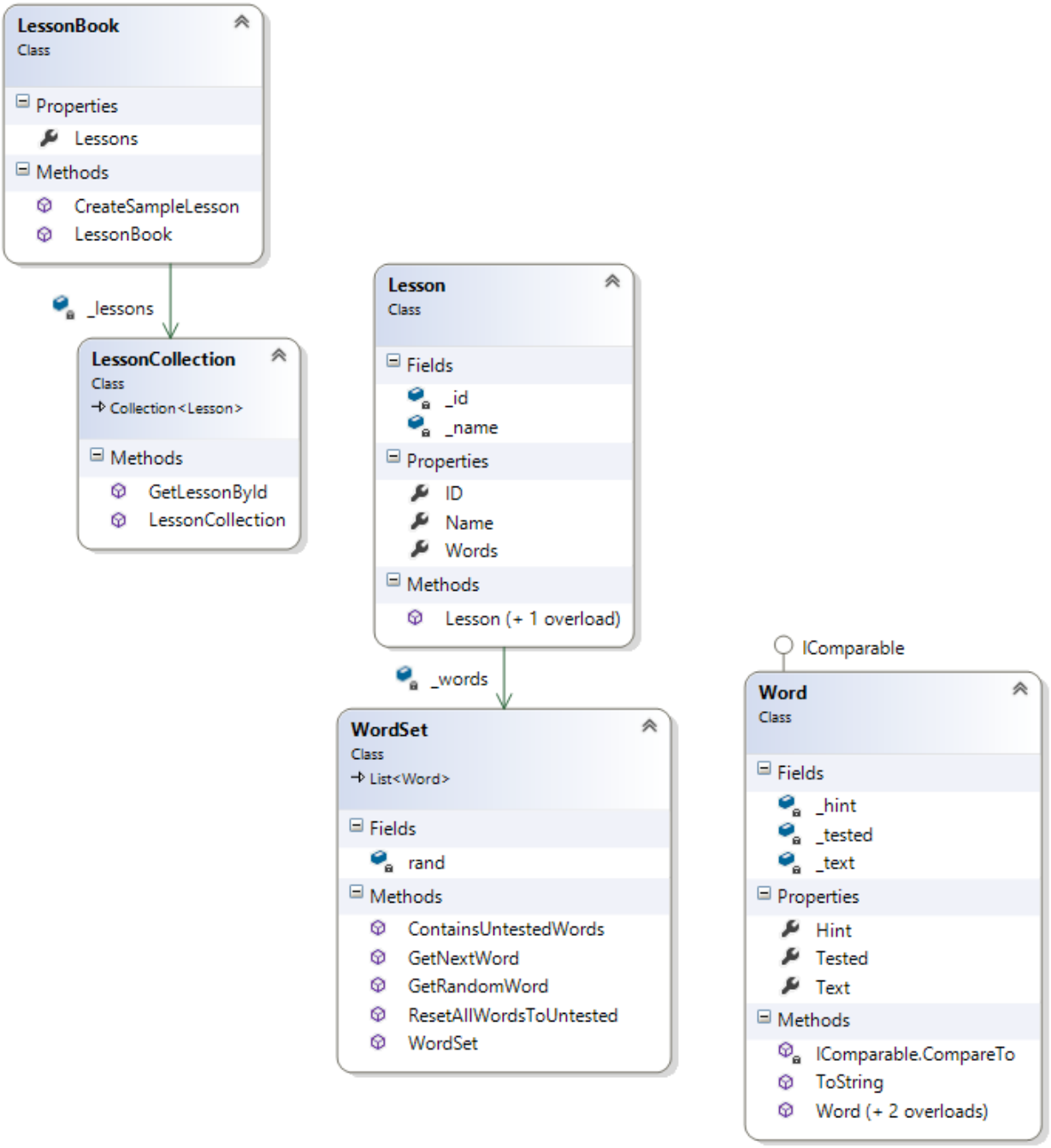
In general, persistent data is stored in a save file. It is loaded into the PlayerData object, which acts as an interface to the file. Upon starting a new game, relevant data present in the PlayerData object will get copied over into the game's persistent Context object. The Context is created when a new game is started and is destroyed when the player has to return to the Main Menu. The EnemyGenerator takes words from the user-selected lesson stored in Context to generate a level's enemies and boss. The Context is also used to keep track of the game's current state, including such things as the current score, how many and which letters have been collected, how many lives the player has, etc. Outside of gameplay, some menus interface directly with the PlayerData object load and save data such as the user's lessons, high scores, and preferences.



7.3 Data Constructs

7.3.1 Lessons

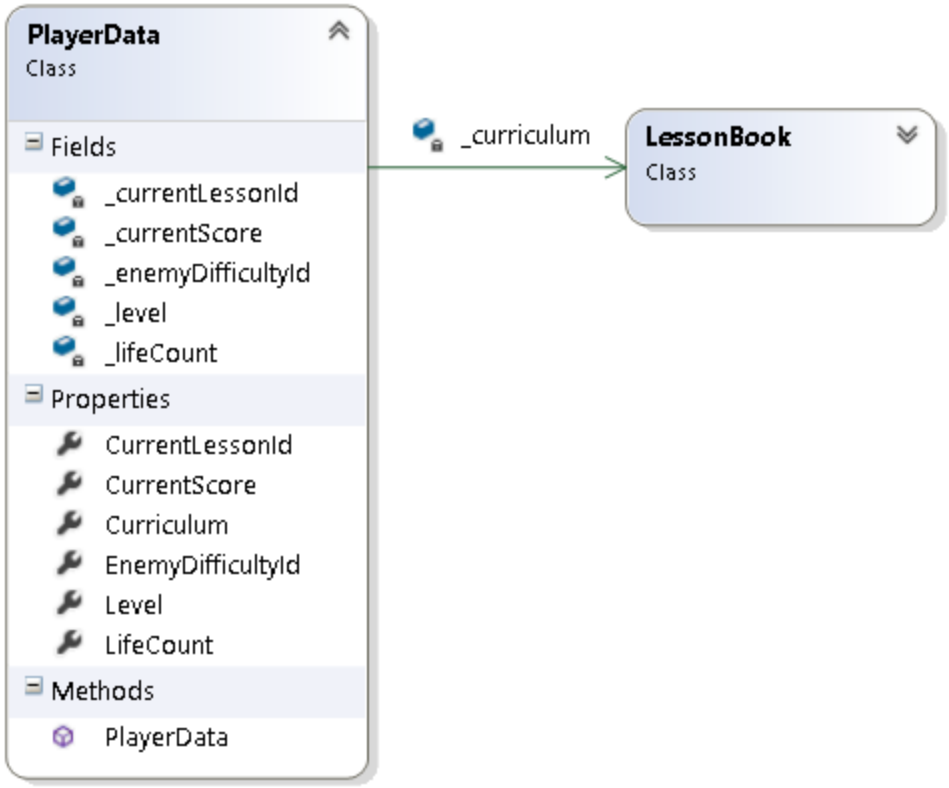
Users can create sets of words to be tested on. These words and lessons are stored and accessed using the following class constructs:



7.3.2 Player Data

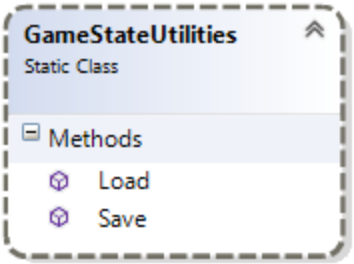
Lessons, high scores, and the game state need to be grouped and stored in a file to save across play session. The PlayerData class is used to group this data for saving and loading.

Please note: The diagram below is based off progress during iteration 2.



7.3.2.1 Game State Utilities

The static GameStateUtilities class is used to load the PlayerData from file and/or save it back.

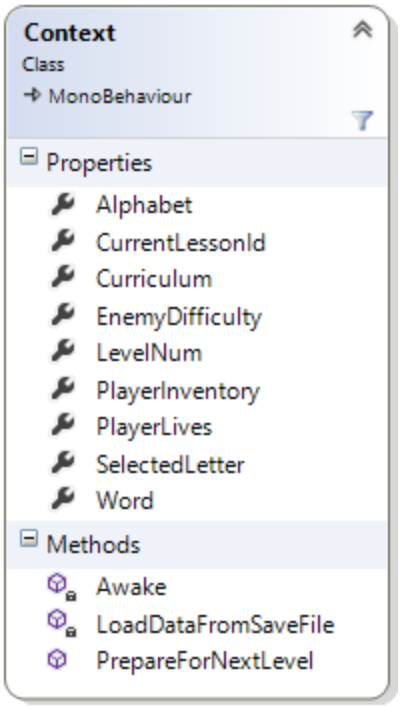


7.3.3 Context

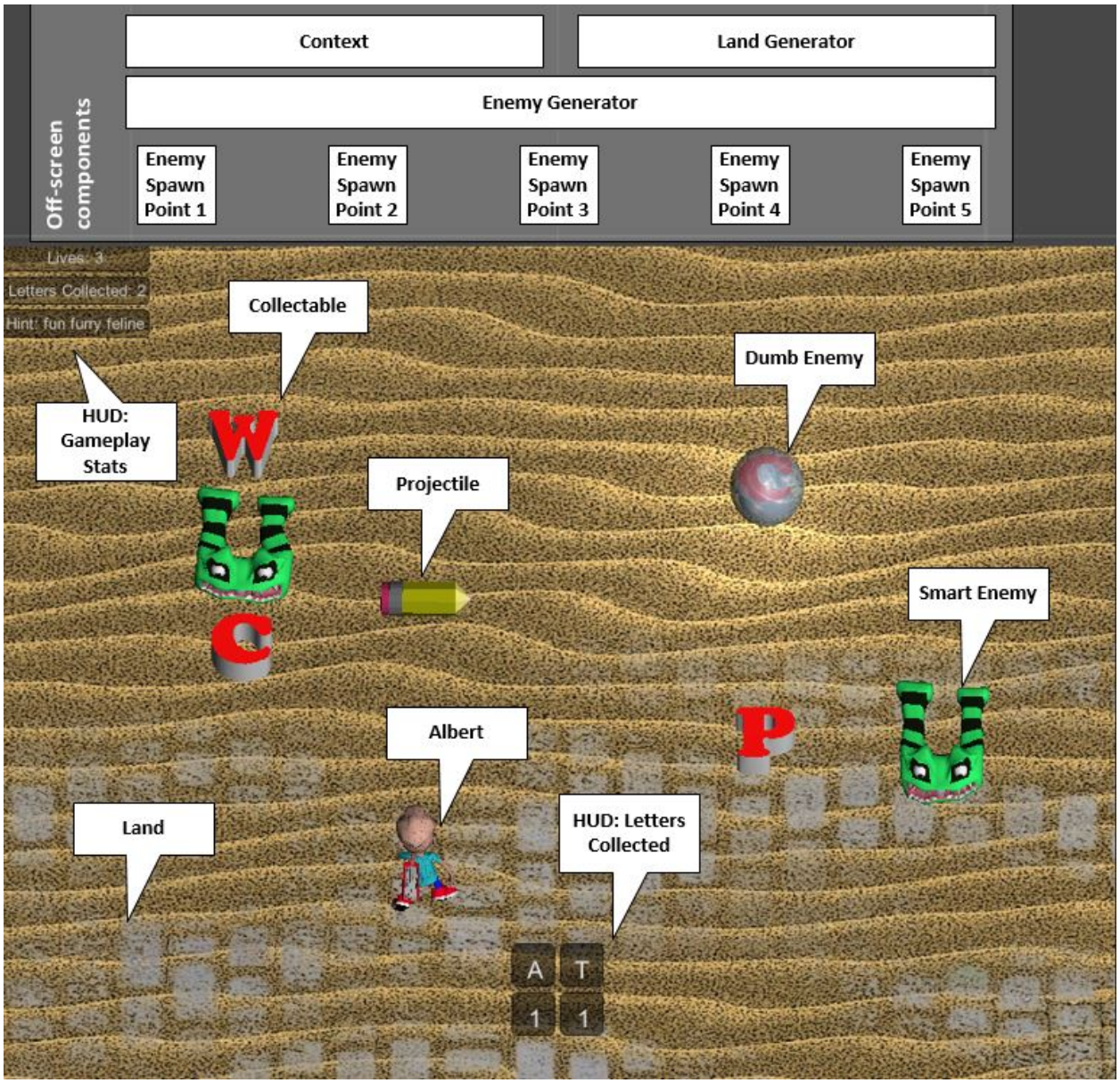
The game Context is used to keep track of the game state during gameplay. It is also used to store data that needs to be accessed by multiple independent entities, such as the HUD and the EnemyGenerator. The Context is created when the user loads his first level and is destroyed when the user returns to the Main Menu. Some if its properties, such as Curriculum, as always populated from the PlayerData object. Its other properties, like the EnemyDifficulty or Lives, may be populated either dynamically based on user actions or from

the PlayerData object when loading a saved game. If the user chooses to save his game, select properties of the Context will be copied back to the PlayerData object before it is saved to file.

Please Note:The diagram below is based off progress during iteration 1. It does not represent the final structure of the class, as additional functionality has yet to be added and/or refactored.



7.4 Game Structure and Objects



While many programs are designed around component interaction and layering, game design within Unity is built around the concept of object interaction within 3D space. When creating a new game, the developer is presented with an empty area of 3D space into which he can add game objects. A game object serves two primary purposes: as a means to add graphical models to the screen, and as a way to add scripts to the game. Each game object will remain fairly nonfunctional until it has a script attached to it or another object's script acts upon it. The developer does not write any core classes to control the program, but instead a series of smaller scripts detailing how each object with interact with its environment. The exception to

this design model are game objects used to intelligently generate other game objects, such as the Enemy Generator.

In the scene above, everything labeled except the HUD is a game object, with the HUD being a GUI object. In contrast to game objects, GUI objects are created entirely by script attached to another game object (usually the camera). The objects used within Letter Storm can be organized as follows:

- Game Objects
    - Visible
        - Primary (Active): Main Character (Albert), Enemies (Active, Passive, Boss)
        - Secondary (Passive): Projectiles, Collectibles, Land
    - Invisible (Empty Game Objects)
        - Data: Context
        - Managers: Enemy Generator, Land Generator
        - Reference Points: Enemy Spawn Points
        - View: Camera
- GUI Objects
    - HUD

Each of the game objects has access to the Context and can share data using it. For example, if the Main Character is hit by an Enemy, the Main Character will update the Lives count in the Context, which the HUD-generation script will then retrieve and display.

The game objects themselves do not have strict separation between View and Controller logic in the traditional sense, though. With the exception of GUI objects, there is very little View code used, since what is visible is actually 3D models generated using Blender. The Controller code is the script attached to these models that instructs them on what actions to take based on the Unity event system.

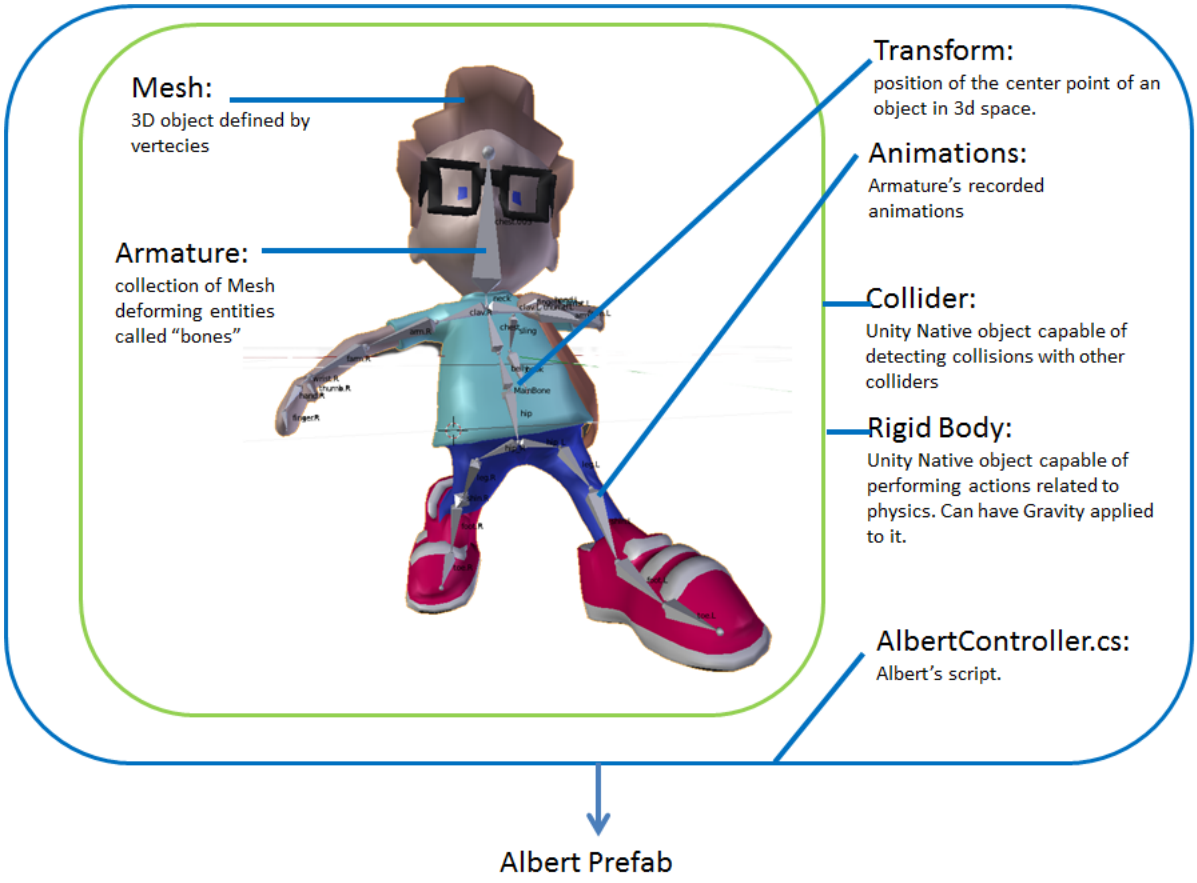
7.4.1 Active Visible Objects

Active visible objects are game objects that have a mesh associated with them, are on-screen, and are either controlled by the player or the game's AI. These include the Main Character (Albert), the active and passive Enemies, and the Boss. They are considered the primary game objects, as all game play is centered on them.

7.4.1.1 Main Character

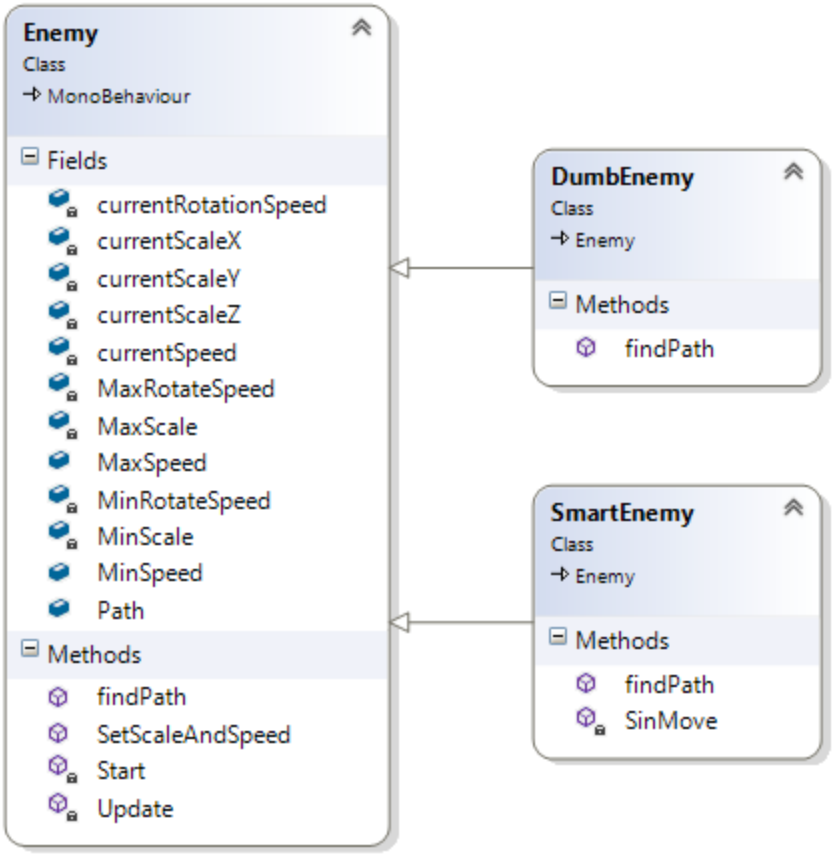
The character was modeled, textured and animated in Blender. It was then exported to Unity as an FBX file. (FBX files contain mesh information, textures, Armature and animations) .
A raw fbx model is quite useless in a game environment, unless components are added to it. In unity, each game object can have a variety of components, albert's components are:

1) Transform: the default component. A transform is nothing but a 3 dimensional coordinate that gives the object an actual location in the 3d world space of the game.
2) Animation Component: animations are a set of recorded bone positions through time. to be exact, the animation component is really attached to the armature of the model (not the mesh itself)
3) Collider: This component allows the object to which it is attached to register collision with other colliders. They come in various primitive shapes: Cube, Sphere, Capsule. Colliders can also be designed to wrap around a more complex shape, but this can be detrimental to performance. The best practice is to attach multiple primitive components to a complex mesh, rather than creating a complex mesh collider.
When colliders collide, the event OnCollisionEnter() is called.
4) Armature: Armatures are not unity components, but they are nonetheless components of the 3d mesh object. an armature deforms a mesh, and is responsible for storing various animations for the mesh.
5) Script: the component that controls all the other components as well as the behavior of the object it is attached to. Scripts can be written c# or Javascript.



7.4.1.2 Enemies

The two types of enemies that a player encounters in-game, consonant shells and vowel monsters, are prefabs that either have the DumbEnemy class script or SmartEnemy class script attached as a component respectively. These classes each inherit from a base Enemy class.



The base class provides a virtual method which each of the children overrides to determine which movement algorithm they will use. Currently, their movements are mostly the same, as the back-and-forth and sinusoidal movements that smart enemies will be capable of are currently in development.

7.4.1.3 Boss

Boss components:
The boss is constructed in the same manner as the main character at the core: it is a 3d mesh made in blender, with an armature that holds a few animations. The difference is tha the boss is composed of multiple objects that are all wrapped as a prefab.

A prefab is a unity default type. It acts a sort of wrapper that helps combine multiple gameobjects and components into one entity.
The boss is composed of 3 game objects, and each one has a separate script component.

1- The Boss's body:

A mesh rigged with an armature, and equiped with a texture componant. The boss's body has 2 script components:

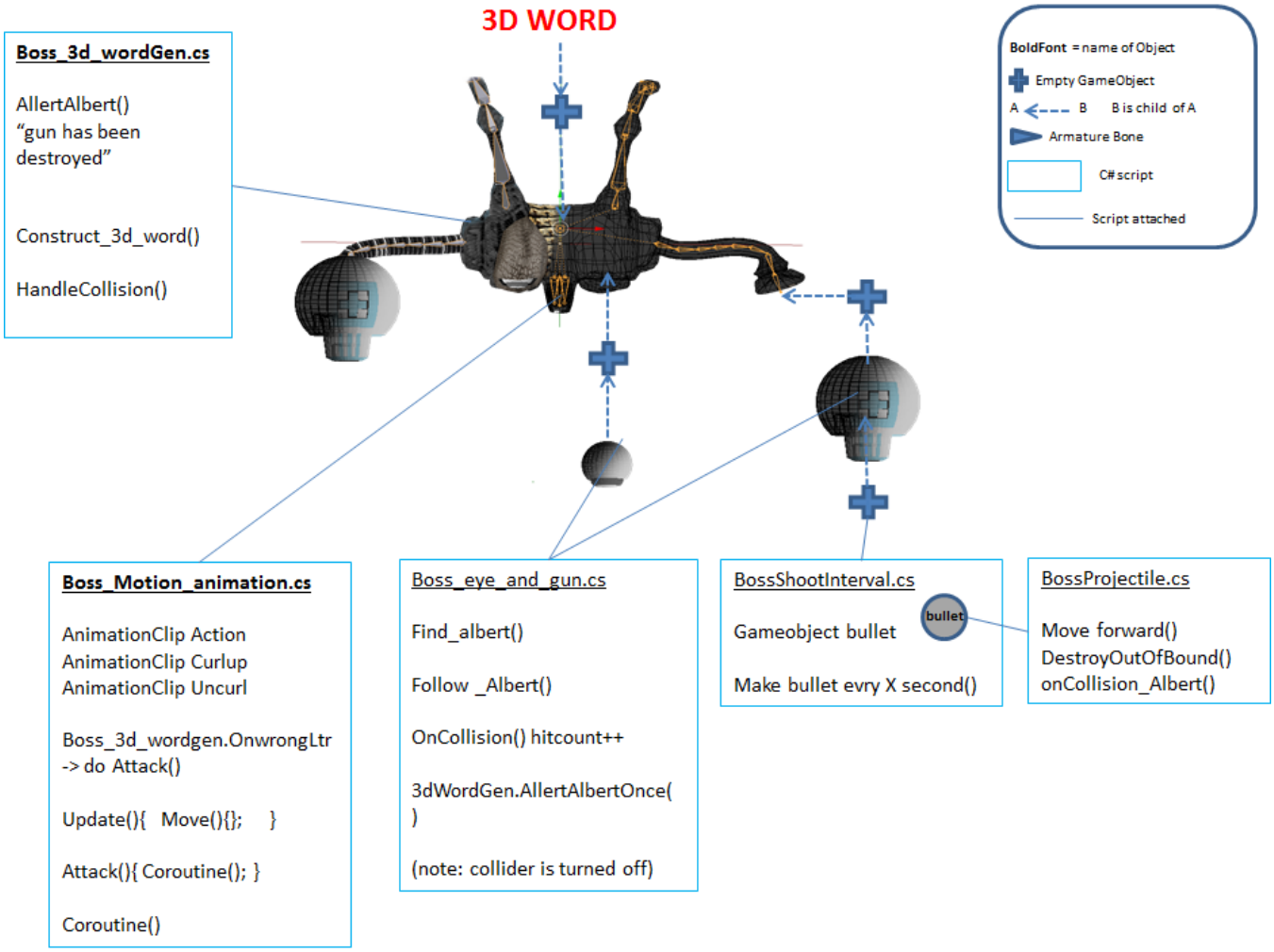
a) Boss_Motion_animation.cs: controls the movement of the boss from side to side, as well as the attack coroutine which makes the boss rush toward the player when a wrong letter was used to spell the word
b) Boss_3d_wodGenerator.cs: is responsible for loading the word to be solved from the context. And It is also responsible for rendering the solved letters on top of the boss.

2-The Boss's eyes:

The eyes are just a 3d Mesh with a texture component and a script component (no armature needed here since the eyes will not move according to a saved animation, but instead they will aim at the main player and follow him around dynamically at runtime)

3-The Boss's cannon:

The cannon is composed of a mesh and has an empty game object parented to it. This mean that the empty game object will always keep the same relative distance from the cannon. The empty object is positioned at the end of the cannon's barrel. This empty game object will serve as a spawn point for cannon bullets. As a bullet is instantiated, it travels away from the cannon, giving the impression that it was actually stored in the cannon and fired from there.



7.4.2 Passive Visible Objects

Passive visible objects are game objects that have a mesh associated with them, are on-screen, have no significant AI, and are generated by another game object, typically a primary object or a generator. These include Projectiles, Collectables, and the Land. They are considered the secondary game objects, as they typically facilitate interaction between the primary objects and each other or the environment.

7.4.2.1 Projectiles

Projectile game objects are created by Albert's script and given a velocity and trajectory that moves them in whichever direction the Main Character is facing whenever the fire key is pressed. They have a collider component attached to them that checks the identity (via an object's tag property) of whatever it is they collided with and handles the event appropriately. In the event of collision with an enemy, the enemy is destroyed, a collectible is spawned, and

a message is broadcast to the EnemyGenerator. In the event of collision with one of the boss' appendages, a local variable "hit_count" is incremented. When the total hit_count has reached a desired amount, the appendage is destroyed. Once all appendages are destroyed, the boss changes state to "word Solving" state, and a message is broadcast to the Albert's script to inform him of the Boss's changed state. At this point, albert can only shoot Letter Projectiles to solve the secret word.

7.4.2.2 Collectibles

Collectibles spawn in the event of an enemy's death. Each enemy represents a letter and naturally, that letter is the one that spawns. Collectibles move towards the player at the same slow pace as the scrolling land to give the illusion of a stationary object that the player is running towards. They have a collider component that detects whether or not the player has made contact and if so, the object is destroyed and the player's inventory is updated to reflect the collected letter.

7.4.2.3 Land

Land is a large tile game object on which all other visible game objects reside. It moves at a slow velocity toward the bottom of the viewable area to simulate the visual of the main character running forward. Since all gravity-based functionality is disabled for this game, Land serves little more purpose than to provide a convincing backdrop.

The Land Generator is responsible for creating new blocks of land to scroll into view as the current one scrolls out. Please see section 7.4.4.2 for more information on intelligent land generation.

7.4.3 Off-screen Data Objects

The primary, and only, data object that exists during gameplay is the Context. It is an empty Game Object located off-screen with a custom script attached. The script primarily contains publicly accessible properties that other Game Objects can access. This is used to facilitate the sharing of common data across game objects. For more information on the Context data structure, please see section 7.3.3.

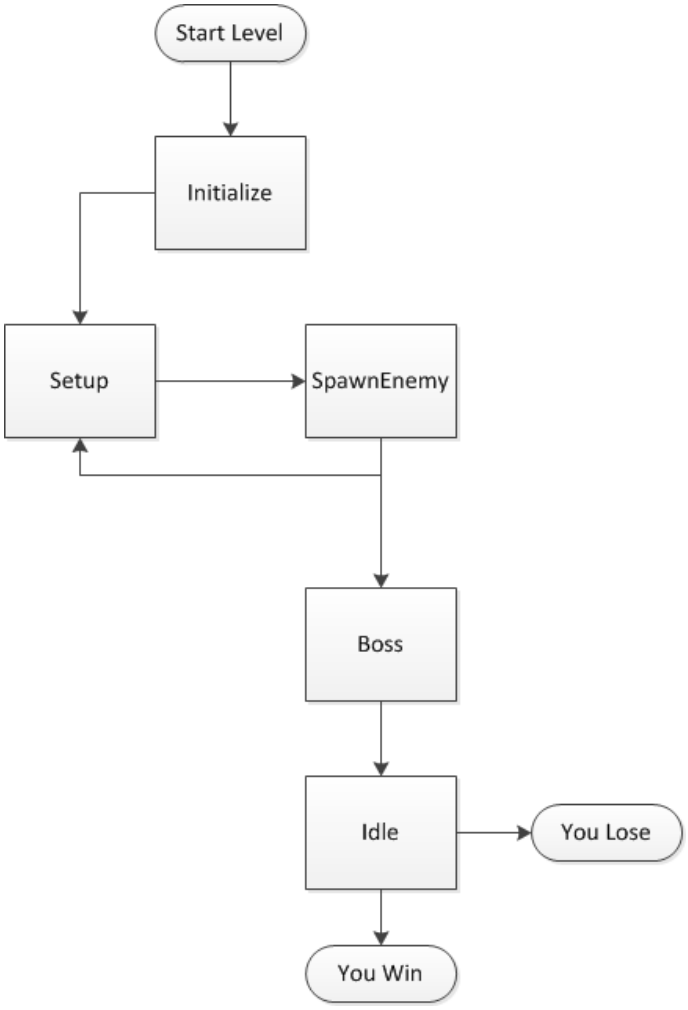
7.4.4 Off-screen Management Objects

Off-screen management objects are game objects that do not have a mesh associated with them (making them not visible on-screen), and are responsible for monitoring the game and

generating primary and secondary game objects intelligently. These include the Enemy Generator and Land Generator.

7.4.4.1 Enemy Generator

The EnemyGenerator is a finite state machine that maintains information regarding everything that is on-screen at any given time, as well as what a player has in his/her inventory



When a level begins, the EnemyGenerator calls the Context to determine what word the player will be tested on and then enters the Initialize state. In this state, the Generator makes sure there are spawn points and enemy prefabs ready to use and prepares certain data structures for the purposes of tracking the player's progress in collecting letters required to spell the pre-determined word.

After initialization, the FSM enters the Setup state, where the EnemyGenerator checks to see if the player has collected all required letters. If the player has, the FSM enters the Boss state, if the player has not, the FSM enters the SpawnEnemy state. If the boss has already spawned, the FSM enters the Idle state.

In SpawnEnemy, EnemyGenerator checks to see how many spawn points are currently available. An available spawn point is any spawn point that does not have a living child on-screen. The Generator then chooses a random number of enemies to spawn, not to exceed the number of available spawn points. There is a two in three chance that the first enemy spawned during this call will be a required letter. All other letters will be chosen at random.

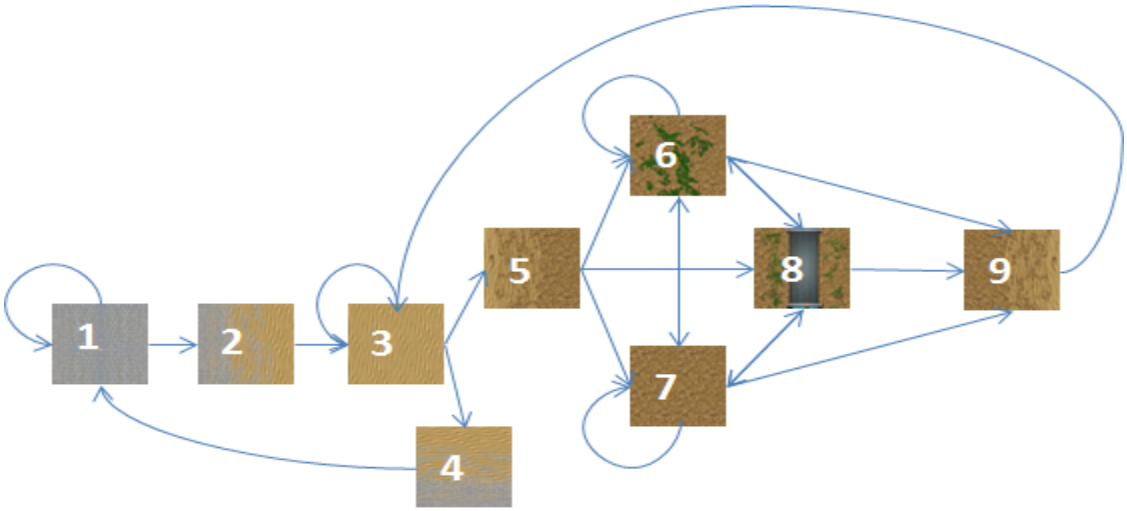
In the Boss state, the boss prefab is instantiated and a flag is flipped indicating that the boss has spawned.

Once the EnemyGenerator has reached the Idle state, its work has finished until the next level begins. The mechanics of the boss fight are all handled by scripts attached to the boss prefab.

7.4.4.2 Land Generator

Land Generator and Land tiles, and and the random state machine:
There are 2 entities that are responsible for the background:
Land Tile:
-a single object with a land_script and a texture attached to it.
Land Generator:
-an empty Game object with the Land_generator script attached to it

The Land generator keeps track of what state it is in , and depending on the current state, it will select the next possible state at random.
Tiles are programmed to move down the screen upon instantiation.
When a tile has moved a total distance equal to its length, the tile prompts the LandGenrator to instantiate a new tile depending on the current state.
Here is all the combinations of states, and the transition between states:
For example: if the land generator has instantiated tile #3 , the current state s markes as 3, and the next possible tiles to generate can be #3 again, #4, or #5.



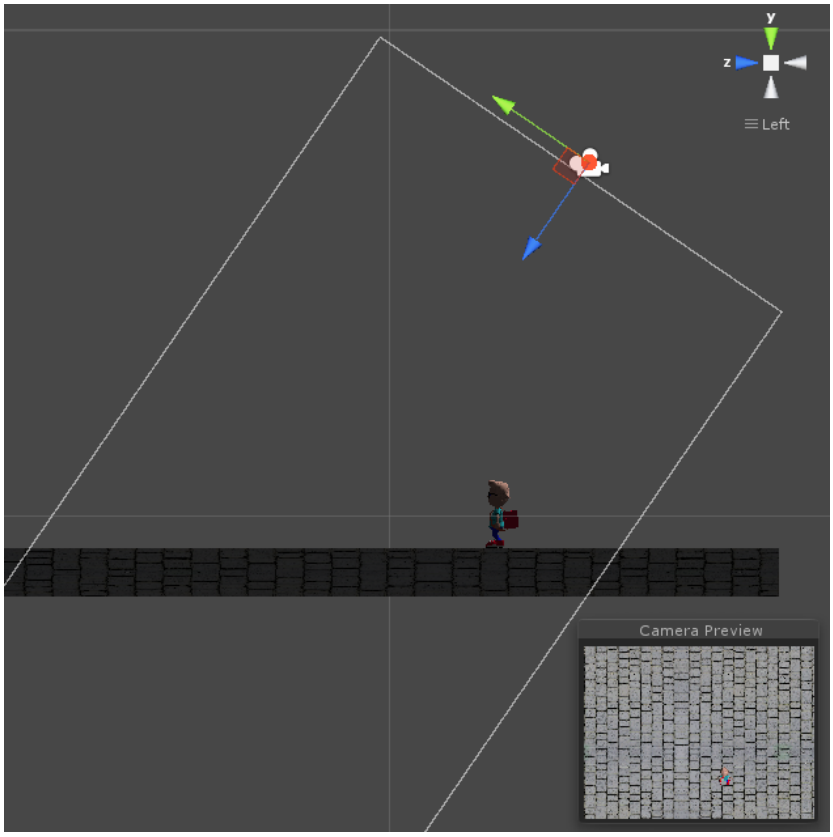
7.4.5 Camera and GUI Objects

GUI objects are not game objects; they are created on the lense of the camera (ie the screen) entirely via script using the GUI and GUILayout APIs provided by Unity. Because of this, their contents can change rapidly, making them ideal for displaying information that changes throughout gameplay. The HUD and menu system are built primarily using GUI objects.

7.4.5.1 Camera

The camera game object's primary purpose is to instruct the game engine on what portion of the 3D environment should be visible to the player and how. While many games allow the player to control the camera, due to the nature of this game it is fixed at a 55.5 degree angle facing downward. An orthographic, vs perspective, view is used to allow for objects that are farther away from the camera to be the same size as those that are close to it, an attribute common to most 2D shooters.

While most game objects have scripts associated with them, the in-game camera does not. This is not because the object cannot host scripts, but instead because a separate empty game object was created to host the HUD script in an effort to promote functional encapsulation. When viewing menus outside of game play, though, their GUI script is typically hosted on the camera object.



7.4.4.2 HUD

The Heads-Up-Display, or HUD, is a GUI interfaces on which information is displayed to the player. It is a standard form of showing statistics relating to gameplay. In Unity, the HUD is composed of a series of GUI objects generated by a script attached to an empty Game Object. The HUD has three main areas: general game info, such as the gameplay statistics (player health or lives) and the hint of the spelling word; the letters-collected button set (namely the inventory); and the pause screen.

The HUD uses the built-in Unity “Monobehaviour” function OnGUI() to generate the HUD controls. The function may get called several times per frame, depending on how many events happen. Events include but are not limited to user input (such as a key press), and processing events. A sample of the OnGUI() code is shown below. DisplayPauseMenu() and DisplayInventoryWindow() are functions within the HUD script that define the pause menu and inventory of collected letters, respectively. This function is called at least two times per frame: once to lay out the GUI, another at least to “repaint” it.

```
/// <summary>
/// Method that updates HUD once every frame
/// Displays game, player information, and player inventory
/// </summary>
0 references | Jeannie Trinh +3, 9 days ago | 22 changes
void OnGUI()
{
    DisplayHints();                     // Draw hints text
    DisplayHPBar();                     // Draw HP Bar
    DisplayScoreBar();                  // Draw Player's Score
    DisplayPauseMenu();                 // Draw Pause Menu when in pause mode
    DisplayInventoryWindow();           // Draw Inventory
}
```

(Above) OnGUI() method that draws all the components of the in-game HUD, separated into organized functions.

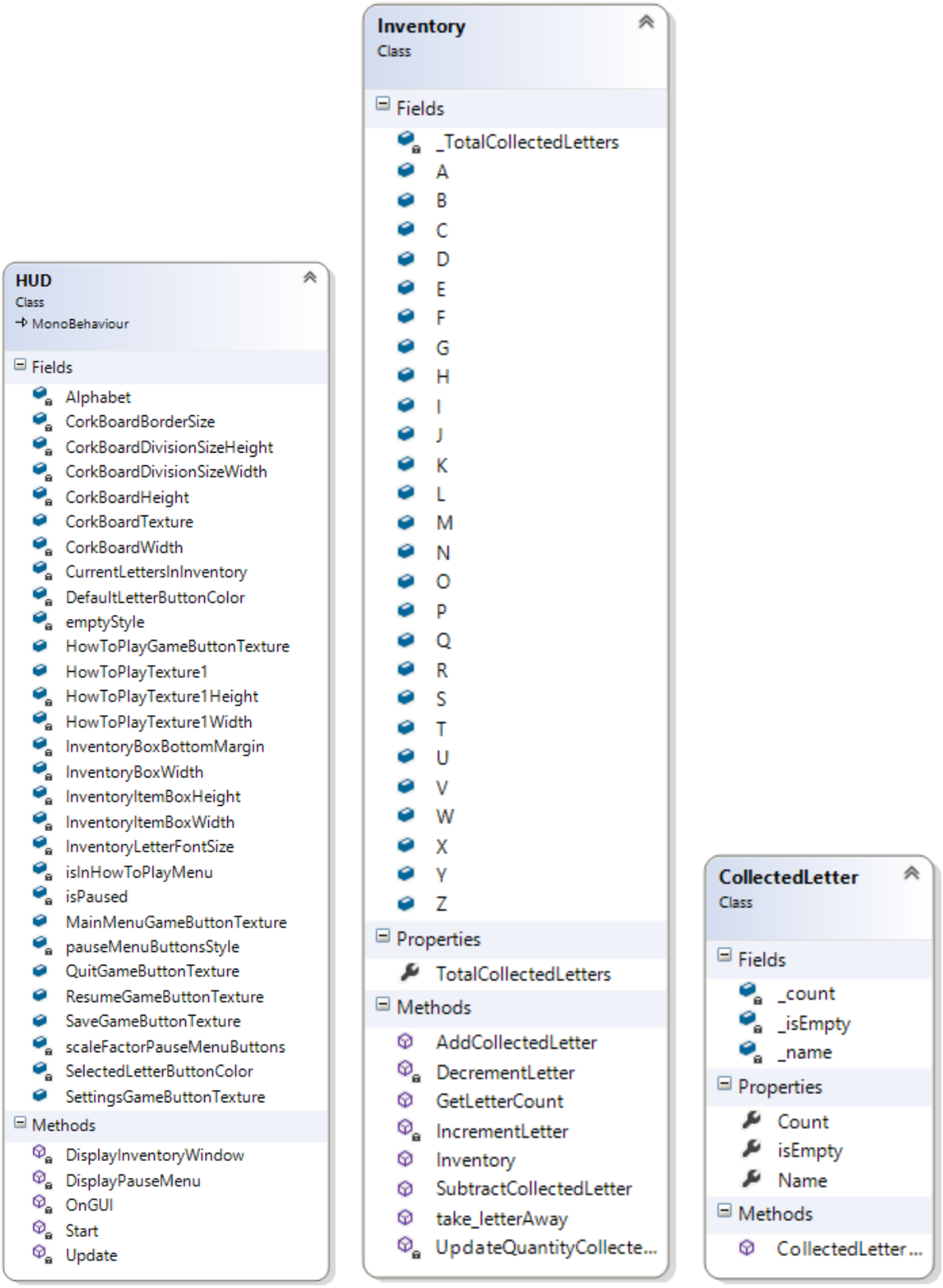
```
/// <summary>
/// Displays HP bar in HUD
/// </summary>
1 reference | Jeannie Trinh, 9 days ago | 1 change
public void DisplayHPBar()
{
    GUI.TextField(new Rect(HPBarX, HPBarY, HPBarWidth, HPBarHeight), "");          // Bar's background
    HPBarStyle.alignment = TextAnchor.MiddleCenter;
    HPBarStyle.normal.textColor = Color.black;
    GUI.TextField(new Rect(HPBarX, HPBarY, HealthBarLength, HPBarHeight),
        "HP: " + CurrentHealth.ToString() + "/" + MaximumHealth.ToString(), HPBarStyle);
}
```

(Above) Example of drawing some information, such as the HP bar, onto the HUD.

Firstly, there are two classes that represent the inventory and the collected letters present in the game. They are the Inventory class and the CollectedLetter class.

In the Context object, a new Inventory object is created upon the game loading. An instance of an Inventory class contains 26 CollectedLetter objects, each corresponding to a letter in the alphabet, and the number of how many letters are collected in total. Updates to the Context class's static variables such as Player Lives or Health, Hints, and letters collected and placed in the Inventory will be accessed and displayed in the HUD class. An example would be updating the count (incrementing, decrementing) of the collected letters in the player's inventory upon events such as the player picking up a letter, or shooting away a letter they have collected. See the class diagrams below for specific layout of the classes.

Each CollectedLetter instance contains the name of the letter (“A”, “B”, “C”, etc.) and the integer count variable to keep track of how many letters are collected during gameplay. A boolean isEmpty is also there to check if we have none of a letter or not, which is useful in determining if a letter should be drawn on the screen or not (i.e. do not draw an “A” to the screen if the *count* of it is 0 (0 means the user has not collected an instance of that “A”).


HUD
Class
MonoBehaviour
Fields
Alphabet
CorkBoardBorderSize
CorkBoardDivisionSizeHeight
CorkBoardDivisionSizeWidth
CorkBoardHeight
CorkBoardTexture
CorkBoardWidth
CurrentLettersInInventory
DefaultLetterButtonColor
emptyStyle
HowToPlayGameButtonTexture
HowToPlayTexture1
HowToPlayTexture1Height
HowToPlayTexture1Width
InventoryBoxBottomMargin
InventoryBoxWidth
InventoryItemBoxHeight
InventoryItemBoxWidth
InventoryLetterFontSize
isInHowToPlayMenu
isPaused
MainMenuGameButtonTexture
pauseMenuButtonsStyle
QuitGameButtonTexture
ResumeGameButtonTexture
SaveGameButtonTexture
scaleFactorPauseMenuButtons
SelectedLetterButtonColor
SettingsGameButtonTexture
Methods
DisplayInventoryWindow
DisplayPauseMenu
OnGUI
Start
Update
Inventory
Class
Fields
_TotalCollectedLetters
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
Properties
TotalCollectedLetters
Methods
AddCollectedLetter
DecrementLetter
GetLetterCount
IncrementLetter
Inventory
SubtractCollectedLetter
take_letterAway
UpdateQuantityCollecte...
CollectedLetter
Class
Fields
_count
_isEmpty
_name
Properties
Count
isEmpty
Name
Methods
CollectedLetter...

Continuing with the OnGUI function, the below screenshot shows part of the DisplayInventoryWindow() method.

```
// Inventory dimensions
GUILayout.BeginArea(new Rect(
    InventoryX,                                                    // X start position
    InventoryY,                                                    // Y start position
    InventoryWidth,                                                // Width
    InventoryHeight));                                             // Height

#region Letter in Inventory ------------------------------------------
// Draw collected letters in Inventory

GUILayout.BeginHorizontal();
GUILayout.FlexibleSpace();
GUI.backgroundColor = Color.black;

GUI.color = DefaultLetterButtonColor;

for (int ii = 0; ii < CurrentLettersInInventory.Count; ii++)
{
    string currentLetter = CurrentLettersInInventory[ii].ToString();
    GUI.color = Context.SelectedLetter == currentLetter ? SelectedLetterButtonColor : DefaultLetterButtonColor;
    if (GUILayout.Button(
        "<size=" + InventoryLetterFontSize + ">" + currentLetter + "</size>",
        GUILayout.Height(InventoryItemBoxHeight), GUILayout.Width(InventoryItemBoxWidth)))
    {
        Context.SelectedLetter = currentLetter;
        GUI.color = DefaultLetterButtonColor;
    }
}

GUILayout.FlexibleSpace();
GUILayout.EndHorizontal();
#endregion Letter Type in Inventory ----------------------------------------

Letters' Count in Inventory ----------------------------------------

GUILayout.EndArea();
```

(Above) A section of the DisplayInventoryWindow() method. Part of the letters drawn in the inventory in the HUD. Demonstrates the logic for deciding which letter(s) should be drawn (i.e. checks for what letters the user has collected or not and draws only the ones that the user has obtained).

The letters in the inventory are drawn onto the screen using the GUILayout class being called by the OnGUI() method. The GUILayout class is the interface for Unity GUI with automatic layout. In this example, we used the BeginArea() method. This initiates a rectangular regions onto the screen, defined by the *float* input of the (x, y) of the top-left corner of the rectangle, followed by width and height. For LetterStorm, the inventory will be placed at the bottom, center of the screen. Then, we define a BeginHorizontal() area. Doing so creates a “row” within the area of the rectangular “layout”. Therefore, by simply calling a “Box” or a “Button” item in Unity sequentially within the horizontal row region (the part of the code between BeginHorizontal() and EndHorizontal(), the GUILayout class takes care of positioning the boxes or buttons next to each other, going from left to right of the “Horizontal” region. A specific size (width x height), and the starting position (x, y), can be given to limit the size of

the GUILayout, and to each individual box. The sizing, position, color, background color, and other properties of the GUI controls can be extended to TextFields, Labels, Buttons, etc.

First, we iterate through the player's inventory (in Context) and keep track of only the letters that are collected in an ArrayList, ignoring all letters we collect none of. This is to simplify the HUD so only the collected letters are shown, avoiding rendering the entire alphabet.

Another method in the HUD class to mention is the Update() function. This method updates once per frame and it is in this code that the numerical values and flags are computed. For example, adjustment of objects that are part of the HUD (like Score Bar) are adjusted depending on the screen resolution (i.e. wider screen means score bar's length drawn must be adjusted in order to not stretch too long).

```
void Update()
{
    // Always keep track of player health
    UpdatePlayerStats();

    // Determine scrolling hint text dimensions
    AdjustScrollingHintDimensions();

    // Determine size of inventory "boxes" depending on screen size
    AdjustInventoryDimensions();

    // Dimensions - HP bar
    AdjustHPBarDimensions();

    // Dimensions - CorkBoard for pause menu
    AdjustCorkboardDimensions();

    // Dimensions - How To Play Menu
    HowToPlayTexture1Width = HowToPlayTexture1.width * scaleFactorPauseMenuButtons;
    HowToPlayTexture1Height = HowToPlayTexture1.height * scaleFactorPauseMenuButtons;

    // Pause/Unpause game flow control
    PauseOrUnpauseGame();

    // Determine which letter is selected based on key presses
    SetSelectedLetterFromKeyPress();

    // If [Esc] is pressed, pause the game
    if (Input.GetKeyDown(KeyCode.Escape))
    {
        // If paused already, unpause
        if (isPaused)
        {
            isPaused = false;
            isInHowToPlayMenu = false;
        }
        // If not paused, pause game
        else
        {
            isPaused = true;
        }
    }
```

7.4.4.2 Pause Menu

As part of the HUD, a Pause screen is implemented. We assume we are in-game, in which the script for the HUD is running. In LetterStorm, when the user presses the Escape button, the Pause menu opens up. The detection of the Escape button is also done in HUD.cs, as part of the Update() method, shown below.

```
// If [Esc] is pressed, pause the game
if (Input.GetKeyDown(KeyCode.Escape))
{
    // If paused already, unpause
    if (isPaused)
    {
        isPaused = false;
        isInHowToPlayMenu = false;
    }
    // If not paused, pause game
    else
    {
        isPaused = true;
    }
}
```

isPaused is a boolean flag representing if the game is currently paused or not. The code calls upon user-input detection properties to detect if the Escape button is pressed. And if it is, tests if the game is currently paused or not. If the game is currently paused, then the code will flip the flag to “unpaused,” and vice versa.

```
/// <summary>
/// Pause or Unpause the game
/// </summary>
1 reference | jtrinhx33, 21 days ago | 1 change
private void PauseOrUnpauseGame()
{
    if (isPaused)
    {
        AudioListener.pause = true;
        Time.timeScale = 0;
    }
    else
    {
        PauseOrUnpauseBGM();
        Time.timeScale = 1;
    }
}
```

The above image shows the PauseOrUnpauseGame() method called in the Update() method (the method that is called once per frame). Depending on the flag isPaused, the game will pause or unpause itself. Time.timeScale is a property that controls the speed of the game, or rather, any code that uses the Time.deltaTime property. For example, movement or animation of an enemy object is dependent on the Time.deltaTime. To properly produce movement such as a translation from one spot to another on the screen, one could multiply the position vectors by Time.deltaTime. This makes sure that movement is a consistent speed across any platform (possibly due to clock cycles differing on different machines). As such, Time.deltaTime is multiplied by the Time.timeScale property. If Time.timeScale is set to 1,

then all objects using Time.deltaTime will move at the normal speed of 1. If it was set to 2, then speed of the objects will be doubled the normal speed. And if it was set to 0, then the objects stop completely, essentially causing the game to have the effect of “pausing”.

As shown above also, audio is paused when the flag is evaluated to true.

The drawing of the Pause menu itself is code done within the function DisplayPauseMenu(), called in the OnGUI() method as shown in one of the images above. The image below shows The specifics of drawing a Pause menu.

```
private void DisplayPauseMenu()
{

    // Draw pause menu
    if (isPaused)
    {
        GUI.skin.button.alignment = TextAnchor.MiddleCenter;

        // Draw pause menu background
        GUI.DrawTexture(new Rect(Screen.width / 2 - CorkBoardWidth / 2,
            Screen.height / 2 - CorkBoardHeight / 2, CorkBoardWidth, CorkBoardHeight), CorkBoardTexture, ScaleMode.ScaleToFit, true);

        if (!isInHowToPlayMenu)
        {
            // Draw pause menu buttons
            // Resume button
            if (GUI.Button(new Rect(Screen.width / 2 - CorkBoardWidth / 2 + CorkBoardBorderSize,
                Screen.height / 2 - CorkBoardHeight / 2 + CorkBoardBorderSize * 2,
                CorkBoardDivisionSizeWidth,
                CorkBoardDivisionSizeHeight), ResumeGameButtonTexture, emptyStyle))
            {
                isPaused = false;
                isInHowToPlayMenu = false;
            }
            // How to play button
            if (GUI.Button(new Rect(Screen.width / 2 - CorkBoardWidth / 2 + CorkBoardDivisionSizeWidth + CorkBoardBorderSize * 2,
                Screen.height / 2 - CorkBoardHeight / 2 + CorkBoardBorderSize * 2,
                CorkBoardDivisionSizeWidth,
                CorkBoardDivisionSizeHeight), HowToPlayGameButtonTexture, emptyStyle))
            {
                isInHowToPlayMenu = true;
            }
            // Quit game button
            if (GUI.Button(new Rect(Screen.width / 2 - CorkBoardWidth / 2 + CorkBoardDivisionSizeWidth * 2 + CorkBoardBorderSize * 3,
                Screen.height / 2 - CorkBoardHeight / 2 + CorkBoardBorderSize * 2,
                CorkBoardDivisionSizeWidth,
                CorkBoardDivisionSizeHeight), QuitGameButtonTexture, emptyStyle))
            {
```

The pause menu is only drawn when the isPaused flag is up. When the game is paused, then the “CorkBoardTexture” is drawn in the middle of the screen. The texture variable references the corkboard png image to produce the image on the screen. All objects are drawn sequentially as code is run (above). Buttons, each having a texture (such as the sticky-note or floppy-disk style shown in the first screenshot for the pause menu) are drawn in front of the corkboard texture. The if-statements evaluate to true when their respective buttons are pressed, at then which actions are performed. For example, the drawing of the Resume button code is placed within an if-statement. That statement evaluates to true when the

Resume button is pressed. As a result the flag isPaused is set to false (to unpause the game). Once the isPaused flag is set to false, the Pause menu will no longer be drawn the next time the OnGUI method is called.

7.5 Menus

Unlike levels, menus are typically composed primarily of GUI objects and thus must be built via code. The creation of a menu's GUI objects and the monitoring of events associated with them is so tightly coupled, though, that it is significantly difficult to separate view and controller code for them. Thus, all existing data-dependent, or smart, menus use a model-view architecture (vs MVC), with static menus existing fairly independently via a single-layer view architecture.

Smart menus include the New Game screen, the Lesson Management screen, the Settings screen, and the High Scores layout. All other menus are static. Smart menus have direct access to the PlayerData object, as there is no Context object loaded for them to utilize. Most smart menus (excluding the New Game screen and High Scores layout) can save their changes back to the PlayerData object so they can be loaded later.

7.5.1 New Game Screen

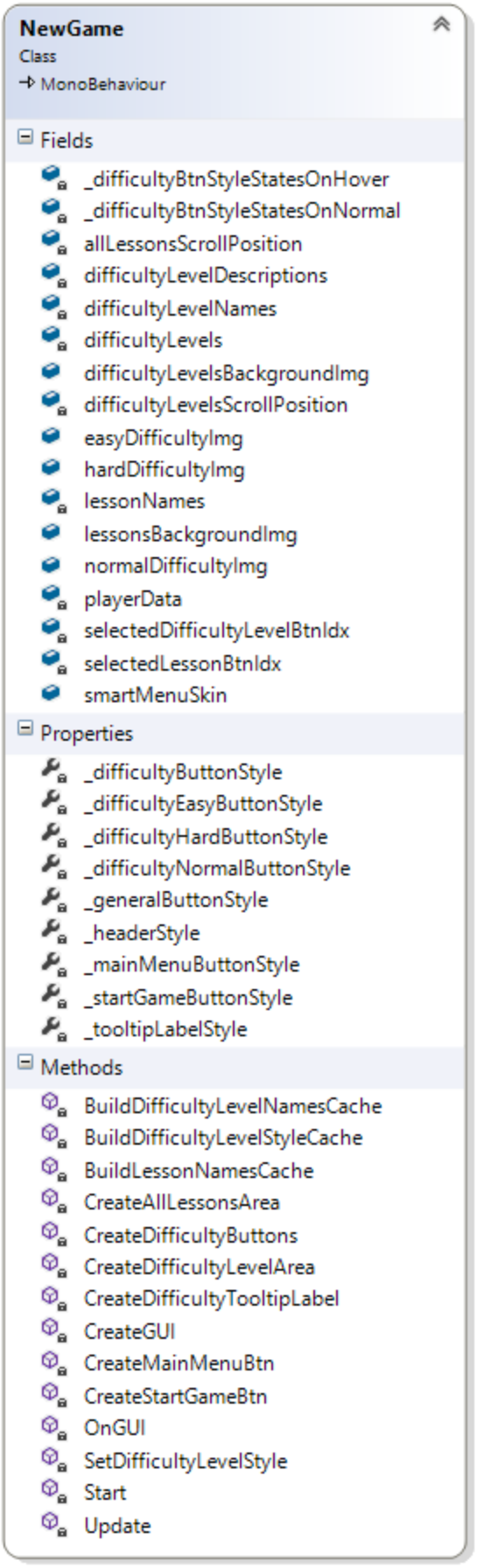
As stated in section 5.11, the New Game screen is used to provide the user with a means to select the desired game difficulty and the lesson/set of words to learn before starting the game. Excluding its background, which is generated using the Land Generator, it is built entirely dynamically by its NewGame.cs script using the GUI and GUILayout APIs provided by Unity. The user's selected difficulty level and lesson are stored in the Context by the script before it loads the first level, allowing it to pass these values to the game.

As noted previously, it has direct access to the PlayerData object in order to display all existing lessons. It also utilizes the DifficultyLevels data collection to build the list of existing difficulty levels and supporting information.

The code flow for NewGame.cs is shown below. The Start() and OnGUI() methods are actually Unity events that get executed when the scene first loads and on every subsequent frame, respectively. All common data used to built the screen, such as the Curriculum and Difficulty Levels, are initialized in the Start() method. The OnGUI() method is used to create all GUI elements on the screen for each frame, and to handle any events fired by these elements.

NewGame.cs – Code flow

```
void Start ()
> private void BuildDifficultyLevelStyleCache(bool resetStyles)
void OnGUI()
> private void CreateGUI()
--> private void CreateDifficultyLevelArea()
----> private void BuildDifficultyLevelNamesCache(bool refresh)
----> private void CreateDifficultyButtons()
------> private void SetDifficultyLevelStyle(int selectedButtonIdx)
----> private void CreateDifficultyTooltipLabel()
--> private void CreateAllLessonsArea()
----> private void BuildLessonNamesCache(bool refresh)
--> private void CreateStartGameBtn()
--> private void CreateMainMenuBtn()
```


NewGame
Class
MonoBehaviour
Fields
_difficultyBtnStyleStatesOnHover
_difficultyBtnStyleStatesOnNormal
allLessonsScrollPosition
difficultyLevelDescriptions
difficultyLevelNames
difficultyLevels
difficultyLevelsBackgroundImg
difficultyLevelsScrollPosition
easyDifficultyImg
hardDifficultyImg
lessonNames
lessonsBackgroundImg
normalDifficultyImg
playerData
selectedDifficultyLevelBtnIdx
selectedLessonBtnIdx
smartMenuSkin
Properties
_difficultyButtonStyle
_difficultyEasyButtonStyle
_difficultyHardButtonStyle
_difficultyNormalButtonStyle
_generalButtonStyle
_headerStyle
_mainMenuButtonStyle
_startGameButtonStyle
_tooltipLabelStyle
Methods
BuildDifficultyLevelNamesCache
BuildDifficultyLevelStyleCache
BuildLessonNamesCache
CreateAllLessonsArea
CreateDifficultyButtons
CreateDifficultyLevelArea
CreateDifficultyTooltipLabel
CreateGUI
CreateMainMenuBtn
CreateStartGameBtn
OnGUI
SetDifficultyLevelStyle
Start
Update

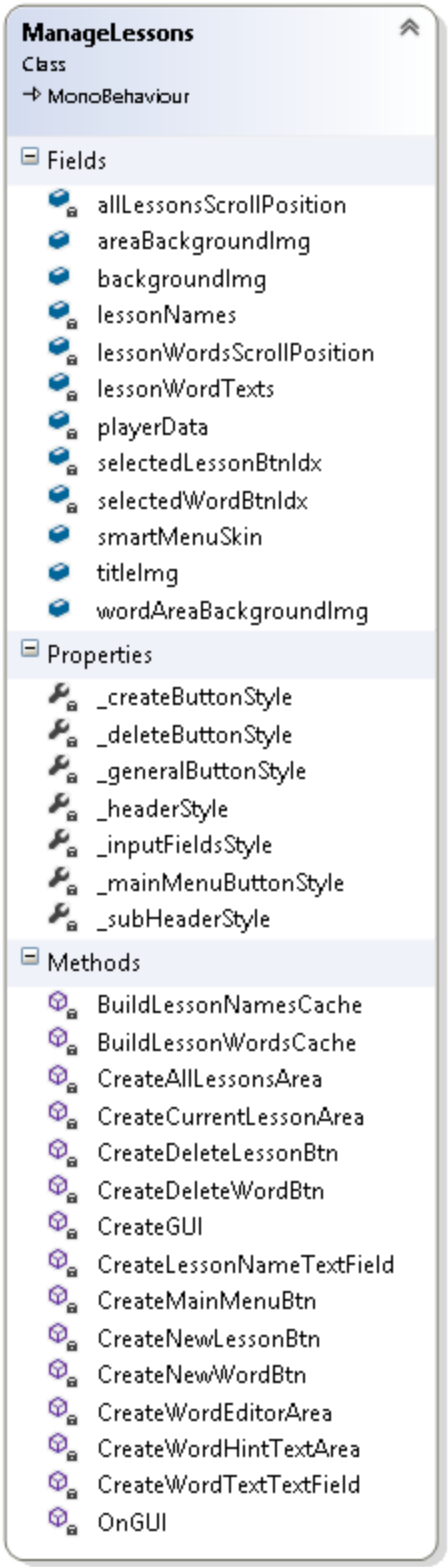
7.5.2 Manage Lessons Screen

The Lesson Management Screen displays to the user all of existing lessons stored in the save file, as well as their contents. The user is able to create, modify, and delete lessons and their words. The screen is built entirely using Unity's GUI and GUILayout APIs from the ManageLessons.cs script.

All data displayed within the Manage Lessons screen is first loaded from the save file into a PlayerData object by the GameStateUtilities. This is done when the scene first loads via the Start() method. During each subsequent frame, the ManageLessons.cs script then dynamically populates the page with interactive buttons, textboxes, and the like based on the contents of the PlayerData object and what the user clicks on. Any changes made to a lesson or word via the screen is immediately stored in the PlayerData object, and is saved back to the save file when returning to the Main Menu.

ManageLessons.cs – Code flow

```
void Start ()
void OnGUI()
> private void CreateGUI()
--> private void CreateAllLessonsArea()
----> private void BuildLessonNamesCache(bool refresh)
----> private void CreateNewLessonBtn(ref int selectedLessonIdx, ref int
oldSelectedLessonIdx)
------> private void BuildLessonNamesCache(bool refresh)
--> private void CreateCurrentLessonArea()
----> private void BuildLessonWordsCache(int lessonIdx, bool refresh)
----> private void CreateLessonNameTextField(int lessonIdx)
------> private void BuildLessonNamesCache(bool refresh)
----> private void CreateNewWordBtn(int lessonIdx, ref int selectedWordIdx)
------> private void BuildLessonWordsCache(int lessonIdx, bool refresh)
----> private void CreateDeleteLessonBtn(ref int lessonIdx)
------> private void BuildLessonNamesCache(bool refresh)
--> private void CreateWordEditorArea()
----> private void CreateWordTextTextField(int lessonIdx, Word wordToEdit)
------> private void BuildLessonWordsCache(int lessonIdx, bool refresh)
----> private void CreateWordHintTextArea(Word wordToEdit)
----> private void CreateDeleteWordBtn(int lessonIdx, ref int wordIdx)
------> private void BuildLessonWordsCache(int lessonIdx, bool refresh)
--> private void CreateMainMenuBtn()
```


ManageLessons
Class
MonoBehaviour
Fields
allLessonsScrollPosition
areaBackgroundImg
backgroundImg
lessonNames
lessonWordsScrollPosition
lessonWordTexts
playerData
selectedLessonBtnIdx
selectedWordBtnIdx
smartMenuSkin
titleImg
wordAreaBackgroundImg
Properties
_createButtonStyle
_deleteButtonStyle
_generalButtonStyle
_headerStyle
_inputFieldsStyle
_mainMenuButtonStyle
_subHeaderStyle
Methods
BuildLessonNamesCache
BuildLessonWordsCache
CreateAllLessonsArea
CreateCurrentLessonArea
CreateDeleteLessonBtn
CreateDeleteWordBtn
CreateGUI
CreateLessonNameTextField
CreateMainMenuBtn
CreateNewLessonBtn
CreateNewWordBtn
CreateWordEditorArea
CreateWordHintTextArea
CreateWordTextTextField
OnGUI