Letter Lizard

CS798 Design Document

CS798 Group 4 - Afiya Nusrat, Alexander Pokluda, Michael Wexler

Introduction

We are creating a game called Letter Lizard for our CS798 project. Letter Lizard is a letter rearrangement game, based off of the online game *Text Twist*. The player is presented with a set of scrambled letters and the goal of the game is for the player to form as many words from the set of letters as possible before the timer runs out.The game will be implemented as a graphical user interface (GUI) application in three different scripting languages: Python, Ruby and JavaScript. We will use a simple game development framework with each application for programming the user interface and managing the user interaction. In order to make the game more engaging and enjoyable, we will enable the user to customize their gameplay experience by providing options to set the number of rounds, the time per round, and the difficulty for each game. Although we wish to create a game that is fun and enjoyable, our primary objective is to compare and contrast the features of each programming language. Where possible, we will strictly adhere to the same design for each implementation while also using the idiosyncratic features of each language as much as possible. The following sections discuss the gameplay, implementation details, challenges and division of work in more detail.

Gameplay

In Letter Lizard, a series of scrambled letters are displayed to the user and the aim is to create as many correct English words as possible before the timer runs out. The various user interaction steps are as follows:

- On beginning the game, a welcome screen or a ‘Splash Screen’ is loaded and displayed to the user. This is the initial screen that is shown to the user before gameplay.The user needs to hit the ‘spacebar’ to proceed to the ‘Main Menu’ screen. (Figure 1)
- The Main Menu allows the user to configure the gameplay. The user may set the difficulty level, number of rounds and the time per round. We provide three configurable difficulty levels; Easy, Medium and Hard. The Main Menu provides two options: ‘Start’ to begin playing the game or ‘Quit’ to exit the game. (Figure 2)
- Once the user clicks on Start, he is transported to the ‘Gameplay’ screen. This is where the actual game will be played. A series of scrambled letters are displayed to the user and a reverse timer begins to count down. The user has to type out as many correct english words as possible constructed from the scrambled letters before the timer reaches 0. The default game lasts for 2 minutes. (Figure 3)
- The game screen also contains buttons for ‘Hints’, ‘Shuffle’ and ‘Main Menu’. At any time in the game, the user may click on ‘Hints’ . Doing so would reveal a few letters already put in place to aid the user in forming a word. (Figure 4)

Figure 1: Splash Screen

Figure 2: Main Menu

Figure 3: Game Screen

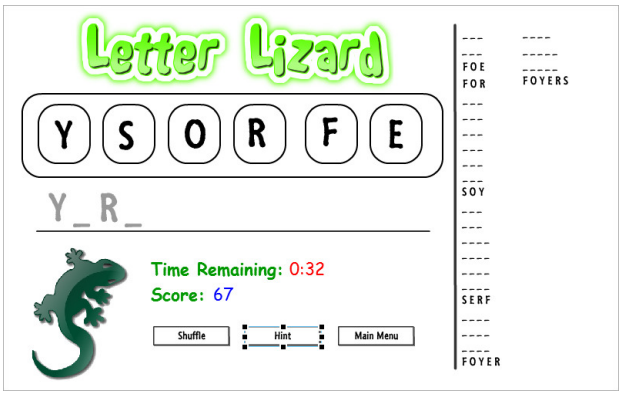

Figure 4: Game Hints

Gameplay Scenarios:

There can be two scenarios at any given instance:

- On entering a correct word: After verification by our game engine, the word is added to the list of correct words displayed on the screen and the score is incremented by the scoring function.
- On entering an incorrect word: The game engine verifies that the word is incorrect and notifies the user by displaying an ‘Incorrect word’ message. The score remains unchanged.

Implementation Details

Choice of Languages

We have chosen to implement our game in Python, Ruby and JavaScript. For each language implementation, we will also use a game framework to help abstract away some of the low-level system details associated with drawing graphics and handling user input. Our choice of languages, as well as the game framework that we will use for each language are described below.

Python

Python is a high-level, general purpose programming language. It has easy-to-use syntax, and very user-friendly coding constructs [1]. Python requires the use of whitespace in order to determine program flow. For instance, all the statements pertaining to the execution of an if-statement must be indented relative to the conditional clause. This is instead of using braces. The benefit of this is that it enforces clean, indented code, which makes it easier to understand program flow and execution.

Game Framework

We will be using Pygame as our game engine for Python. Pygame makes it possible to create games quite easily, and has a very user-friendly API for generating games in a very useful manner.

Ruby

Ruby is an object-oriented, dynamic, reflective, general-purpose programming language [2]. In Ruby, everything is an object, which makes it possible to create software systems that are easily modified and maintained, with code that is easily readable.

Game Framework

We will be using Gosu as our game framework for library. Gosu is a popular framework for making games in Ruby, with a strong user-base and user-friendly coding features.

JavaScript

JavaScript is a browser-based programming language. Combined with HTML5, JavaScript makes it possible to create rich, interactive browser games [3].

Game Framework

We are currently investigating a good game library for JavaScript, and are currently comparing the most commonly used game engines, including EaselJS, CraftyJS, and LimeJS.

Design

For each implementation, our application consists of a main driver file, which reads in games from a file games.txt. In Python, we have named this main application letter_lizard.py. These games are generated ahead of time by another program, in the case of Python, called game_generator.py. In each implementation, there is an infinite loop, which consists of three main steps: 1) process all events, 2) perform game logic, and 3) re-render entire screen. This high-level architecture is illustrated in Figure 5.

In general, it is not a good idea to re-render the entire screen too often, since it is computationally expensive. However, since our application is not very graphically intensive, the latency would not be noticeable, and hence the benefit of simplicity of our code due to re-rendering of the screen would outweigh the latency cost.

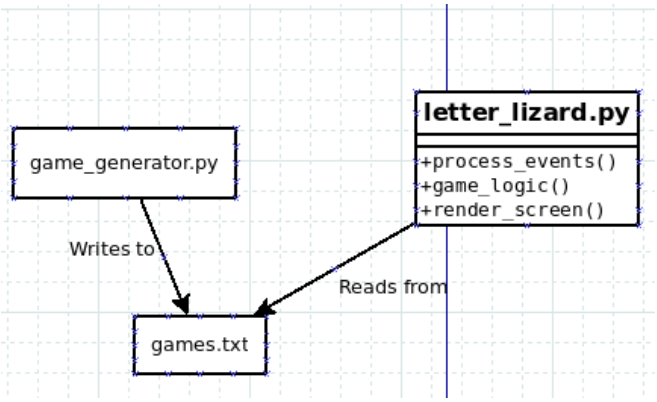

Figure 5: High-level structure of our game implementation

Game Generator

We have already implemented a game generator in Python that can be run as a standalone, command line program or be imported as a module. The game generator works by loading a set of dictionaries, generating a set of scrambled letters, and then finding dictionary words that can be formed by the set of scrambled letters. The set of scrambled letters, together with the dictionary words that can be formed, is called a “game.” The output of the tool is a text file where each line consists of the set of scrambled letters followed by all of the dictionary words that can be formed from those letters. The game generator accepts parameters that configure the number of scrambled letters in each game, the number of games to generate, and the difficulty of each game. The dictionary that we used (Spell Checking Oriented Word Lists by Kevin Atkinson) is partitioned into words that are more frequently used and less frequently used in common English. When choosing a harder difficulty, the list of words for the user to find will contain more words that are used less frequently.

When generating the set of scrambled letters, we pick each letter of the alphabet with a probability corresponding to the frequency that it appears in the dictionary (which we precomputed ahead of time). In order to find the dictionary words that can be formed, we considered two approaches: 1) generate all possible combinations of each possible size of letters from the scrambled set and check if that “word” is in the dictionary, and 2) iterate through the dictionary and see if each dictionary word can be formed from the set of scrambled letters. The first approach takes time O(n! log n) while the second approach takes O(501,002) which is the size of the dictionary including all of the least frequently used words (technically, this is O(1)). While the first approach is faster for games with approximately 8 letters or less, it is impractical for generating games with 11 letters or more. For this reason, we chose to implement the game generator using the second approach which is more predictable and performs adequately for all games sizes.

Challenges

There are several challenges that we expect to encounter. In order to simplify the process of drawing the user interface and handling user interaction for each implementation, we have decided to use game development libraries. This will allow us to abstract away many of the

low-level system details and focus on our game logic. However, each has a different interface for drawing graphics, handling user interaction, and managing game state. We will overcome the challenge of having to work with different interfaces by using only the basic features from each game framework (such as basic interface drawing code) and writing our code in a platform independent manner. Where necessary, we will code an interface layer in our application so that the logic and design remains the same across each implementation.

Another related challenge that we expect to encounter is making sure that each implementation follows a consistent design. We will overcome this challenge by sketching out a basic design for our game and then creating a “reference implementation.” Rather than working on all three implementations in parallel, we will complete most of the implementation of the Python version first. We will then implement the Ruby and JavaScript versions in parallel using the same function and class names as much as possible.

There is an inherent challenge learning each language and being able to use it effectively, including the idiosyncrasies of each language. We will overcome this challenge by devoting sufficient time to studying each language and take it upon ourselves to learn each language in depth. Each group member will be assigned to one implementation and will be responsible for developing a thorough understanding of their assigned language; however, we will each review each implementation and provide suggestions for how to better use language features to the primary developer.

Division of Work

As we will be implementing the game in three different languages, each language will have one lead developer and one secondary developer. The lead developer will be in-charge of developing the entire game in the assigned language with helpful inputs from the secondary developer. However, these are ‘loose’ assignments as opposed to strict and we expect that each one of us will be involved in all the implementations as we want to familiarize ourselves with all three of our chosen languages. Testing activities will be undertaken by the other two members apart from the lead. We hope that each member will be involved in the development and testing process in all the three implementations. We have divided our work as follows:

Python:
Primary developer: Michael
Secondary developer: Afiya

JavaScript:
Primary developer: Alex
Secondary developer: Michael

Ruby:
Primary developer: Afiya
Secondary developer: Alex

We will all be involved in developing the initial prototype so that we can familiarize ourselves with the game design and use the same variable and function names. This way we hope to parallelize and imitate our design in the other two languages. Maintaining naming consistency within the three codes will also help us in comparing and contrasting our implementations for the final report.

References

1) http://en.wikipedia.org/wiki/Python_%28programming_language%29
2) http://en.wikipedia.org/wiki/Ruby_%28programming_language%29
3) http://en.wikipedia.org/wiki/JavaScript