TypeRoids - Typing Accuracy Asteroids - Report

Marios Igkiempor - 10335752

May 2019

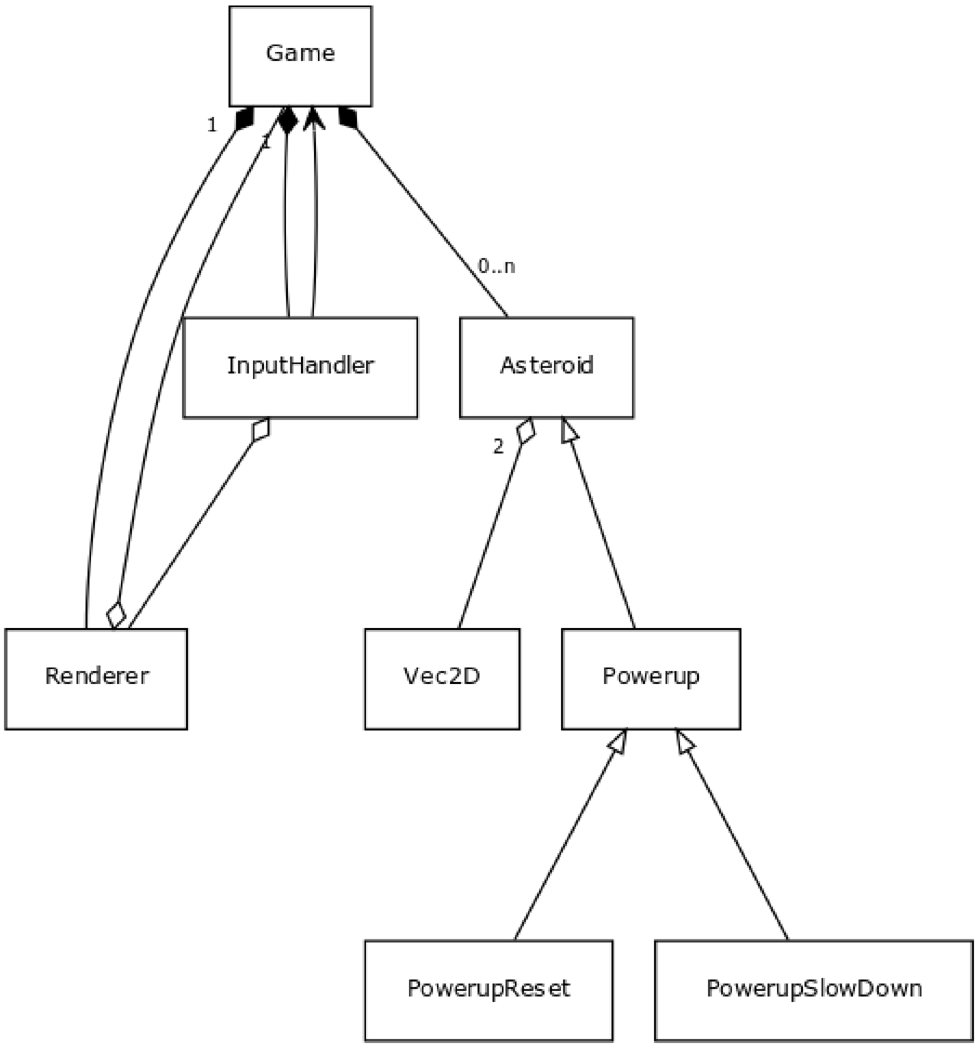
1 Description of the Game

The game is based on typing words(referred to as asteroids) as they fall down the screen. Typing a word correctly deletes it from the screen. Typing a word incorrectly still deletes it from the screen, but replaces it with another word. The new word is shorter with a length equal to the size of the original word minus the number of letters correctly typed, with a minimum length of 3. Occasionally, magenta power-up words appear on the screen. These words are either *reset* or *slowdown*. The reset word moves all the remaining words to the top of the screen, whereas the slowdown word slightly reduces the speed at which words fall down the screen. The player loses when any word that isn't a power-up word touches the bottom of the screen. The player wins when all words have been removed from the screen.

As a dependency, the game requires the SDL2 library, on which it relies for the graphics rendering. SDL was chosen for its cross platform capabilities, allowing the game to be compiled and played on many systems, including all modern computer operating systems.

2 UML Expression of Classes

Figure 1: UML Diagram of Classes And Relationships in the Program



3 Fulfillment of Requirements

The program uses C++ object oriented techniques appropriately and correctly. Examples of inheritance and polymorphism are achieved through the use of the base class `Asteroid` and its subclass `Powerup`. `Powerup` is an abstract class which declares a pure virtual method `UsePower()`. Making this method pure means that an object of type `Powerup` cannot be instanciated directly. `Powerup` is in turn subclassed into `PowerupSlowDown` and `PowerupReset`, which both define their own implementation of the `UsePower()`. `PowerupSlowDown` and `PowerupReset` objects are then spawned into the world with a probability defined in the `Constants.h` file. This hierarchy of classes allows the program to store all the types of ovject in a single `vector` of `Asteroid` pointers, and cast the pointers in the vector to use subclass-specific methods when appropriate.

The simple AI implemented in the game is the one that finds the closest asteroid to the word typed by fuzzy string matching. It uses the Levenshtein distance between every asteroid and the word typed to determine which asteroid is most similar to the input word. To optimise the process, it stops calculating Levenshtein distances once a distance of 0 has been found, as it means the word has been typed correctly. The nearest edit distance, if not 0, is then used as a determinant for the size of the new word to spawn.