Software Engineering Group Project
The Project Maintenance Manual

Author:	Nathan Williams - naw21, Samuel Jones - srj12, Rhys Evans - rhe24, Lampros Petridis - lap12 Alex Thumwood - alt38
Config. Ref.:	GP01-FS-PMM
Date:	2018-05-01
Version:	1.0
Status:	RELEASE

Department of Computer Science,
Aberystwyth University,
Aberystwyth,
Ceredigion, SY23 3DB,
U.K.

CONTENTS

1 INTRODUCTION

1.1 Purpose of this Document

The purpose of this document is to act as a point of reference for future installers or maintainers of the JoggleCube project. This document should serve to answer any questions they may have or allow them to gain a deeper understanding of the code base.

1.2 Scope

This document covers anything that may effect the maintenance and installation of the JoggleCube application that is not mentioned in the design document[3].

1.3 Objectives

This Document aims to:

- Answer any questions a maintainer may have about maintaining the project.
- Provide an overview of all the modules, methods, algorithms and major data structures used in this program.
- Explain how to build and test the project source.
- Cover all external file uses and how to interface with them
- Provide suggestions for future improvements.

2 Program description

JoggleCube is a single player computer game. The purpose is to form as many words as possible from a given 3 by 3 by 3 grid of randomly generated letters, within a three minute time period (indicated to the player with timer).

The player is able to compile a list of words by moving around the grid horizontally, vertically or diagonally while making sure to not reuse any of the available letters. The more words the player manages to form within the game time, the higher the score will be. Different letters have different point values, so the structure and complexity of submitted words can also effect the player's final score.

3 Program structure

3.1 Brief overview

The program is split into two parts; the frontend and the backend. This is so the frontend or backend logic can be swapped out independently if/when each part is updated. The program begins in the Main class, where the backend and frontend are launched.

A singleton model is used in both the frontend, and the JoggleCube class in the backend, to allow ease of communication between the front end and back end of the system. The IJoggleCube Interface is the definition of how the frontend interacts with the back end. The backend can directly control most the view controller classes, however it very rarely does this, due to the event driven nature of the system.

4 Control flow

After it's initial launch, the program is event driven. You can follow the control of the program by viewing the sequence diagrams in section 5.1 of the Design Specification[3].

4.1 Program modules

The program implements the Model, View Controller (MVC) pattern, and therefore the system is comprised of two modules: Model and UI, with the UI module handling both the 'View' and 'Controller' aspect of the pattern. This modular approach allowed the system's logic and interface to be developed independently, making the codebase easier to maintain, and more robust. Both these modules are described below.

4.1.1 Model

This module contains all of the classes needed to perform the game operations and handle the system logic. Each class within this module handles a specific aspect of the system.

4.1.2 UI

This module contains all of the classes needed to manage the system's User Interface. The module is split into general UI classes and FXML controller classes. The general UI classes are used in the system to manage the frontend's control flow and ensure the correct scenes and overlays are displayed. It also handles the program's popup dialogs.

4.2 List of methods

4.2.1 Main Class

4.2.1.1 main(java.lang.String[] args)

Returns: static void
Starts the JavaFX thread and calls the start method

4.2.1.2 start(javafx.stage.Stage primaryStage)

Returns: void
Starts the JavaFX

4.2.2 Block Class

4.2.2.1 getLetter()

Returns: String
Returns the letter in the block.

4.2.2.2 setLetter()

Returns: void
Sets the letter of the block and handles it.

4.2.3 Cube Class

4.2.3.1 getBagOfLetters()

Returns: ArrayList<String>
Returns the bag of letters.

4.2.3.2 getBlock(int x, int y, int z)

Returns: Block
Returns the given block using the parameters.

4.2.3.3 getCube()

Returns: Block[][][]
Returns the Block[][][], the 3D array that makes up the cube.

4.2.3.4 getNeighbours(int x, int y, int z)

Returns: ArrayList<int>
Returns an arraylist of Int[] that contains all of the neighbours of the given coordinates of the block.

4.2.3.5 getScores()

Returns: HashMap<Letter, Value>
Returns the HashMap of the scores using key-value pairs.

4.2.3.6 loadCube(java.util.Scanner file)

Returns: boolean
Loads the cube using the Scanner passed to the method, into the correct parts of the current object.

4.2.3.7 populateCube()

Returns: void
Creates the random cube from the bag of letters and the the cube given

4.2.3.8 saveCube(java.io.PrintWriter file)

Returns: boolean
Saves the cube in the correct format when given the passed PrintWriter object.

4.2.3.9 setBlock(int x, int y, int z, Block block)

Returns:void
Sets the block at the coordinates provided.

4.2.3.10 setLanguage(java.lang.String language)

Returns: void
Sets the current loaded language

4.2.4 Dictionary Class

4.2.4.1 getDictionarySize()

Returns: int
Returns the size of the local HashMap.

4.2.4.2 loadDictionary(java.lang.String filename)

Returns: void
Loads the dictionary into the local HashMap handled by the object that is dictionary using the filename given.

4.2.4.3 searchDictionary(java.lang.String word)

Returns: boolean
Searches the dictionary utilising the HashMap for the word and returns true if the word is present false if not.

4.2.5 GameTimer Class

4.2.5.1 finishTimer()

Returns: void
Ends the game.

4.2.5.2 getCurrentTime()

Returns: java.time.Duration
Returns the current time remaining.

4.2.5.3 interrupt()

Returns: void
Sets interrupt variable to true.

4.2.5.4 isInterrupt()

Returns: boolean
Returns the boolean interrupt.

4.2.5.5 resetTime()

Returns: void
Resets currentTime back to default duration

4.2.5.6 run()

Returns: void
Launches startTimer in a seperate thread.

4.2.5.7 setCurrentTime(java.time.Duration currentTime)

Returns: void
Sets the Duration of currentTime.

4.2.5.8 startTimer()

Returns: void
Launches and controls the game timer in a separate thread.

4.2.6 HighScores Class

4.2.6.1 addScore(IScore score)

Returns: void
Adds high scores to the arraylist.

4.2.6.2 getHighestScore()

Returns: IScore
Sorts the arraylist and returns the high score

4.2.6.3 getScores()

Returns: ArrayList<IScore>
Returns all the scores in the arraylist

4.2.6.4 loadScores(java.util.Scanner file)

Returns: void
Loads an arraylist containing the high scores

4.2.6.5 saveScores(java.io.PrintWriter file)

Returns: void
Saves an arraylist containg the high scores to a file

4.2.6.6 toString()

Returns: String
Returns string. This method is used for testing

4.2.7 JoggleCube Class

4.2.7.1 clearHighScores()

Returns: void
Clears all highscores in the overall highscores variables as well as the stored files!

4.2.7.2 generateRandomGrid()

Returns: void
Starts a brand new random game; when called it resets variables and create a new random grid.

4.2.7.3 getCubeData()

Returns: String[][][]
Formats the cube into a way that can be utilised by the front end.

4.2.7.4 getCurrentCubeHighScores()

Returns: ObservableList<IScore>
Return the current cube's highscores in a format usable by JavaFX.

4.2.7.5 getGamesStateNew()

Returns: boolean
Returns whether or not the game is 'fresh' or loaded from a file.

4.2.7.6 getHighestScore()

Returns: int
Returns the top highscore.

4.2.7.7 getInstance()

Returns: static IJoggleCube
Returns an instance of the JoggleCube object.

4.2.7.8 getLoadedDictionaries()

Returns: HashMap<String,Dictionary>
Returns a HashMap with String and Dictionary as the key, value pair, of all of the currently loaded dictionaries.

4.2.7.9 getOverallHighScores()

Returns: ObservableList<IScore>
Returns the overall HighScores across all cubes in a format usable by JavaFX.

4.2.7.10 getRecentGrids()

Returns: ObservableList<String>
Returns a list of all saved grids and returns them in a format expected by the front end.

4.2.7.11 getScore()

Returns: int
Returns the score for the current game.

4.2.7.12 getWordScore(java.lang.String word)

Returns: int
Generates and returns the word score for this word, using each letter's scrabble score * 3

4.2.7.13 interruptTimer()

Returns: void
Stops the timer.

4.2.7.14 loadGrid(java.lang.String filename)

Returns:boolean
Given the filename parameter, it will load the saved game and return a value depending on whether it's successful.

4.2.7.15 loadNewDictionary()

Returns: void
Rebuilds the backend dictionary object with the currently selected language.

4.2.7.16 resetGameState()

Returns:void
Takes all the variables and changes them so that the game can be started again. If the grid has been saved previously then it saves over it's file.

4.2.7.17 saveGrid(java.lang.String filename)

Returns: boolean
Save the grid that is currently loaded, using the given filename, in the folder where save games are stored.

4.2.7.18 saveOverallScores()

Returns:void
Saves the overall highscores to a file

4.2.7.19 setLanguage()

Returns: void
Using the first to letters as an example, sets the language (using "en" for American English.)

4.2.7.20 setName(java.lang.String name)

Returns: void
Sets the name of the player.

4.2.7.21 startTimer()

Returns: void
Starts the in0game timer in a new process thread.

4.2.7.22 testWordValidity(java.lang.String word)

Returns: boolean
Tests whether or not the word is a valid given the parameters in the functional requirements.

4.2.8 Score Class

4.2.8.1 equals(java.lang.Object o)

Returns: boolean
Checks whether score is equal.

4.2.8.2 getDate()

Returns: String
Gets the date of a given high score entry.

4.2.8.3 getName()

Returns: String
Gets the name of a given high score holder.

4.2.8.4 getScore()

Returns: int
Gets the score value of a given high score entry.

4.2.8.5 saveScore(java.io.PrintWriter file)

Returns: void
Saves this score to the file.

4.2.8.6 toString()

Returns: String
Prints information about the score.

4.2.9 Dialog Class

4.2.9.1 getTextInputDialog()

Returns: javafx.scene.control.TextInputDialog

4.2.9.2 isValidInput(java.lang.String input)

Returns: boolean
Verifies that the user input is correct

4.2.9.3 showConfirmationDialog(java.lang.String title, java.lang.String contentText)

Returns: java.util.Optional<javafx.scene.control.ButtonType>
Create a confirmation dialog to prompt the user for confirmation for a given action.

4.2.9.4 showInformationDialog(java.lang.String title, java.lang.String contentText)

Returns: void
Create a information dialog to notify the user

4.2.9.5 showInputDialog(java.lang.String headerText, java.lang.String contentText, java.lang.Strin defaultValue, javafx.scene.image.ImageView graphic, boolean allowCancel)

Returns: String
Create a text input dialog to prompt the user for information

4.2.10 Navigation Class

4.2.10.1 add(ScreenType name, javafx.fxml.FXMLLoader loader)

Returns: void
Adds a screen to the hashmap.

4.2.10.2 getInstance()

Returns: static IViewNavigation
Gets the instantiated instance of the Navigation singleton.

4.2.10.3 getMain()

Returns: javafx.scene.Scene

4.2.10.4 getScreens()

Returns: java.util.HashMap<ScreenType,javafx.fxml.FXMLLoader>

4.2.10.5 hideOverlay(ScreenType overlay, BaseScreen parent)

Returns: void
Hides a given overlay.

4.2.10.6 remove(ScreenType name)

Returns: void
Removes a Screen from the HashMap.

4.2.10.7 setMainScene(javafx.scene.Scene main)

Returns: void
Set the main screen of the game.

4.2.10.8 showOverlay(ScreenType overlay, BaseScreen parent)

Returns: void
Shows a given overlay.

4.2.10.9 switchScreen(ScreenType newScreen)

Returns: void
Switches to a given screen.

4.2.11 Settings Class

4.2.11.1 clearHighScores()

Returns: void
Clear the highscores and prompt user.

4.2.11.2 getAlreadySelectedColor()

Returns: String
Gets the colour used to represent already selected cubes.

4.2.11.3 getAvailableColor()

Returns: String
Gets the colour used to represent available cubes.

4.2.11.4 getCurrentlySelectedColor()

Returns: String
Gets the colour used to represent currently selected cube.

4.2.11.5 getCurrLang()

Returns: static String

4.2.11.6 getCurrLangPrefix()

Returns: static String

4.2.11.7 getInstance()

Returns: static ISettings
Returns an instance of the singleton class settings.

4.2.11.8 getLanguages()

Returns: static String[]

4.2.11.9 getTimerLength()

Returns: static int

4.2.11.10 getUnavailableColor()

Returns: String
Gets the colour used to represent unavailable cubes.

4.2.11.11 isColorBlindEnabled()

Returns: boolean
Check if the colour blind option is enabled.

4.2.11.12 setCurrLang(java.lang.String currLang)

Returns: static void

4.2.11.13 setTimerLength(int timerLength)

Returns: static void

4.2.11.14 toggleColourBlind()

Returns: void
Set the colour blind option to true/false

4.2.12 UIlass

4.2.12.1 getInstance()

Returns: static IFrontend
Get the singleton instance of the UI object

4.2.12.2 initialize(javafx.scene.Scene main)

Returns: void
Initialize the game by creating the necessary scenes and starting the JavaFx

4.2.13 BaseOverlay Class

4.2.13.1 parentController

Returns: protected BaseScreen
An instance of the overlay's parent controller

4.2.14 BaseScreen Class

4.2.14.1 root

Returns: protected javafx.scene.layout.StackPane The root node of a given screen

4.2.15 End Class

4.2.15.1 btnHighScoreClicked()

Returns: void
When the High Score button is pressed, change scene to high score screen.

4.2.15.2 btnMenuClicked()

Returns: void
When the 'return to menu' button is clicked change scene to menu scene.

4.2.15.3 btnReplayClicked()

Returns: void
When the 'replay' button is clicked restart a game

4.2.15.4 btnSaveClicked()

Returns: void
When the 'save' button is clicked prompt user to chose a save location.

4.2.15.5 getHighScore()

Returns: String

4.2.15.6 getInstance()

Returns: static End

4.2.15.7 getScore()

Returns: String

4.2.15.8 prepView()

Returns: void

4.2.16 GameView Class

4.2.16.1 btnClearClicked()

Returns: void

4.2.16.2 btnEndGameClicked()

Returns: void
When the End Game option is clicked it will load the EndGui scene.

4.2.16.3 btnExplodeClicked()

Returns: void
Handles the explode button being clicked.

4.2.16.4 btnMenuClicked()

Returns: void
Handles the hamburger menu being clicked.

4.2.16.5 btnSubmitClicked()

Returns: void

4.2.16.6 getColorBlindIcon()

Returns: javafx.scene.control.Button

4.2.16.7 getCubeContainer()

Returns: javafx.scene.control.TabPane
Gets cubeContainer

4.2.16.8 getDialog()

Returns: Dialog

4.2.16.9 getFoundWords()

Returns: javafx.collections.ObservableList<java.lang.String>
Returns the list of found words so it can be used in the backend as it is currently only stored in the frontend.

4.2.16.10 getGridDisplayer()

Returns: GridDisplayer
Gets GridDisplayer.

4.2.16.11 getHamburgerContext()

Returns: javafx.scene.control.ContextMenu
Gets hamburgerContext

4.2.16.12 getInstance()

Returns: static GameView

4.2.16.13 getMenuButton()

Returns: javafx.scene.control.Button
Gets the menuButton

4.2.16.14 getScoreLabel()

Returns: javafx.scene.control.Label
Gets the scoreLabel.

4.2.16.15 getText()

Returns: String
Gets the text input within textField

4.2.16.16 getTimerLabel()

Returns: void
Gets the timerTable.

4.2.16.17 prepView()

Returns: void
Initializes the game screen.

4.2.16.18 setText(java.lang.String text)

Returns: void
Sets the string of text in to the textField variable

4.2.16.19 setTimerLabel(javafx.scene.control.Label timerLabel)

Returns: void

4.2.17 GridDisplayer Class

4.2.17.1 buildGrids(java.lang.String[][][] letters)

Returns: void
sets up the 3d enviroment, loads letters into labels and boxes and adds them to the display.

4.2.17.2 isActive(int x, int y, int z)

Returns: void

4.2.17.3 setAllActive()

Returns: void
Resets the cube so every cube can be clicked.

4.2.17.4 setInActiveP(int x, int y, int z)

Returns: void

4.2.17.5 toggleExplode()

Returns: void

4.2.18 Help Class

4.2.18.1 btnLeftNavClicked()

Returns: void
Handles the right navigation being pressed by decreasing the current help screen index by one and looping around using modulo, updating the currently displayed page

4.2.18.2 btnRightNavClicked()

Returns: void
Handle the right navigation being pressed by increasing the current help screen index by one and looping around using modulo, updating the currently displayed page.

4.2.18.3 closeBtnClicked()

Returns: void
Handles the close button of the overlay being clicked

4.2.18.4 getHelpPageContainer()

Returns: javafx.scene.SubScene

4.2.18.5 getInstance()

Returns: static Help

4.2.18.6 initialize(java.net.URL location, java.util.ResourceBundle resources)

Returns: void
Initializes the help overlay for first time use

4.2.18.7 prepView()

Returns: void
Preps the overlay by loading the default page every time

4.2.19 HighScore Class

4.2.19.1 changePage()

Returns: void
Utility function to change the page of the high score table; if it's not filtering to the top 10, compare the data in table:
highScoreTable.getItems().equals(overallScores);

4.2.19.2 getCurrentCubeHighScores()

Returns: javafx.collections.ObservableList<IScore>

4.2.19.3 getInstance()

Returns: static HighScore

4.2.19.4 getOverallScores()

Returns: javafx.collections.ObservableList<IScore>

4.2.19.5 getText()

Returns: String

4.2.19.6 highScorePageLabel()

Returns: javafx.scene.control.Label

4.2.19.7 initialize(java.net.URL location, java.util.ResourceBundle resources)

Returns: void
Sets up the score tables

4.2.19.8 prepView()

Returns: void
Populates IScore table with highscore data

4.2.19.9 setItems()

Returns: short

4.2.19.10 setText()

Returns: short

4.2.20 LoadGrid Class

4.2.20.1 btnBackClicked()

Returns: void
When the back button is clicked it will load the Start scene.

4.2.20.2 btnStartGridClicked()

Returns: void
When the Start Grid button is clicked it will load the Game scene.

4.2.20.3 getFileName()

Returns: String

4.2.20.4 getInstance()

Returns: static LoadGrid

4.2.20.5 handleMouseClick(javafx.scene.input.MouseEvent event)

Returns: void
Handles when a user clicks an option from the selection.

4.2.20.6 prepView()

Returns: void
Gets recently played cubes from backend.

4.2.21 SettingsOverlay Class

4.2.21.1 clearHighScoreClicked()

Returns: void
Handles the clearing of highscores.

4.2.21.2 closeBtnClicked()

Returns: void
Handles the close button of the overlay being clicked.

4.2.21.3 colorBlindToggleClicked()

Returns: void
Handles Colour Blind Toggle.

4.2.21.4 getInstance()

Returns: static SettingsOverlay

4.2.21.5 initialize(java.net.URL location, java.util.ResourceBundle resources)

Returns: void
Called to initialize a controller after its root element has been completely processed.

4.2.21.6 prepView()

Returns: void

4.2.22 Start Class

4.2.22.1 btnLoadGridClicked()

Returns: void
When the Load Grid button is clicked it will load the LoadGrid scene.

4.2.22.2 btnStartNewGridClicked()

Returns: void
When the Start New Grid button is clicked it will load the Game scene with a new grid.

4.2.22.3 getInstance()

Returns: static Start

4.2.22.4 getLanguageSelector()

Returns: javafx.scene.control.ComboBox<java.lang.String>

4.2.22.5 initialize(java.net.URL location, java.util.ResourceBundle resources)

Returns: void
Called to initialize a controller after its root element has been completely processed.

4.2.22.6 prepView()

Returns: void
Prepares the View to start

4.2.23 BlockFx Class

4.2.23.1 changeState(BlockState state)

Returns: void

5 Algorithms

5.1 Dictionary load

Dictionary Load has already been discussed in detail in the design document[3].

To change the dictionary that is being loaded in you will need a file that adheres to the section 8.4. Then you need to make sure that the dictionary file also adheres to the naming convention also in 8.4. The file prefix also needs to adhere to a language that can be selected in the main menu on the front end; to assure this, you need to go to the "Settings.java" file in the Main.UI.Controllers package and add the language to the private instance variable array languages[]. After this is completed, the language will load and also be available for selection. However, this will only affect the dictionary itself: to make the required letters generate within the cube it you also need to load a letters file (the way in which to do this is discussed in section 5.6.)

5.2 Dictionary search

Dictionary Search has already been discussed in detail in the design document [3].

This algorithm should work regardless of the language being implemented, so there's very little reason for it to be edited in any way.

5.3 Neighbouring blocks

Neighbouring blocks has already been discussed in detail in the design document [3].

Increasing or decreasing the size of the cube requires changing a lot of hard coded values in each of the for loops: for example, the point at which the for loop ends, in most cases, would need to be incremented by 1 for each time the cube would have been increased in size by 1.

5.4 Saving the grid

Saving grids has already been discussed in detail in the design document [3].

Changing the method in which grid data is saved requires changing a lot of the core code; it's likely that this would be a struggle to accomplish without removing the entire pre-existing algorithm used for saving, and starting from scratch. It's recommended that this process is only undertaken if you are sure of the benefits of your new saving algorithm (i.e. increasing efficiency by saving grids in an XML format)

5.5 Loading the grid

Loading grids has already been discussed in detail in the design document [3].

The loading of the grid should use the same format as the saving but in reverse. There is no real reason to change this unless you are changing the size of the cube or the way scores are handled; much like Save Grid 5.4 hard-coded values are used for loading files.

5.6 Random letter generation

Random letter generation is already discussed in detail in the design document [3].

The letter generation can be changed as long as you adhere to the way the letters are stored in 8.3. The letter generation is derived from the letter file given the first two letters of the file as the first two letters of the language you are using, i.e. English = en_letters. As long as the letters file is in the right part of the resources, and the pre-requisites for dictionary and a dictionary has already been met in section 5.1 for localising with a new dictionary and letter combination. An example of these changes can be seen utilising a Cymraeg/Welsh dictionary.

5.7 Creation of the JoggleCube file in the Documents folder

The creation of the JoggleCube file in the documents folder is already discussed in detail in the design document [3].

This algorithm is one of the few instances where another library is used; the commons-io.jar library is required for this to run properly as the algorithm utilises the copy entire files functionality the library offers.

If you want to change the location of where the JoggleCube documents folder is made then you will need to change the hard coded values of where the program is searching and subsequently copying the files to. It currently utilises the default value for System.home as a property to find the documents folder assuming that System.home is the User's directory on the computer.

5.8 Recent grid

The recent grid algorithm is already discussed in detail in the design document [3].

The recent grids could be changed to suit and would consequently need to be changed if the location of where the JoggleCube documents folder has been moved, because it searches the same hard coded location utilising the System.home value as the user's directory on the computer.

5.9 Displaying the grids

The Displaying of the grids is already discussed in detail in the design document [3].

This is a very much hard coded logic however if the aim is improvement of logic then that is certainly possible; indeed it has been theorised that the current method isn't necessarily the best way to handle it.

5.10 Loading overall high scores

The loading of the overall high scores is already discussed in detail in the design document [3].

This is very similar to Recent grid and creation of the JoggleCube document folder, it is hard coded to utilise the System.home environment variable to assume that it is the documents folder it is using.

5.11 Word score

Word score is already discussed in detail in the design document [3].

The easiest way to edit the way that things are scored it to either change the base scrabble values which are stored in the letters file 7.2.5. If the aim was to change the multiplier from x3 then it is recommended to find the hard coded x3 in the code and change it.

5.12 Help Overlay

In order to display a multi-page, navigable help page we implemented a carousel view. This included a sub-scene to represent each page, a row of usable page indicators and two navigation icons. Due to the fairly specific requirements this was implemented from scratch using FXML subscenes to inject the pages into the help overlay.

This algorithm is implemented in Help.java, it uses an initialized list of all the help pages available along with their FXML file names. Using this list it creates the correct amount of page indicators and allows circular navigation. In order to create and include a new help page, an FXML file must be created in the 'helppages' package and included in the 'helpScreens' array list.

5.13 High Score Displayer

As there is no option in the JavaFX TableView to prevent column re-order it was necessary to prevent it manually. This was done by adding an event listener to a moved column and instantly moving it back to its original position.

5.14 Dialogue Validation

In order to ensure all user input from dialogs are correct a simple regex check for A-z and 0-9 is performed. The user is recursively prompted for the input until a valid input is provided. This is implemented in Dialog.java.

5.15 Grid Displayer

Grid Displayer has already been discussed in detail in section 5.2.13 in the design document[3].

Maintaining the implementation of this algorithm in its current state will be quite a challenge. If changes need to be made to how the grid displays a full re-write of the algorithm is recommended, the beginnings of this is located in the UI/Cube directory. The current implementation was originally intended as a trial, simply to get the software to a working state, but due to time constraints the re-write was never finished. The implementation currently relies on the css background of the 2d display which restricts it's adaptability significantly, and this limited implementation has lead to the overall design being slightly constrained.

5.16 Toggle Explode

To click the centre letter of the 3D grid we needed a way to move the other letters out the way. To achieve this, we created an explode toggle, which moves the left letters to the left and the right letters to the right to reveal the centre of the cube. This is done with a button that toggles a boolean variable. An animation time line is create for the movement by the method, based on the coordinate of the letters and the direction they need toggling.

6 The main data areas

6.1 Dictionary Storage

The way that dictionary is kept persistent is discussed in detail in the design document [3]. However the way it is loaded is by utilising the dictionary load (5.1). It is loaded into a HashMap<String, String>, where both Strings are the word being loaded. We utilised the HashMap due to it's ability to easily search the entire dictionary rapidly. Keeping search times below 1 millisecond in many cases. It is of course possible to change this structure, as only the search and load methods would need to be changed.

6.2 The Cube

The way that the cube is kept persistent is discussed in detail in the design document [3]. However, unlike dictionary it is not loaded from just one file and can be loaded from an infinite number of files given the file name. The actual loading and saving of these cubes are discussed earlier (5.5 and 5.4). The cube itself is stored in the program as a 3D matrix of Block objects, the most efficient method for a cube of defined size. Blocks are retrieved using their co-ordinates in an intuitive way: x, y, z being the co-ordinates of the first, second and third arrays respectively.

6.3 High Scores

The overall HighScores are loaded in from the .highscore file and stored in an ArrayList of 'Scores'. The decision to use an ArrayList was made to allow for dynamic sizing and easy access. In addition to being stored during runtime the HighScores ArrayList is stored persistently in a .highscores file (see 7.1.1).

6.4 Bag O' Letters

All letters that can be found on the grid are retrieved from the 'bagOfLetters' ArrayList; an ArrayList containing strings of all letters available. It is populated from the letters file, hence it was important that the letters used were dynamic and not hard-coded, in order to support different grid languages. It should be noted that the ArrayList is of type String and not char because several languages use multiple characters as a single alphabetical letter, for example (Rh) in Welsh. More information on the bagOfLetters can be found in section 5.3.3 of the Design Specification [3]

6.5 Scores HashMap

When the letters file (see 7.2.5) is being read into the system, each letter is stored in a HashMap along with it's corresponding score value within the game. Using a given letter as a key within the HashMap means its score value can be easily retrieved.

6.6 Screens HashMap

All the screens used by the game's User Interface are stored in a HashMap, in which the key is a ScreenType enum and the value an FXMLLoader object that encapsulates the scene and its corresponding controller. The decision to use a HashMap was made to allow easy retrieval of screens and increased code readability. An alternative method would have been to use an array and a set of constant integers to index it; for example (screens[START]). However we felt that using enumerated types was a cleaner and more robust solution.

7 Files

This program depends on several external files and resources; in this section we will discuss in reasonable depth the purpose, use and importance of each external file used.

7.1 Local JoggleCube Folder and Data

The program creates a JoggleCube folder in the user's Document directory with two additional subfolders: 'highscores' which is used to save the overall high scores, and 'saves' which is used to store grids. In order to create and populate this folder, all of its contents has been 'pre-created' within the 'Data' directory, located in /src/cs221/GP01/main/. The contents of this directory are then copied to the user's local 'JoggleCube' folder, ready for use by the program. Below is a description of the 'JoggleCube' folder's contents

7.1.1 highscores

This folder contains all of the user's high scores (see 8.2). In its default state this folder contains a file 'overAll.highscores'; this file is created each time the program is run. If the file already exists then its contents are overwritten.

7.1.2 savedgrids

This folder contains all of the user's saved grid files (see 8.1). By default it contains three .grid files (grid_1.grid, grid_2.grid and grid_3.grid); these are example files that can be used for testing, amongst other things.

7.2 Resources

The resources folder is located in /src/cs221/GP01/main/resource and stores the files needed by the application for the UI and cube behaviour.

7.2.1 css

This folder contains all the .css files needed to skin the FXML files. For each .css file there is a corresponding FXML file and two global .css files.

7.2.2 dictionary

This folder contains all the dictionary source files that are loaded by the program to generate and handle the grid. (See 8.4)

7.2.3 font

This folder contains the fonts required for the program's UI, most notably the digital clock font for the timer label.

7.2.4 img

This folder contains all the images used by the program. It is split into two sub-directories: help and icon. The 'help' folder contains all the .gif files used within the help overlay, the 'icon' folder contains all the icons and their variations used throughout the game's UI.

7.2.5 letters

This folder contains all the letter files used by the game to generate the grid and work out the word scoring. Within each letter file is a list of all the letters and their corresponding score and frequency. For each dictionary file there should be a corresponding letter file. (See 8.3)

7.2.6 view

This folder contains all the FXML files used within the game's UI; for each scene within the program there is a corresponding .fxml file. There is also a sub-directory, 'helppages', to store all of the .fxml files used in the help overlay's sub-scene.

8 Interfaces

In order to load information for use in various operations, the system interfaces with input files to populate data structures within the application. As the application only interfaces with plaintext files, the rules for interfacing are primarily focused on formatting. A breakdown of all interfaced files and their formatting rules are given below.

8.1 Grid file structure

Grid files (.grid) are used to store grids and associated data. A grid file contains the following information:

- Grid Language
- Grid Contents (Letters)
- Grid Highscores
  - Date and Time (YY/MM/DD HH:MM)
  - Score
  - Player Name

  In order to ensure the above information is read and written accurately and correctly these files must be formatted specifically. See Appendix C in the Design Specification for a breakdown of these formatting requirements. [3]

8.2 Overall high score file structure

Highscore files (.highscore) are used to store all the overall highscores of the game. These files store the following information:

- Date (YY/MM/DD)
- Time (HH:MM)
- Score
- Player Name

A full breakdown of the formatting requirements for this file can be found in Appendix D of the Design Specification [3]

8.3 Letters file structure

Letter files are used to store all the letters required for a grid, and their corresponding score and frequency (see 7.2.5.) In addition to the formatting requirements of this file's contents, it is also important that the file is named correctly. The naming format for this file should be as follows: <language prefix>_letters for example, 'en_letters'. The formatting requirements for the file itself can be found in Appendix B of the Design Specification [3]

8.4 Dictionary file structure

Dictionary files are the largest but simplest input file in terms of formatting. They are used to store all recognised words for a given language/locale. Much like letter files, they must be named correctly (<language_prefix>_dictionary). Each entry in the dictionary must be in a new line and in uppercase; a more detailed breakdown of the dictionary's formatting requirements can be seen in Appendix A of the Design Specification [3]

9 Suggestions for improvements

- Load UI element text from xml and add translations into other languages. This would allow the whole UI language to be changed and not just the Grid/Game language.
- Add sound effects and relevant options in the settings screen. This idea was toyed with during implementation but was removed due to time constraints.
- Re-Write the Grid Displayer to be easier to maintain and use. In it's current form the state machine is based on the background colours of the UI elements this limits it's capabilities. For example, with the Grid Displayer in it's current state, the player is unable to change the colours of the grid mid-game.
- After a re-write of GridDiplayer a de-selection or undo feature on each letter of a word would be possible to implement using a simple stack (The stack serving to keep track of progress.)
- Re-write the fxml and css to either be a higher resolution than the fixed 600x600px or use use responsive css to allow the window to be re-sized however the player would like.
- Increase the resolution of the gifs in the help screen using paid/more professional tools.
- Use Icon font or a more adaptable format such as svg for the Icons instead of using the fixed resolution and colour icons provided by the png images currently in use.
- Find a work around to the issue of accessing the home directory from network mounted drives, maybe prompt user to enter a custom save location.

9.1 Currently Known Bugs and Fixes

9.1.1 Starting new grid button not working

If the grid has been started and then quit 3 times, then the game will throw an error out of bounds exception when running the cube.populateCube() code in the Cube.java. We find that the game will not start, the new grid and will keep throwing exceptions every time we click the button, this has something to do with the bagOfLetters Array List in the same file causing the issue, given more time I would like to fix this or at least add a work around to compensate for the error. We would accomplish this by either adding a new Java based error, or checking for the size of the bag of letters and if it is below 0 we should reload the bag of letters to handle the error, it's more of a work around but it should still work. This would have been realised before code submission however the bug was only brought to light on how to reproduce the error post submission.

10 Issues that could arise when making changes

When editing the backend, make sure to consider that some methods call methods in the frontend such as the Timer display and score display. The display does reset between games but if these methods are called out of place they may update the display (such as displaying an unwanted score) at the beginning of the game before it is next updated with the current score.

When making changes to the styling of the UI it should be noted that although all FXML files have a corresponding .css file, all FXML files also use BaseStyle.css. Therefore some of the FXML node's styling is potentially sourced from there.

11 Physical limitations of the program

The Program requires a minimum of 1500MB of system memory to run, due to the overhead of the java virtual machine. Approximately 4MB of space for the compiled .jar and 1MB space for the Highscores and saved grids.

If the program is being run on a network mounted drive, the persistent save features will likely not work. This is due to the program being unable to locate the user's local copy of 'JoggleCube' (see 7.1). If this is the case, the user is notified via a dialogue popup that this feature will be unavailable whilst on a network mounted drive.

12 Rebuilding and Testing

The Building and Testing of this project is described using intellij idea[2], as this is the environment the project was developed in and is simple to use with IntelliSense features. However, the project can also be built and linked through command line and furthermore, most IDE's have an import project option, should that be your desired route.

12.1 Building

Creating the project from existing sources and pressing the build and run option. The libraries should be auto detected. If not go to project-Structure − > Libraries and manual add the lib folder in the root of the repository. This includes the testing and libraries used in the project.

12.2 Building a executable JAR

Going to Project-Structure − > artefacts − > (plus symbol) − > JAR − > from module with dependencies, add the main class and click ok. This will create a artefact with all the libraries and testing classes. To remove the testing libraries select all the libraries except the commons-io x.x.x library. Now apply the changes and close the window.

Next we need to build the artefact. Going into build − > artefacts − > and clicking build will build the .jar in out/artefacts/. A little clean-up is required to remove the testing classes from the jar. Opening the archive, you can remove everything in the /cs221/GP01/Test/ directory. This will have no effect on the running of the application, but will reduce the jar archive size.

12.3 Testing

A suite of tests have been written using Junit which are in the test folder and associated libraries. They can be found in src/cs221/GP01/test directory. The frond end tests are contained in the UI directory, the backend in the backend directory. There are also 3 files called xxxSuiteTests.java. One runs all tests on the system, one runs only the backend tests, and the other only runs the frontend tests.

REFERENCES

[1] *Software Engineering Group Projects* JoggleCube Game Requirements Specification. C. J. Price SE.QA.CSRS. 1.0 Release.

[2] *Intellij Idea* `https://www.jetbrains.com/idea/`

[3] *Design Specification* GP01-UIS-DS 2.0 Release.

[4] *User Interface Document* GP01-UIS-UCD 2.0 Release.

DOCUMENT HISTORY

Version	CCF No.	Date	Changes made to Document	Changed by
0.1	N/A	2018-05-01	Initial creation	NAW21
0.2	N/A	2018-05-02	Begun writing content	ALL
0.3	N/A	2018-05-03	More writing	ALL
0.4	N/A	2018-05-04	Fixing spelling and grammatical errors	ALT38
1.0	N/A	2018-05-05	Reviewed and finalised	NAW21