UNIVERSITY OF HERTFORDSHIRE

School of Computer Science

BSc Honours in Computer Science

6COM1056 - Software Engineering Project

Final Report

April 2019

TITLE OF PROJECT

MOBILE DEVELOPMENT OF CARROTO 2029, A MOBILE GAME DEVELOPED IN UNITY.

Author's initials and surname

W MACIEJEWSKI

Supervised by:

ALEXANDRA MARCZYK

University of Hertfordshire
School of Computer of Science

ASSIGNMENT BRIEFING FRONT SHEET (2018/19 Academic Year)

Assignment Title	Project Final Report	**Date submitted**	07/05/2019
Module Title	BSc Computer Science Project	**Module Code**	6COM1053/54/55/56
Tutor	Paul Morris	**GROUP or INDIVIDUAL Assignment**	Individual

****

STUDENT TO COMPLETE

Ethics Approval Statement (complete if appropriate)

I have not been issued with an Ethics Protocol for any study with human subjects associated with this project.

Relevant protocol number(s) :

I declare that this protocol is now complete, and that it was conducted in accordance with University of Hertfordshire ethics regulations.

(Comments on this assignment by students can be made on the back of the assignment briefing sheet).

By completing **BOX A** below, I certify that the submitted work is entirely mine and that any material derived or quoted from the published or unpublished work of other persons has been duly acknowledged. **[refer UPR AS12, section 7 and UPR AS14 (Appendix III)].**

Please print your forename and surname in capitals, provide your; - ID number, the study year code (e.g. CS1, ASE1), actual time spent on the assignment and your signature.

BOX A

Student Forename *(in CAPS please)*	**Student Surname** *(in CAPS please)*	**Student ID Number**	**Year Code**	**Actual Time Spent by the Student (hours)**	**Signature of Student**
WIKTOR	**MACIEJEWSKI**	**14162410**		**About 300**	Maciejewski Wikt

****

This sheet must be submitted with the assignment, signed and BOX A filled in.
LATE SUBMISSION WILL ATTRACT A STANDARD LATENESS PENALTY.

Contents

Chapter 1: Introduction

Motivation

Mobile computing is at the peak of its powers. When the first punch card computer was invented by Herman Hollerith to calculate the 1880 U.S. census (Zimmermann, 2017), nobody could have predicted that computing would evolve to the point where now an average person can carry a computer in their pocket which has incomparable processing power to the computers that were installed on the spacecraft for the Apollo 11 spaceflight. Since Apple first opened the App Store in 2008, there has been tremendous growth in the iOS app development field. By 2018 iOS developers cumulatively earned £100bn dollars (Richter, 2018). Consider all the other mobile platforms and their app stores and you're looking at an industry larger than many.

In 2017 the Apple app store contained about 2 million apps and the Google play store over 2.2 million (Saifi, 2017). Therefore, the exploration of the Mobile Development field seems to be necessary, since the objective of the project is not only to develop a mobile application but one which could potentially be successful. Gaming has been an interest of mine since I've been a kid. Hence, the application will be a game.

My personal motivation to explore this segment of Computer Science comes from an ambition of pursuing a career in the Mobile Development field, although I have not come into touch with this branch of software development before. However, I have got some previous experience with programming, which I first started to explore in secondary school and college, where I was taught how develop simple code in BASIC. At university I was introduced to Python and later to Java at which I probably feel most proficient in compared to all the languages previously mentioned. Through the use of Java I was able to grasp some Object-Oriented concepts which should be of help in the attempt of creating this medium sized piece of software.

Aim

The original objective as stated in the project idea part of my "Project Planning" report is as follows:

"The application will be a text and simple 2d graphic based android game developed using Android Studio as the IDE and using Java as the programming language. I aim to implement the agile and plan driven development techniques interchangeably to produce the software. The project will explore the Object-Oriented Development and Mobile computing aspects of the Computer Science field."

Although, this short description of the aim is bound to change as the project progresses. The prime objective is to develop a mobile game which will be engaging to the user using

graphics and exceptional gameplay. It must also encourage repeated play. To do achieve this objective research into what makes a successful game will have to be undertaken.

Report Structure

The research will explore different game genres and how successful they are. It will also analyse games that have already done well on the app stores and which features make them engaging to the players. Using this research, a decision on which genres the technical solution will target shall be made.

When the reader progresses through the literature review, and all the vital technical decisions are done, the descriptive application idea for the game awaits. Quickly followed by the functional and non-functional requirements.

Short after is the design part of the project in which the application is described using UML diagrams. Subsequently, the portrayal of the software implementation will be presented to illustrate how and with what thought process the software was developed.

Lastly, presents itself the evaluation of the project and personal suggestions for further development of the application.

Chapter 2: Literature Review

As mentioned in the introduction the Google Play Store and Apple App Store combined offer over 4 million applications to their users. Some require payment before use, some are free. The industry is enormous. However, only a few (relative to the total amount) applications are successful. One of the project aims is to analyse successful examples of games within different genres and which features of these games contribute to their success. The technical solution attempts to incorporate such features into the game. To achieve that it is essential to firstly review most downloaded games on the respective app stores, along with dissecting studies which explore this topic.

Endless runner genre

The first genre to investigate is the endless platform/runner games. It's a type of game in which the player gets placed into a world which automatically generates random obstacles that the player must avoid whilst progressing through the game by the character's movement forward. Usually, the player can collect bonuses along the way and the gameplay gets progressively harder usually by speeding up the pace of the character. Typically, the aim of the game is to achieve the highest possible score through collecting the most bonuses and surviving for the longest amount of time you can (Medeiros and Medeiros, 2014). Examples of games which are very popular within this genre are:

1. "Subway Surfers" which in September 2015 had over 1 billion downloads and 27 million daily active users across all platforms (Jordan,2015). In October 2018 the number of downloads exceeded 2.1 billion (Gumina, 2018).
2. "Flappy Bird" a phenomenon of a game which in January 2015 had been downloaded over 50 million times and was reportedly making its creator Dong Nguyen $50,000 a day, before he decided to withdraw the game from the app stores due to supposedly

being worried about the addictive effect it was having on its players (Stuart,2015).However, many people later re-made the title with various slight changes.

3. "Temple Run" and "Temple Run 2" of which the first one in 2014 exceeded 1 billion downloads, the combined play time from all users amounted to 216,018 years and 50 million digital meters have been travelled (Takahashi, 2014).

The three games detailed above all have many things in common. All of them are single-player where the primary objective is to beat your previous high score. Inferring from the statistics mentioned above, all of them are pretty addictive in their nature and one could assume that is a big contributor to why they're so popular. The gameplay is very simple, the player gets a single or very few ways of controlling the in-game character. In "Temple Run" and "Subway Surfers" you can swipe left and right to switch the lanes of running and swipe up to jump over or onto obstacles. In "Flappy Bird" the bird is constantly moving forward whilst simultaneously falling down, the only way to control the in-game character is to tap the screen every time you want the bird to "flap" its wings and therefore fly in the upwards direction. All the games randomly spawn obstacles for the player to avoid. "Temple Run" and "Subway Surfers" also spawn coin-like objects which increase the score when picked up. To diversify the gameplay there are powerup objects spawned which, when picked up by the character help them in progressing through the world for a certain amount of time, by shielding them from danger in some form. One could guess that since the in-game world is never exactly the same (in all the titles), the player gets better through repetitive play and therefore improving their understanding of the game mechanics which in turn allows them to increase their high score. Every time the high score is broken, the feeling for the player is very satisfying as it's an obvious indicator that their skill level of manipulating the mechanics has improved.

Strategy and Simulation genre

Although strategy and simulation are technically two separate genres, the examples of games presented further within the literature review contain aspects of both genres, hence they're described together.

The strategy genre requires the player to be thoughtful and planned with the use of in-game resources. Usually, the game provides the player with a start territory and some resources like money, building materials etc (Wolf, 2012a, p627). The objective is to use the provided resources to develop the territory e.g. by building better defences or improving resource production by purchasing upgrades. It could also include recruiting different types of armies to fight off enemies. Of course, this is just a brief generalisation of the genre which comes with all types of various playing forms and styles.

The simulation genre in general covers games which assign the player to be in control of a "dynamic system". A game covered by this genre resembles a simulation of real or fantasy life where the player controls what happens to the world, typically from a "outside" or "above" position. The player can manipulate different variables within the game which affect the progress of the in-game world (Wolf, 2012a, p581). Again, resource management to develop the in-game world is a common aspect of such games.

Examples of mobile games that would fall under both categories are:

1. “Clash of Clans” in 2018 had over 600 million downloads across all platforms and was earning about $1million dollars for Supercell (the developer) (Video Games Stats, 2019).
2. “FarmVille” and “FarmVille 2” in 2014 were able to attract over 400 million players (Zynga, 2014). This title made its first appearance on the games section of Facebook, however, as the popularity grew “Zynga” the developers decided to also release mobile versions.

“Clash of Clans” is a multiplayer game in which the player is in charge of a “clan” community. At the start the player receives a basic territory with a town hall. The currency within the game is gold with which the user can improve the territory’s defences like: wall, cannons, mortars etc. The other currency within the game is elixir and dark elixir (a better version of elixir), this is used to improve the offensive aspects of the game like army camps, barracks etc. It can also be used to train (and thus improve their statistics) the armies of barbarians, wizards (and others) at the players disposal. The community can then get involved in raiding computer generated villages and the online aspect of the game allows the player to raid territories owned by other players. The users can form clans which they develop together and then engage in “clan wars” with other groups fighting for each other’s resources. Every time a game process, like an upgrade of a certain defence, is initialised a timer is initialised to stop the player from developing the particular game object further until the time has passed. Each level the timer is longer. Furthermore, the game allows players to buy another currency called “gems”, using microtransactions. Such “gems” are used to shorten the timer so that the payer can develop the territory quicker. In essence, the objective of the game is to become the most respected and feared community possible.

“FarmVille” is again a multiplayer game where the user is in control of a farmer. At the start you receive a farm with a farmhouse, barn, well and some land to start growing your crops. The basic currency within the game is coins, however “keys” and “stamps” are also available. The latter have more benefits than coins as they can be used to shorten the time necessary to upgrade the farm’s facilities, buy special animals and unlock secret places within the in-game world. Players can plant various plant crops and harvest them when they have flowered or purchase animals like cows which produce milk. These can in turn be sold for coins which you use to develop the farm further. “FarmVille” incorporates a level system. Whenever tasks like building a Windmill or harvesting the crops are completed the user earns experience points which eventually increases the level of the farmer. The interface also provides “an open book” which contains objectives. When you complete the objectives, you receive rewards like the previously mentioned keys or stamps. Moreover, the game incorporates a social aspect using Facebook, it allows you to contact other farmers which reaps rewards through helping them or using them as “farmhands”, accelerating the development of both players’ farms. The overall goal of the game is to develop a farm with the highest production possible, level up and become a rich farmer. The motivational factor is introduced through the application of leader boards which make the game more competitive and therefore engaging.

Chapter 3: Application Idea

Game Context

It's the year 2029. The world has been invaded by abnormally intelligent aliens from a planet called "Carroto". The aliens believe that to save planet Earth from global warming which humans are causing, they need to collect all of Earth's carrot supplies because they need them to power atmosphere filters which the Carrotians developed to restore the Earth's atmosphere back to normal. To make this happen the Carrotians (Residents of the planet Carroto) force the world governments to collapse so they can implement their own rule over Earth. However, as with any drastic change on our planet there is a rebellion: Vegetarians. They believe that carrots are an essential part of their diet and they simply cannot give up (what they believe to be) the most important resource that the Earth has provided for humanity.

Because the Carrotian authorities made possession, growing and sale of carrots illegal the resource has become scarce, therefore, the monetary rewards from selling carrots have rapidly escalated, but so have the costs of running such a business. You are in control of a Vegetarian going by the name of Phil which has the ambition to be the biggest carrot distributer on this planet, the only obstacle? The Carrotian Police.

Game Description

At the start of the game the player gets 10 carrot seeds, access to an insecure allotment, and a £1000. One seed has the potential to flower into one carrot, however, it may not flower at all. The player can decide whether they want to harvest before the carrots release seeds which takes 1 second per one carrot or whether they want to collect after the seeds are released which takes 2 seconds per one carrot, however, if the flowering is unsuccessful the carrot cannot release seeds. Once carrots are harvested the player should be displayed a report of how many carrots flowered successfully. To increase the probability of successful flowering the player can invest in allotment upgrades. There are two ways of selling the carrots, you can sell them one at a time or a whole batch at once. If you sell them one at a time then the reward is £10 per one carrot, however, the risk of getting caught by the Carrotian Police is higher, it also takes 1 second to sell one carrot, and therefore, the more carrots you want to sell the longer you must wait. If you decide to sell the carrots all at once the risk of getting caught is smaller and you receive the money instantly, however the profit is £4 per one carrot. If the player gets caught selling carrots the game should enter chase mode. Chase mode will be a 2d game mode where the player is in control of an in-game character which is constantly running, as the player runs randomly placed platforms should be generated. In order to survive the player must jump onto the platforms and not fall off. There should be randomly generated spikes on some of the generated platforms. If the player touches the spikes they lose. There should also be randomly generated carrots for the player to pick up and add to the amount of they have. If the player falls of the screen they lose. The objective of this game mode is to survive as long as possible. The longer you survive the more experience you gain which eventually increases your level and allows you to purchase better allotment upgrades. The player also must make sure that they invest in securing their allotment so that the risk of getting caught decreases, once the player has enough money they should be able to buy a new, plain allotment without upgrades to increase the carrot production. The goal of the game is to achieve the highest level that you can.

The application will be developed in Unity3D and C# will be the programming language used as that is the language Unity operates in.

Functional Requirements

1. App should set up initial game values i.e. 10 seeds, £1000, plain, insecure allotment.
2. App should keep track of the game status, which includes the experience points, money in player's account, all the upgrades bought, risk of getting caught and save it.
3. App should be able to retrieve the game status at start-up and display the same state of the game that was present on the previous run.
4. The app should have a main screen which contains buttons to direct the player to all of the applications functionalities.
5. Allotment upgrade screen should display all allotment upgrades available, cost of each upgrade, and a description of the benefits the upgrade provides. Also, it should have a "Buy Seeds" option, which allows the player to purchase seeds.
6. "Plant" screen should have "Plant & Harvest Seeds", "Plant & Don't Harvest Seeds" options which act according to the description provided above.
7. "Sell" screen should have "Sell one at a time", "Sell whole batch at once" options which act according to the description provided above.
8. On the top of all screens within the app all relevant game status data should be displayed to the user
9. The app should have a transaction algorithm in place so that the right amount of money gets deducted once a player buys seeds, allotment upgrades, and the right response for the transaction should committed to the game status.
10. App should support time restrictions and calculate the right amount of time that the player must wait once a restriction is applied.
11. App should have an algorithm that calculates the risk of getting caught based on the allotment upgrades that have been bought by the player. The higher the risk the more often the player should be caught.
12. App should include a "chase" mode as explained in the game description.

Non-functional Requirements

1. When selling carrots, the app should check if the quantity of carrots to sell does not exceed the quantity that the player possesses in their game account. If it does it shouldn't allow the transaction to commit.
2. If the player chooses the "sell one at a time" option, the app should set a timer for sale completion which should be calculated by quantity of carrots*1 second. The game status should only commit when the timer is finished counting down. Risk of getting caught should be higher than the "sell all at once" option. If the player gets caught some money should be deducted from the account and the game should enter "chase" mode.
3. If the player chooses the "sell whole batch at once" option, the quantity of carrots*£4 should be immediately added to the players account. The risk of getting caught on this option should be smaller.

Chapter 4: Design

Use-Case Diagram

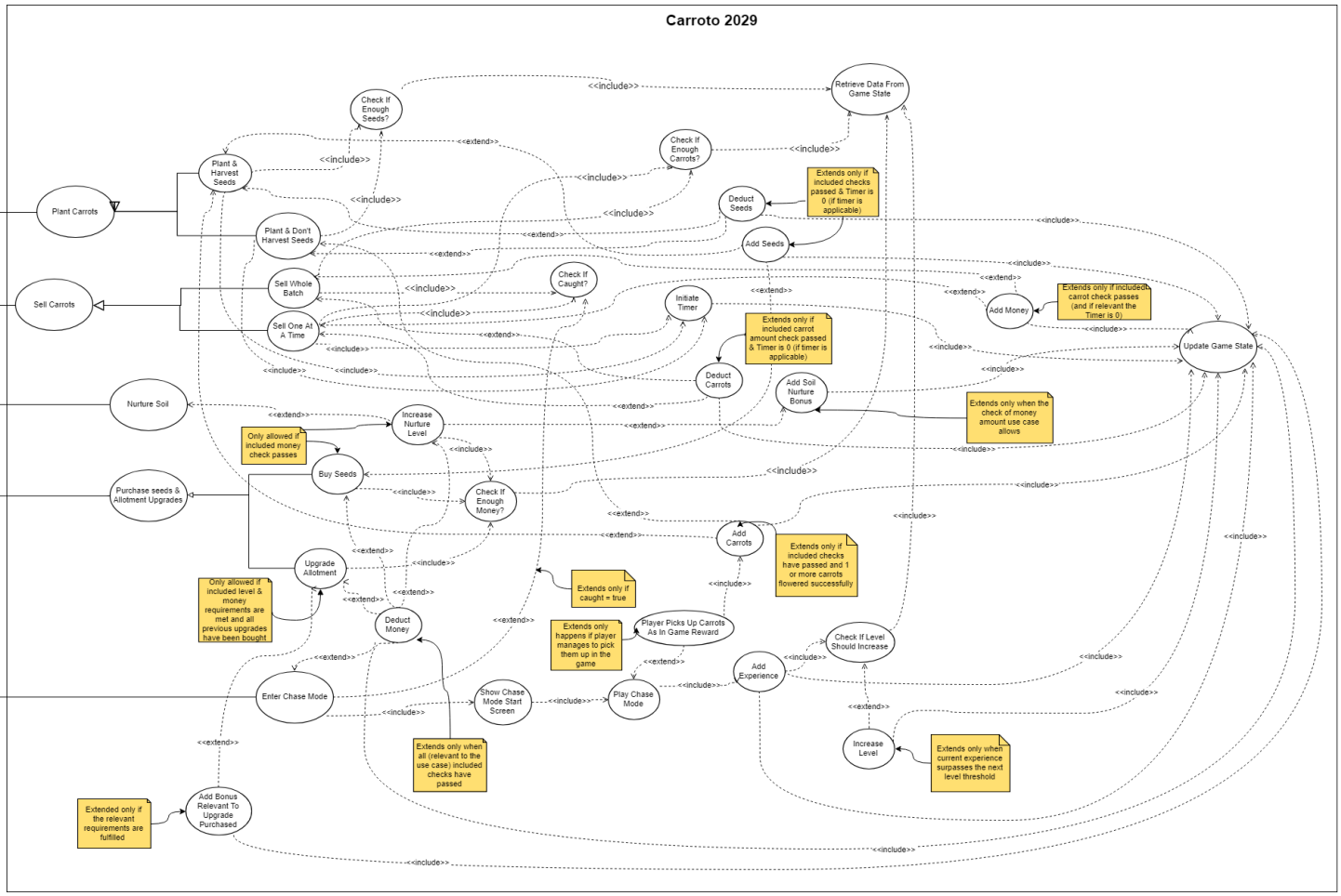

Figure 1. Use-case diagram for the Carroto 2029 application (Made in: draw.io)

The “Player” is the only actor that will interact with the application. It is also the primary actor for this use-case diagram because it initiates the use of the system. There are no secondary (reactionary) actors for this increment of the technical solutions, however, they could have to be added if proposed further improvements described near the end of this report were to be implemented.

The actor should be able to interact with the application through five main use-cases which are “Plant Carrots”, “Sell Carrots”, “Nurture Soil”, “Purchase seeds & Allotment Upgrades” and “Chase Mode”.

If the actor interacts with “Plant Carrots” they have the option of choosing between two child use-cases: “Plant & Harvest Seeds”, “Plant & Don’t Harvest Seeds”. Both have an include relationship with a use case which checks whether the amount of carrot seeds that is retrieved from the game state (through another include relationship) is enough to go through with the aforementioned use-cases. If there are enough seeds, then the included relationship for the “Initiate Timer” use-case starts processing, after the timer is done the extended relationship “Deduct Seeds” and “Add Carrots” get triggered (if the actor selects the “Plant & Harvest

Seeds" then the extended relationship "Add Seeds" becomes triggered as well). These in turn activate the extended relationship with the "Update Game Status" use case so that the changes are taken into account by the application.

When the actor initiates interaction with the "Sell Carrots" use-case there are two child use-cases: "Sell Whole Batch" and "Sell One at A Time" available for selection. Both have an include relationship with a check for whether the amount of carrots retrieved from the game state is enough to go through with the sell use-cases, and whether the player has been caught by the "Carrotian Police Departament". If they are caught the extended "Enter Chase Mode" use-case is activated which is explained in greater detail soon. If the carrot amount check is approved and "Sell One at A Time" has been chosen then the included relationship with "Initiate Timer" is processed, after it is done, the extended relationship with "Add Money" and "Deduct Carrots" is triggered, both have an include relationship with "Update Game Status" so that relevant changes to the game status are made. If the carrot amount check is approved, but this time the "Sell Whole Batch" is chosen, the same sequence happens but the included relationship with "Initialise Timer" does not exist, since there should be no delay when this action within the game is run.

The "Nurture Soil" use-case which the player can interact with has an extend relationship with a use-case called "Increase Nurture Level". This relationship is only triggered when the include relationship between "Increase Nurture Level" and "Check If Enough Money?" returns that there is enough money. This check also triggers the extended relationship between "Increase Nurture Level" and "Add Soil Nurture Bonus" to process which in turn activates the include relationship with "Update Game Status" so that the bonus is added to the game state and takes effect on the next planting action initialised by the user. "Check If Enough Money?" returning that there is enough money, also gives the extended relationship between "Increase Nurture Level" and "Deduct Money" the go ahead, which again updates the state of the game to incorporate the change in money amount.

Another use-case that can be interacted with by the player is "Purchase seeds & Allotment Upgrades". This parent use-case has two children: "Buy Seeds" and "Upgrade Allotment". Both children again have an include relationship with "Check If Enough Money?", when that check gives the green lights the "Buy Seeds" triggers the extend relationship with "Add Seeds" and "Deduct Money" which then utilise the include relationship with "Update Game State" so that changes are updated. When "Upgrade Allotment" is selected and all conditions (provided in the note attached to this use-case within **Figure 1.**) are met, the extended relationship with "Add Bonus Relevant to Upgrade Purchased" is triggered, this then activates an include relationship with "Update Game Status". This is done so that when an upgrade is purchased by the player the game state is promptly updated to make sure the application can add the relevant bonuses to the game behaviour.

The final use-case that the player can interact with is "Chase Mode", this can be initialised by the player, however, it is also worth to note that the extend relationship with "Check If Caught?" (which is triggered by "Sell Carrots") can also determine whether "Chase Mode" is initialised. When it is, then if the game state has enough money, "Deduct Money" is run, which as previously mentioned this deduction gets updated to the game state instantly. Then an include relationship takes the player to "Chase Mode Start Screen", when the player prompts the application to "Play Chase Mode" the relevant include relationship makes that

happen. If the “Player Picks Up Carrots as In Game Reward” whilst interacting with the chase mode, then the extend relationship attached to that use-case activates. This then adds carrots and updates the game state to consider the change in the amount of carrots possessed by the player (both are include type relationships). After playing chase mode the include relationship with “Add Experience” is triggered. This gets updated in the game state, if the experience surpassed the next level threshold then an extend relationship with “Increase Level” is triggered, the level gets increased by one and updated to the game state.

Class Diagram

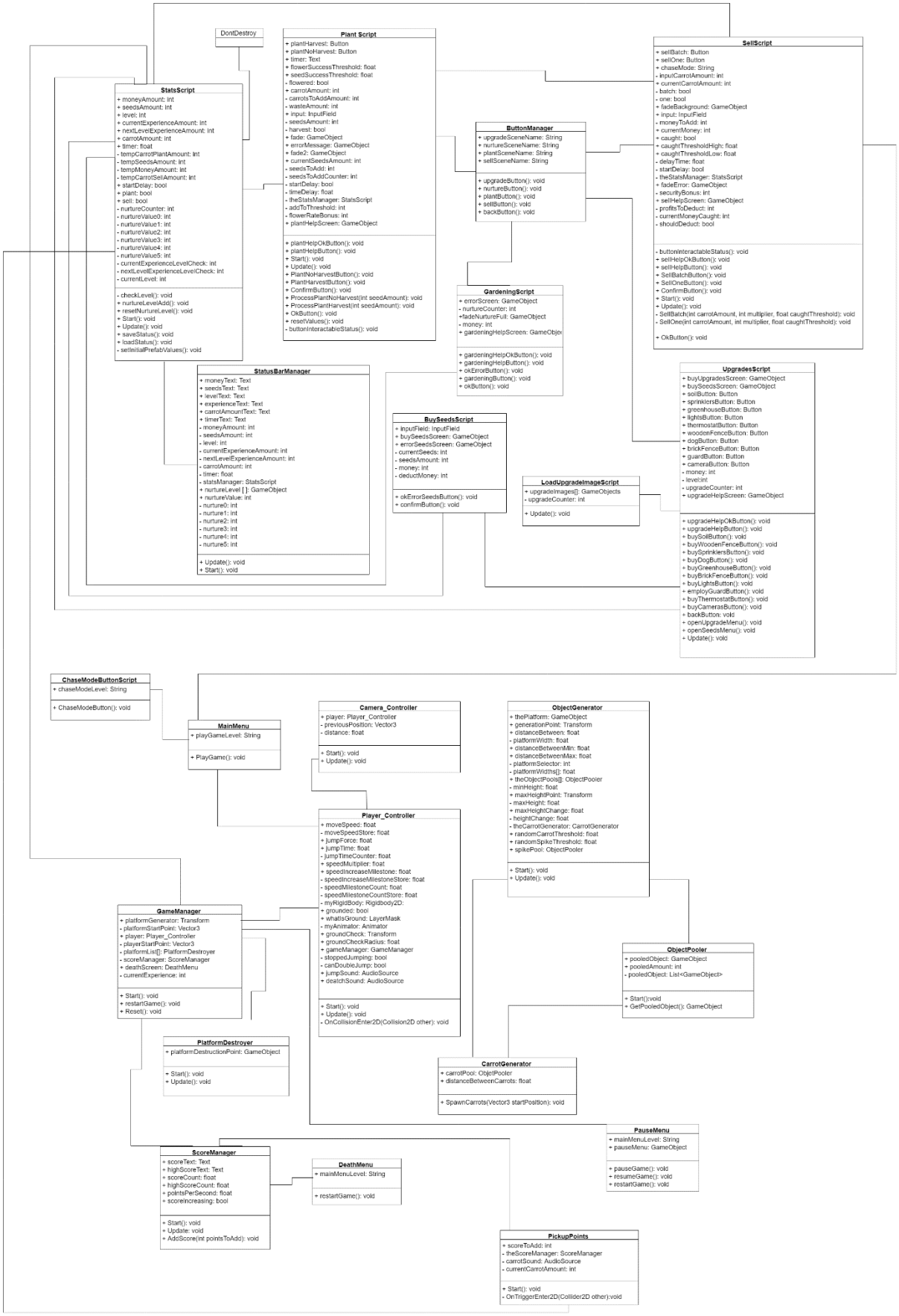

Figure 2. Class Diagram for Carroto 2029 (Made in: draw.io)

Designing the graphics

All the images used as graphics for this application were made with free web applications called: sketch.io and sketchup web.

Chapter 5: Software Implementation

Allotment Mode

GUI

When contemplating on where to start to develop the simulation/strategy part of the game, it was decided that the best starting point would be laying out the basic GUI components. This was done so that further programming can be done with structure, where every time programming would take place, it would be based on the current screen and GUI components present within the screen. “Allotment Mode” consists of five scenes: “NurtureScene”, “OverviewScene”, “PlantScene”, “SellScene”, “UpgradeScene”. The scenes include all Game Objects and GUI components necessary to process the functionality of “AllotmentMode”.

The best place to start was the “OverviewScene” which acts as a main menu for the game and has buttons which then branch out the program in different directions based on player interaction with the software. Based on prior planning and design it was known that there would be five main functionalities available to the user to manipulate and play the game. These are: planting carrots, selling carrots, purchasing seeds and upgrades to the allotment, nurturing soil and chase mode. Therefore, the “Overview Scene” had to be a screen which provides a way of navigating to those functionalities. Hence, “Buy_Button” was created to act as a way of accessing the “UpgradeScene”. The “Nurture_Soil_Button” was created to transfer the user to the “NurtureScene”. The “Plant_Button” was introduced to direct the player to the “PlantScene”. The “Sell_Button” was applied to take the user to the “SellScene”. Finally, the “ChaseMode_Button” allows the player to access “ChaseMode”, which is the endless runner part of the game and will be detailed further later within the chapter.

All scenes within this mode also include two image Game Objects: “statusBar” and “levelBar”. The “statusBar” has four text UI GameObjects as its children: “moneyText”, “seedsText”, “carrotAmountText”, “timerText”. The “levelBar” includes two text UI Game Objects as its children: “levelText”, “experienceText”. All the text UI Game Objects act as containers which can be manipulated by code to display the relevant game statistics to the user. The two image Game Objects act as containers for the text boxes and contain source image files (stored within the Unity project’s Art folder). It’s important to note that the level bar also contains six image Game Objects called “nurtureLevelImage”, “nurtureLevelImage(1)” and so on. These are by default set to be inactive and only become

active in certain circumstances defined by the code. These images are set active consecutively to act as an indicator of the level of soil nurture. The exact way of how this is implemented will be described further on within this chapter. Another noteworthy point is that every scene within this mode includes Unity generated "MainCamera" Game Object in their respective hierarchies. All such Game Objects have eleven image Game Objects attached as its children. The source image files for these Game Objects are different dependent on the Scene (all stored within the Unity project's Art folder). They are all set as inactive by default and become active only when instructed to by the code. Their purpose is to display the background image for the scene relevant to the level of upgrades purchased by the player, every time a player upgrades the allotment the image changes to reflect these changes. Once again, the full extent on how this process operates will be explained later in the chapter.

"PlantScene" along with the GUI components that all scenes encompass, consists of four buttons that are by default set to be active within the hierarchy. The purpose of "Plant_No_Harvest_Button" is to activate the "FadeBackground" Game Object and indicate to the compiler that the user does not want to harvest seeds when initiating the plant process. "Plant_And_Harvest_Button" has the same purpose but instead indicates that the player wants to harvest seeds when commencing the plant process. "FadeBackground" by default is inactive, it includes three Game Objects. A UI input field called "InputField" which acts as a way of getting user input on how many seeds they wish to plant. A "confirmButton" which is supposed to be pressed when the decision on the amount is made. Lastly, a UI text field, the body of the text field explains the what data the input field is expecting. "BackButton" is a component which directs the player back to the "OverviewScene". This type of button is present in all the scenes that form this mode apart from the "OverviewScene", so is the "plantHelpButton" (name changes dependent on the scene). The purpose of the latter is to set active the "plantHelpScreen" (name changes dependent on the scene) Game Object, again present in all scenes but "OverviewScene". These screens all consist of a text filed child Game Object which displays a description of what exactly is happening when the user interacts with each respective scene and the consequences that apply to the game state through the interaction. Another child of the help screens is an "OkButton" which deactivates the help screen and returns the user to the scene that was present prior. There is one more Game Object which is present at every scene that requires the player to sacrifice any type of resource to gain another (in this scene called "FadeBackgroundError"). Again, its set as inactive by default and only gets shown to the user if they don't have enough of the resource that is necessary to complete the relevant process. They all have two children. An error message text field which act as a description of why the process couldn't be initiated and a button allows the player to return to the scene that was interacted with prior to the error.

Next was the "SellScene". This scene consists of all mutual GUI Game Objects described earlier, plus two buttons called "Sell_All_Button" and "Sell_One_Button". Both, when pressed display a screen the same as "FadeBackground" mentioned earlier, the only difference is the description of what type of user input the input field is expecting. "Sell_All" indicates to the code that the process of selling the whole batch of carrots at once is to be initialised, whereas, "Sell_One" specifies to the code that the selling carrots one by one process is to be initiated. The details of how they differ in effect will be explained further on.

The "NurtureScene", again consists of all mutual GUI Game Objects described earlier. The difference is one button called "Gardening_Button", which tells the code to initiate the

process of adding to the soil nurture level at the cost of some money available to the player. Once again, the process will be depicted later.

The final scene in this mode is “UpgradeScene”. In addition to all the mutual GUI components mentioned prior, two buttons: “buySeedsButton” and “upgradeAllotmentButton” are on display to the user. The first activates a “buySeedsScreen” which is identical in application as the “FadeBackground” screen in “PlantScene” with a different description of what the program is asking for from the user. This button lets the script know that the buy seeds process is about to be called. The second button serves the purpose of displaying the “buyUpgradesScreen” which is essentially a menu for purchasing allotment upgrades like better soil etc. Within that screen there’s plenty of text fields describing to the user what can be bought, how much it costs, what benefits it will apply to the game and the cost of each upgrade. Available to the user are buttons which indicate to the script which upgrade the user is buying, so that relevant changes can be made. By default, all the buttons’ intractability is set to false, and only turns to true when particular game state conditions are met, which in turn allows the player to purchase the upgrade. The conditions are code defined and explained later.

Scripting

Button Manager

The first C# script to be developed was the “ButtonManager” script. This script is very simple. It has five public String variables and five public void methods. All each method does is use `Application.LoadLevel(<SceneName>);` to tell the application to load relevant scene defined by the previously mentioned string variables. The content of the string is defined within the Unity editor. This script is assigned as a component of the “ButtonManager” Game Object which is present at every Scene within this mode. Each method is assigned as a trigger to the relevant button’s OnClick() method through the Unity editor. The goal that this script achieves is to manage transitions between scenes within this mode.

Game Status, Status and Level Bars and Soil Nurture Management

The purpose of the “StatusBarManager” script is to manage values relevant to both the “statusBar” and “levelBar” and constantly update them to account for any changes of game status. This class imports the `UnityEngine.UI;` library in order to define public text variables which are then assigned values through the editor. These values are all text field Game Objects, the children of “statusBar” and “levelBar”. It also defines private variables which are then used to manipulate the values within those text field Game Objects. The script also takes advantage of benefits available through the use of object-oriented aspects of C#. An object of the “StatsScript” class is created and within the Start() function it is linked to the variable by using `FindObjectOfType<StatsScript>();`. It also defines a public array of GameObjects called nurtureLevel which has its values assigned to it in the editor. These values are the image Game Objects called “nurtureLevelImage”, “nurtureLevelImage (1)” and so on. The script also defines 6 integer values called “nurture0” “nurture1” and so on.

These are later used to check which elements of the “nurureLevel” array should be set to active and which ones to inactive.

In the Start() method first takes place the link of the “statsManager” variable to the object of the “StatsScript” as detailed above. This is done so that values within that script can be manipulated through this one.

In the Unity provided Update() method, this method is run every frame. The first statement to take notice of is the `statsManager.loadStatus();` statement which makes a method call to the loadStatus() method within the “StatsScript”. This is necessary so that every time any value is changed the status and level bars are both instantly updated. The code then assigns all values from the stats script that are relevant in this one to the private fields responsible for manipulating the game status values, mentioned prior. These values are then assigned to the Text field variables with the .ToString() extension added to them to avoid type mismatch errors. This causes the game state data to be displayed directly to the user. The program then makes a call to the `statsManager.nurtureLevelAdd();` method which checks the soil nurture level value stored as a player preference called “NurtureCounter” every time the nurture counter value is increased the respective “Nurture0” or “Nurture1” (and so on) is set to 1. This in turn causes the next image within the “nurtureLevel” array is set to active, until it reaches the last one. The “NurtureCounter” player preference is manipulated by the GardeningScript by using the gardeningButton() method the “NurtureCounter” is assigned to a private integer variable called nurtureCounter like so:
`nurtureCounter = PlayerPrefs.GetInt("NurtureCounter");` this value gets increased every time the user presses the Gardening_Button until the counter reaches 5 so that it does not go over the array size of nurtureLevel as they are directly linked. Every time the nurtureCounter value is incremented £10 is taken away from the players resources. If the player runs out of money an error screen is displayed. All this is done by using if else statements.

Focusing back on the “StatusBarManager” script, after the nurtureLevelAdd() method is called the script retrieves the “Nurture0”, “Nurture1” (and so on until “Nurture5”) player preferences and assigns them to the private integer variables called “nurture0”, “nurture1” and so on. Then a series of if statements checks whether the value of the private integer variables is 0 or 1. If it is 0 then the respective element of the nurtureLevel array is deactivated and therefore not displayed to the user as part of the soil nurture level on the level bar. If its 1 it is set to active and the respective image in the nurtureLevel array is displayed to the user as part of the soil nurture level on the level bar.

The “StatsScript” also has a method by the name “resetNurtureLevel” which sets “NurtureCounter”, “Nurture0”, “Nurture1”, “Nurture2”, “Nurture3”, “Nurture4”, and “Nurture5” back to 0 and therefore deactivates all the images from the nurtureLevel array. This method is called within the method attached to the “confirm” button in the “PlantScene” so that every time the button is pressed and therefore a plant process is initiated the soil nurture level is reset back to 0, adhering to the requirements set out before the coding started. The “NurtureCounter” player preference is also used within the code for that button to add the relevant bonus of chance for the carrot to flower that comes as an effect of the soil nurture level.

The “StatsScript” method called “saveStatus()” is then called to save any changes made to the game status in this frame. It is important that all of this is placed within the Update() method as all of this information needs to be constantly updated and displayerd back to the user.

Making sure “StatsScript” doesn’t get automatically destroyed when scenes are switched

Since some of the functionality within this application requires a timer to count down and apply changes to values after the timer hits 0, it was problematic when during development it was discovered that the game status values which were to be applied after the timer was done if the player switched scenes in the meantime. Therefore, some research needed to be done to implement a way of not destroying the “StatsScript” between scene transitions. That’s when the “DontDestroy” script was created. This script simply uses the Unity provided Awake() function. In that function an array of Game Objects called “objects” is assigned the “StatsManager” (which has the “StatsScript” added as a component) Game Object by using its tag “stats”. Even though its only one object the array was necessary as when scene transitions would happen the “StatsManager” game object would be created again once the Scene was switched back to the “OverviewScene”. This could prove to be problematic so there’s an if statement which checks if the array length is more than one. If it is then the new instance of the object is destroyed. After the if statement the `DontDestroyOnLoad(this.gameObject);` statement is made which ensures that the “StatsManager” does not get destroyed.

It is important to note that this code is based on a YouTube tutorial video. Source: https://www.youtube.com/watch?v=JKoBWBXVvKY&t=2s

Processing Carrot Planting

The “PlantScript” takes care of almost all the carrot planting functionality within the application. There are two options for planting carrots. Planting and harvesting seeds from the carrots that flower successfully, and planting but not harvesting these seeds. The main difference between the two is that the first’s completion time is longer than the second’s completion time. Another difference is that the process which includes harvesting seeds must add not only the flowered carrots to the game state but also the seeds that are collected after they have flowered. To imitate this behaviour two methods were developed: “ProccessPlantHarvest(int seedAmount)” and “ProcessPlantNoHarvest(int seedAmount)”. Both methods take “seedAmount” as a parameter. This is the amount of seeds that the user wants to plant which is passed through by parsing the content of the seedsAmount Input Field. As mentioned in the GUI description the “PlantScene”, has two buttons: “Plant_No_Harvest_Button”, “Plant_And_Harvest_Button”. These buttons have methods within the “PlantScript” attached to them which run when their OnClick() method is run, these methods just set the “harvest” Boolean value to true or false dependent on which one is chosen. Both methods also make the “fade” GameObject active. “Fade” is the screen which asks the user for input and displays a confirm button when they have decided on the amount of seeds they want to plant. When the user presses the confirm button the “ConfirmButton()” method is run. This method firstly retrieves the “NurtureCounter” value from the player preferences and multiplies it by 5 to then add this value to “flowerSuccessThreshold” and

“seedSuccessThreshold”. This is important as the soil nurture bonus needs to be added to these thresholds so that the probability of carrots flowering and the probability for how many seeds that these carrots will release needs to be adjusted based on the soil nurture bonus. So if the soil nurture level is 0 no bonus will be added to the probabilities, but if the level is 5 (maximum) then the probabilities will increase by 25%. The code then checks whether the amount of seeds that the user input is not lower than the amount stored in the game state. If it is lower an error screen is displayed with a message as to what happened. If it is higher then the code checks whether harvest is true or false, When it’s true the “timeDelay” value is set to two multiplied by the amount of seeds the user wants to plant, the “startDelay” boolean value is set to true, and the “ProcessPlantHarvest” method is called with the user input passed through as the parameter. In the case that “harvest” is false, the “timeDelay” value is set to one multiplied by the amount of seeds the user wishes to plant, “startDelay” is once again set to true and this time the “ProcessPlantNoHarvest” method is called (with the user input passed through as a parameter again).

“ProcessPlantNoHarvest” method contains a for loop. This loop’s index value starts at 0 incrementing by one every time it loops, and it loops whilst the index value is below the amount of seeds that the user input. Every increment the code generates a random float value between 0 and 100. If that value is below the total of “flowerSuccessThreshold” and “flowerRateBonus” (a value which increases as the player buys in-game upgrades, explained later in greater detail) then “flowered” is set to true. If it’s above “flowered” is set to false. Then (still within the for loop), the code checks whether the “flowered” variable is true or false. If it’s true, then “carrotsToAddAmount” increments by one, if it’s false, then “wasteAmount” increments by one. When the loop is complete the “currentSeedsAmount” value has the amount of seeds the user planted deducted from it, so that the amount of seeds used for this operation are deducted from the game state. Then the “carrotsToAddAmount” is added to “carrotAmount” so the amount of successfully flowered carrots is added to the game state. The code then manipulates three variables from “StatsScript”. First it sets “plant” to true, then it assigns “timer” the content of “timeDelay” set within the “ConfirmButton()” method, and lastly it assigns “StatsScript’s” “startDelay” the content of “Plant Script’s” “startDelay”. It then goes on to set the “TempCarrotPlantAmount” player preference to the value of “carrotAmount” and the “TempSeedsAmount” player preference to the value of “currentSeedsAmount”. This is important as these temporary values will be assigned to “Carrots” and “Seeds” player preferences (which are responsible for storing the game state values of the amount of carrots and seeds the user has at their disposal) after the timer counts down from “timeDelay” to 0.

The programming for the incorporation of the timer had to be done within “StatsScript” since as explained earlier it does not get destroyed when the user switches scenes. So, when the “startDelay” value within this script changes to true as a result of the “startDelay” value changing to true within “PlantScript” the delay is initiated within the “Update()” function. Every frame the time it took to process that frame is deducted from the “timer” value (which holds “PlantScript’s” “timeDelay” value). The “Timer” player preference is set to “timer” so that the user can see the amount of time left on the timer. The player preference values: “sellBatchActive”, “sellOneActive”, “plantHarvestActive” and “plantNoHarvestActive” are set to 0. This is done as both “SellScript” and “PlantScript” have a method called “buttonInteractableStatus()” which is within their respective “Update()” methods and

therefore called every frame. This method checks if the player preference values just mentioned are set to 0 or 1. If it's 0 then the "SellScene" and "PlantScene" buttons' intractability is set to false, if it's 1 it's set to true. This makes sure the player cannot use the functionality of those scenes until the timer is done, as when the timer hits 0 the player preference values just mentioned get set back to 1, therefore making the buttons interactable again. When the "StatsScript" timer hits 0 the code check whether "plant" is true (this is set to true by the "ProcessPlantNoHarvest" or "ProcessPlantHarvest" methods when they are called). If it is then the temporary player preference values (which are storing the new amount of carrots and seeds that should be in the game state after the plant process is done) are assigned to the "Carrots" and "Seeds" player preference values, so that the game state is updated but only after the delay. The temporary player preferences are then set back to 0 so that they can be used again when the planting of carrots process is initiated again, and "plant" and "startDelay" are set to false so that the code exits out of the if statements.

The "ProcessPlantHarvest" method is very similar to "ProccessPlantNoHarvest" the only difference is that when "flowered" is true a random value between 0 and 100 is generated and if that value is below the "seedSuccessThreshold" value then between one to three (randomly selected) seeds are added to "seedsToAddCounter". When the for loop is complete, the "seedsToAddCounter" is added to "currentSeedsAmount", which makes sure that the seeds harvested from the successfully flowered carrots are also added to the game state.

Brief Description of Other Allotment Mode Scripts

Upgrades Script

This script makes sure that when the player meets the requirements for an allotment upgrade the button for the upgrade is set to be interactable. When the button to purchase a particular upgrade is pressed the relevant game status changes are made, the button is set to inactive (so that the player cannot do the transaction twice), and the flowering probability or allotment security bonuses that the purchased upgrade promises to the player is applied.

Sell Script

The main purpose of "SellScript" is to manage the different selling operations available to the player within the "SellScene". It makes sure that when the "Sell One at a Time" button is pressed the profit per one carrot sold is £10, a timer is initialised and only after the timer is done the game state is updated, and that the likelihood of the player being caught and therefore transferred to "Chase Mode" is higher. When the "Sell Whole Batch at Once" button is pressed the profits are set to £4 per one carrot, there is no delay and the rewards get updated to the game state immediately and the probability of the player being caught is lower. This script is very similar to the "PlantScript"

Load Upgrade Image Script

There are different images which are to be displayed as the background whenever an allotment upgrade is purchased, this is done to visibly show the user their upgrades. This script constantly (every frame) loops through an array which contains the various images. When the allotment upgrade is purchased by the player the "UpgradeLevel" player preference changes. Based on what integer value is stored in that player preference the relevant image within the array is set to be active within the scene, and all others are set to be inactive, and therefore not visible to the user.

Buy Seeds Script

This script provides the instructions necessary to enable the user to buy seeds for £1 per seed. The script makes sure that the correct amount of money and seeds is updated to the game state when the operation is complete.

Chase Mode Button Script

This script is attached to the "Chase Mode" button which is present in the "OverviewScene". This button exists to allow the player to enter "Chase Mode" when the button is clicked. The only thing it instructs is that when this button is pressed the application loads the "ChaseStartMenu" scene.

Chase Mode

Ground Detection

Ground detection is made simple with Unity. All that is done is the "Player" GameObject and any platform that is generated within this mode have a Box Collider 2D component attached to them. This way Unity automatically recognises that if the player's box collider touches a platform's box collider it should not be able to just "fall through".

The Player

"Player_Controller" script controls what happens to the in-game character that the player is in control of. Most of the core functionality that this script provides is coded within the "Update()" method. First the boolean variable grounded is assigned a value provided by the "Physics2D.OverlapCircle()" method. This method uses the position of the "groundCheck" Game Object (attached to the bottom of the Player's Rigidbody2D), the radius of the "groundCheck" GameObject and the "whatIsGround" layer mask (all platforms that are generated within this mode have a "ground" layer mask) and returns true if the groundCheck overlaps any layer mask called "ground". The code then sets the velocity attribute of the Rigidbody2D component attached to the script to move along the x-axis by moveSpeed every frame and leaves the y value the same as it is. Then an if statement checks if the jump key has been pressed, if it has it checks whether grounded is true, if it is then the y value of the velocity attribute of the attached Rigidbody2D is set to "jumpForce" which makes the in-game character move up the y axis by the "jumpForce" amount. The "stoppedJumping" boolean value is set to false (its true by default) and the "jumpSound" audio source is played. The code then has an if statement which checks if "grounded" is false and "canDoubleJump" is true. If that's the case "jumpForce" is applied again to the y coordinate of the player's Ridigbody2D velocity attribute. This creates a double jump effect when the player presses the jump key twice. This if statement then resets the "jumpTimeCounter" back to its original value, sets "stoppedJumping" and "canDoubleJump" to false, and plays the "jumpSound" audio source.

The code then checks if the jump key has been held down. If it has then as long as "jumpTimeCounter" is above 0, jumpForce is added to the y coordinate of the velocity attribute within the attached Rigidbody2D. To make sure the jumping cannot be never ending the time it took to process the last frame is deducted from "jumpTimeCounter" every frame, so that when it is below 0 the compiler exits this if statement. There is another if statement

which ensures that when the jump key is let go the “jumpTimerCounter” is set to 0 and “stoppedJumping” is set to true, so that “jumpForce” cannot be added further.

The final if statement resets the “jumpTimeCounter” back to the original value and sets the “canDoubleJump” boolean to true when the “grounded” boolean is true. This makes sure that every time the player’s box collider touches a platform the player can jump again.

“Player_Controller” also has a method called “OnCollsionEnter2D” which makes sure that when the player collides with a game object with the “killbox” tag the game is reset and restarted. That game object is placed just below the camera and moves with the player so that when the player falls of the screen the two always collide and therefore trigger this method to run.

Object Pooling: Design Pattern

In the “Start()” method of “ObjectPooler” script a new list of game objects is created. Then a for loop which starts the index at 0 increments it by one every time it loops runs while the index is below the “pooledAmount” value. Every time it loops the “pooledObject” gets instantiated, set to inactive within the hierarchy and added to the list. The “pooledObject” and “pooledAmount” are assigned within the Unity editor. This is done so that the game objects which need to be automatically generated (e.g. platforms) can be added to the “pooledObjects” list.

There is also a method called “GetPooledObject()” which returns a game object when called. This method is there so that once the “pooledObjects” list is set up the method can return the first inactive game object that it finds within the list and if there is none, then add one to the list and then return it.

The “ObjectPooler” script is attached to any game object that needs to be generated as the player moves. The prefab for that game object is passed as “pooledObject” and the number of objects that the list should contain is set within the editor. An object pool can then be passed to another script to make use of its functionality, like the “CarrotGenerator” script. This script defines a public variable called “carrotPool” of the “ObjectPooler” type. Within the editor the variable is assigned the “CarrotPool” game object, this game object has the “ObjectPooler” script attached as a component and the carrot prefab passed as the “pooledObject” variable, with the “pooledAmount” variable set to 15. The “SpawnCarrots(Vector3 startPosition)” method in the “CarrotGenerator” script makes a call to the “GetPooledObject()” method from “ObjectPooler”, and an inactive carrot game object is returned. The “CarrotGenerator” script then sets the position of the carrot game objects to the one that is passed through when the “SpawnCarrots” method is called and sets it active within the scene. When the player collects the carrot by running into it, the carrot is set back to inactive. The call to “SpawnCarrots” is made within the “Update()” method of the “ObjectGenerator” script. It is only called when a random number between 0 and 100 is below the “randomCarrotThreshold” value. The position of where the carrot is placed is the same as the most recently generated platform so that the carrots spawn only on platforms.

It is important to note that this concept was developed to resemble the Object Pool Design pattern described in Figure 3. This design pattern makes the constant creation of randomly generated game objects more memory efficient. If the generation of random game objects was programmed in a way where a new game object is instantiated every time, quickly the

memory of the device on which the application is running would get used up until the application becomes either very slow or shuts down because of a lack of memory. With this design pattern, the game objects get constantly reused, and only a defined amount of them will be created.

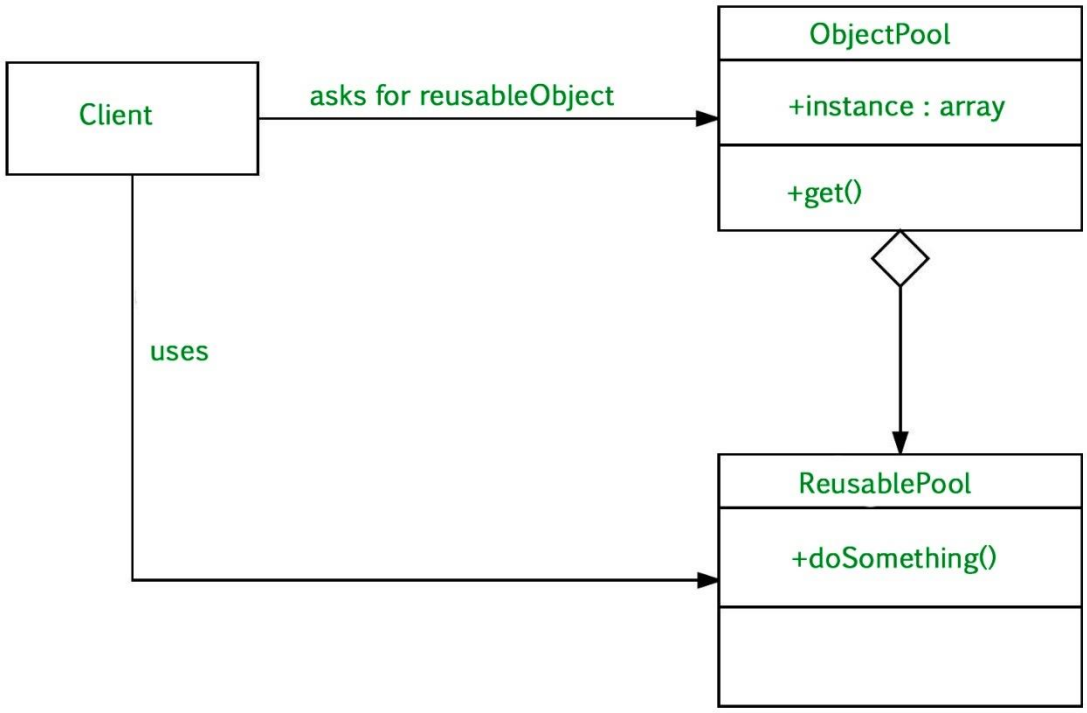

Figure 3. Object Pool UML Diagram (source: https://www.geeksforgeeks.org/object-pool-design-pattern/**)**

Brief Description of Other Chase Mode Scripts

Object Generator

The "ObjectGenerator" script is responsible for generating all game objects which are to be randomly and automatically displayed on the screen. Its "Update()" method is where all of the generation of platforms happens, including which platform type should be generated, and at what position within the scene. It is also where the randomly placed spikes and carrots are generated.

Camera_Controller

This script is responsible for moving the camera simultaneously with the running of the in-game character. It is added as a component of the "MainCamera" game object which is automatically generated by Unity.

Score Manager

"ScoreManager" is responsible for the score system that is implemented in "Chase Mode". Every second that the player survives, it adds a certain amount of points to the score. If the score is higher than the high score, then that score becomes the high score.

Platform Destroyer

This script's sole purpose is to deactivate platforms that are behind the "platformDestructionPoint". This is an invisible to the eye game object that is some distance behind the "Player" game object. Every time the destruction point passes a platform, that platform is de-activated from the scene, and can be reused by the platform object pool.

Pickup Points

"PickupPoints" makes sure that when the player runs into a randomly generated carrot the score is increased, a sound is played, the amount of carrots in Allotment Mode's game state is increased, and the carrot game object that has been collided with is deactivated.

Pause Menu

The purpose of this script is to provide the code for the "PauseMenu" screen. This screen gets displayed when the pause button is pressed. The game is stopped, and a resume game and restart game buttons are displayed. The resume button just deactivates the "PauseMenu" screen and starts the game again. The restart button resets the game and takes the player back to the "OverviewScene" in Allotment Mode.

Main Menu

The "ChaseStartMenu" scene is the one that is first displayed to the user when the application enters "Chase Mode". This scene provides the user with a description of the mode's rules and instructions on how to play. It also provides a button which allows the user to play the game and therefore make the application switch to the "ChaseMode" scene, and therefore start the endless runner mode. The "MainMenu" script's sole purpose is to provide the code for the "Play Game" button, this code just prompts the application to load the "ChaseMode" scene.

Game Manager

The "GameManager" script contains two methods: "restartGame()" and "Reset()". The first is called when the player dies. It makes sure that the score stops increasing, that the score achieved in "Chase Mode" is added as experience in "Allotment Mode", that the player game object is deactivated and the "deathScreen" displayed to the user. The second is called when the program wants to reset "Chase Mode". It deactivates the "deathScreen", makes all game objects with the "PlatformDestroyer" script attached to them inactive, resets the start position for the "Player" and "PlatformGenerator" back to the original value. Then it sets the player back to active, resets the score back to 0 and makes sure the score starts increasing the again.

Death Menu

After the player falls of the screen and dies or runs into spikes, the application displays a death screen. This screen informs the player that they have died and has a button which resets the game and then takes the player back to the "OverviewScene" in "Allotment Mode". The "DeathMenu" script is where the instructions for that button's functionality are placed.

Chapter 6: Evaluation

Report Writing

If there was another attempt at conducting this project. The main change that could be suggested for the process of writing the report is leaving more time for the report to be constructed. The mistake made was allocating a much larger amount of time to the development of the technical solution, and not leaving enough time to write the report. Because of that the report is not to the standard that it would have been had enough time been spared for writing it.

Suggestion of Further Improvements to Technical Solution

There are many ways that the technical solution could be improved further. First and foremost, there should be two timers. One responsible for dealing with the time it takes for planting carrots, the other to deal with the time it takes to sell the carrots. The code could be amended so that it allows the player to plant and sell simultaneously. Another suggestion to further evolve the timer could be to make it so that when the game is shut down and then returned to the timer should calculate the amount of time the player was away for and adjust the timer accordingly. This could be applied by making use of the OnDestroy() and OnAwake() methods, available within the Unity library. The program could save the real time just before quitting the game and store it as a player preference. When the program starts again the real time could once again be retrieved and the difference between the two values calculated.

Along with the real time being stored before quitting the game as mentioned above, all new values that were to be changed when the timer hit 0 would have to be stored temporarily, most likely by making use of player preferences again since they don't erase when the application quits. When the application starts again these values should be retrieved and applied accordingly.

Moreover, there could be a report displayed to the user after each plant or sell sequence. This was planned in the initial objectives but wasn't implemented due to time constraints. The report would contain basic information on the amounts of carrots flowered successfully, seeds wasted due to not flowering, how many seeds were used in total, and (if the harvest seeds option was chosen by the player) how many seeds were harvested. It could also include the amount of time it took for the process to complete.

As for the report displayed after the selling sequence, that could include information about the profits made from the selling, how many carrots were sold, (if the player chose the sell one at a time option) how much time it took for the sell process to complete. Most importantly it should display whether the player was caught by the Carrotian Police Department. If they were it should display all the resources that the police seized. Such reports could be stored within a "mail/report box" so that the user can analyse them it they want to but don't have to.

Another improvement could be to allow the player to keep buying plain new allotments which they could start developing from scratch, once they had enough resources to do so. This would make the player's engagement with the game longer.

After all the above changes were applied, online connectivity could introduce. This would provide the game with lots of new possibilities to expand. For example, players could employ people to defend your allotment and attack other ones in order to get their resources. It would also provide a platform for an online leader board which could be connected to a social media account in order to compete with your friends, and maybe create tasks that friends could do in co-operation. It would also encourage players to keep playing so that they climb the world leader boards in the hope of becoming the top carrot distributor in the Carroto 2029 world.

Bibliography

References

Gumina, S. (2018). *Subway Surfers Experience to reach Extreme Heights with SYBO TV*. [Pdf] Available at: https://sybogames.s1.umbraco.io/media/21425/sybo-tv-final.pdf [Accessed 20 Feb. 2019].

Iivonen, J. and Liu, Y. (2014). What Lead to the Successful Mobile Phone Game? – Story of birds who cannot fly but they have enough angriness. *Journal on Innovation and Sustainability. RISUS ISSN 2179-3565*, 5(1), p.71.

Jordan, J. (2015). *Subway Surfers breaks 1 billion downloads barrier, boasting 27 million daily active players*. [online] pocketgamer.biz. Available at: https://www.pocketgamer.biz/interview/62038/subway-surfers-does-1-billion-downloads-boasts-27-million-daily-active-players/ [Accessed 20 Feb. 2019].

Medeiros, R. and Medeiros, T. (2014). Procedural Level Balancing in Runner Games. *2014 Brazilian Symposium on Computer Games and Digital Entertainment*.

Moreira, Á., Filho, V. and Ramalho, G. (2014). Understanding mobile game success: a study of features related to acquisition, retention and monetization. *SBC Journal on Interactive Systems*, 5(2).

Richter, F. (2018). *Infographic: The App Gold Rush*. [online] Statista Infographics. Available at: https://www.statista.com/chart/9671/developer-earnings-apple-app-store/). [Accessed 9 Feb. 2019].

Saifi, R. (2017). *The 2017 Mobile App Market: Statistics, Trends, and Analysis*. [online] Business 2 Community. Available at: https://www.business2community.com/mobile-apps/2017-mobile-app-market-statistics-trends-analysis-01750346#SurHlwaeRLfIotS2.97). [Accessed 9 Feb. 2019].

Stuart, K. (2015). *Flappy Bird lands on arcade machine*. [online] The Guardian. Available at: https://www.theguardian.com/technology/2015/jan/12/flappy-bird-arcade-machine [Accessed 20 Feb. 2019].

Takahashi, D. (2014). *Even with a Temple Run empire, Imangi Studios wants to stay indie at heart (interview)*. [online] VentureBeat. Available at: https://venturebeat.com/2014/06/04/even-with-a-temple-run-empire-imangi-studios-wants-to-stay-indie-at-heart-interview/ [Accessed 20 Feb. 2019].

Video Games Stats. (2019). *Interesting Clash of Clans Statistics and Facts*. [online] Available at: https://videogamesstats.com/clash-of-clans-stats-facts/ [Accessed 3 Mar. 2019].

Wolf, M. (2012a). *Encyclopedia of video games: the culture, technology, and art of gaming*. 1st ed. Santa Barbara, CA [etc.]: Greenwood Press, p.627.

Wolf, M. (2012b). *Encyclopedia of video games: the culture, technology, and art of gaming*. 1st ed. Santa Barbara, CA [etc.]: Greenwood Press, p.581.