****

*Andrew Davis*

Evaluation Report

# Changes from the Plan

I am very happy with the overall outcome of my game. I achieved almost everything that I set out to include in the project and am pleased with the final product.

One thing I did not include was the Mini-Game for generating popcorn (a currency in the game) to send to Facebook friends. I said in the original Project Plan that this was going to be an additional feature and would only be included if time permitted – so not having time to implement it was not a huge disaster. Instead of this feature, I added a simple button to send a 1 popcorn to the friend. I feel that this is a justified substitution to make as it took a lot less time.

As well as not having time to implement that optional feature, I also changed the primary target of the App from iPhone to iPad. This did not have a big impact on the overall game as the layout is consistent for the two platforms. However, in terms of testing, I found it easier to test the game on the iPad rather than the iPhone, purely because the larger screen size made it easier to see if something was wrong.

In the original plan, I specified that the main platform would be iOS, with the option for expanding to Android if possible. Although it was not possible to also run my game on Android during this project (again, due to time constraints), I have found that it is very easy to ‘port’ the game to different platforms, meaning that I could easily make an Android (and even a PC) version of the game if needed. This idea is also helped by my Facebook connection feature, which stores the player’s data in a database and can be loaded into the game. So the data is consistent across all platforms.

There were also several smaller additions from the initial plan that I would have liked to have implemented but I ran out of time. These features included additional jobs for the staff members to do and a ‘Premiere’ feature in which the player’s cinema would play host to celebrities for a night time special “bonus” level. It came down to a decision between implementing these additional small features, and attempting to implement Facebook integration. In the end I think I made the right decision as I got almost everything I wanted from the Facebook integration to work and I feel that this is a good feature to have as it improves connectivity between players. However, I plan to continue working on my game and ideally release it on to the iOS App Store so the features that I was unable to complete will likely be implemented before release.

I also made a last minute decision to disable the feature which allows the user to choose the carpet colour. I did this because when the player attempted to use this feature on the iPad, it crashed the game. Having tried everything to fix this, I was still unable to. So for the sake of the hand-in, I disabled the feature and labelled it as a “Premium” feature!

I knew that I would not be able to complete everything that I set out to do within the timescale as I was quite ambitious with what I could do. Since I plan to continue work on my game after the project deadline, I included a lot of features in my plan, which are all features that I will eventually add to the game – even if I did not have time to during this project.

# Design Patterns

I used an ‘Object Pool’ to generate and store the ‘Customer’ objects in my game. I found this feature massively improved the performance of the game. Before implementing it, the game started to ‘freeze up’ when too many customers were spawning – due to them being ‘Instantiated’ as and when they were needed. The Object Pool helped improve the performance as ‘Pre-Instantiated’ objects were stored so that the processing time-consuming ‘Instantiate()’ function (*Unity*) was run less often. The code of the actual Object Pool is in a Script called ‘*Object Pool*’ and the ‘*Time Controller*’ script accesses the pool to get objects for customers when they ‘arrive’ at the cinema. I also used the Singleton design pattern with the Object Pool.

Singleton involves a class storing a static instance of itself, so that data can be accessed from it without having to store a local instance of the class. Although I did not specify this pattern in the original plan, I found that it would be useful in some areas of my game, for the Object Pool and Facebook connectivity in particular, so researched the proper implementations of it and used them. Evidence of the Singleton design pattern can be found in the ‘*Object Pool*’ and ‘*FBScript*’ scripts of my project.

I did, of course, include MVC in my game as well. Although I initially found it tricky to implement MVC with the Unity Engine, I think my final implementation of it is good. I have Models to store the details of each component of the game (Screens, Customers, Staff etc.). These models are accessed by the various controllers of the game (*TimeController*, *StaffController* etc.). Originally, I had one large controller which handled the majority of the functionality. However, with each new feature added, this controller grew and grew and I quickly realised that this approach was terribly inefficient and horrible practice! So I decided to split the controller into several ‘specialised’ controller – each controlling groups of related methods. Finally, the View part of MVC is managed by Unity’s GameObjects.

So for each action, the relevant Controller will perform any necessary calculations, then call the Model to update the necessary values (if any). Once these updates have been done, the Controller will call the method to update the relevant information on the screen (if needed). For example, when upgrading the level of a Screen, the Controller will:

1. Work out the new level of the screen, and perform other calculations such as getting the new capacity of the screen.
2. Update the necessary values in the relevant Screen model.
3. Change the sprite of the screen to match its new level.

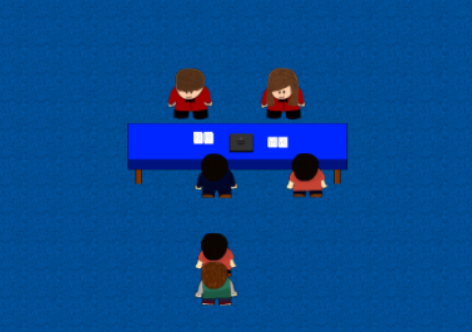
MVC is evident is several areas of my code. The most obvious ones are for moving and updating customers (‘*Customer\_Controller’, ‘Customer’*) and purchasing items from the Shop (‘*ShopController’*).

As well as using MVC for the main functionality of the game, I also used it with the Facebook integration feature of my game – both on the Client side and on the Server side. In the Unity project, the code is in the ‘*Login*’, ‘*UpdateDetails*’, and ‘*AddUser*’ classes. The Web API code (written in Java) is almost entirely in the MVC format. The View is the Game which calls the web servlet (Controller), which then calls the model to update/get details from the database. The retrieved data is then returned to the servlet which forwards it back to the game.

The ‘Update’ pattern is also used a lot in my game. I implemented this pattern using the Update() function in Unity, or the similar FixedUpdate() function on occasion. These functions will run once per frame so are ideal for moving objects on the screen or updating counts. The main implementation of them in my game is for moving customers around the cinema (‘*MovementScript*’) but it has also been used for various other elements of the game: from complicated things such as dragging the staff members to the desired ‘posts’ (‘*mouseDrag*’), to more straightforward things such as incrementing the time of day (‘*TimeController*’).

Another design pattern which I used is the ‘*Observer*’ pattern. Specifically, I used it with delegate methods – used to update instances of other classes in the project. One example of such a delegate method is in the ‘*Controller*’ class. When the user chooses to move an object, the delegate calls a method in the ‘*TileManager*’ class which then updates where the tiles are stored. I did have quite a lot more of these delegate methods but the restructuring of my controllers made a lot of them obsolete so I removed them.

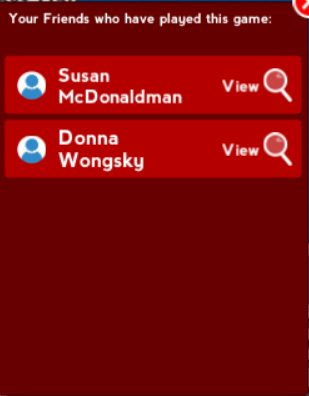
# Game Aspects

The Game aspect that is most prominent in my game is Concurrency. I have used this to good effect in the Queue system I created. Each staff member assigned to a queue runs its own thread (with varying delays based on their upgrade levels), meaning that several customers from the same queue can be served at the same time – at different times, with different delays. If a staff member is removed from the job/post, then the customer they were in the process of serving is returned to the head of the queue and the staff member’s thread is destroyed. The code for this can be found in the ‘*CustomerQueue*’ class. I am very happy with how this code works as it truly reflects how such a queue would work in a real cinema.

[Figure 1]:

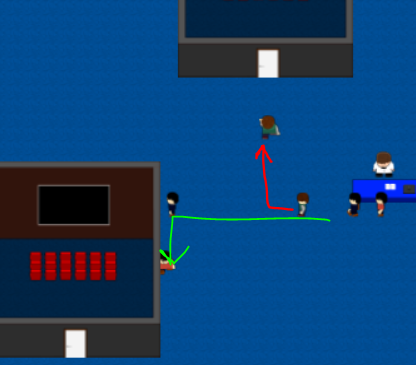
Each staff member runs on a separate thread.

I also used concurrency when saving the game state to the database. I found that the process of saving so much data (in BLOB format) to the database was a very slow one! And hence, when trying to run it in the same thread as the main gameplay, the entire game would freeze and wait for the save action to finish. This was obviously no good so I moved saving to a separate thread. This kept the main thread open for processing the game while the second thread carried out the save actions. The code for this is in ‘*MainController*’.

I also included online connectivity in my game – through Facebook integration. This feature means that the cinema data is stored in, and retrieved from, a database, meaning that the player can access their cinema on multiple devices using their Facebook login. The player can also view which of their Facebook friends have played the game – as well as being able to view their friends’ cinemas. They are not able to perform any actions in a cinema which is not their own (i.e. they can’t purchase any items, move staff etc.) but they can send a gift of popcorn (a currency in the game as mentioned before) which can help their friend to improve the cinema. I am happy with the way the Facebook functionality works as it will increase competitiveness as friends compare their cinemas and compete to have the ‘best’ cinema. The code for the Faceboo­k integration is in ‘*FBScript*’, as well as several areas for loading/saving in ‘*Controller*’. The Web API code also deals with Facebook integration. Note, that there is also an offline save game option, to allow the user to play the game even if they do not have internet connection.

[Figure 2]:

List of friends who have played the game

I also successfully implemented Artificial Intelligence in my game. The ‘*Floor*’ class has a ‘*FindPath()*’ method which uses the Algorithm outlined in my project plan to find a path from point to point (e.g. from the ticket queue to the relevant screen). In the image in Figure 3, you will see that customers will take different paths to go to the screen that they are ‘visiting’.

I originally carried out the path finding process as and when a new path was needed. For instance, when a customer is finished in the ticket queue, they would find a path from the ticket queue to the relevant screen. However, I noticed that this slowed down the game massively. I considered moving the pathfinding to a separate thread (to leave the main thread open for processing the game), but finally arrived at a different decision on how to fix this problem. I realised that the path from point to point will be the same throughout the day (the user is unable to move objects during the day, hence the paths will not be affected) so I decided to simply calculate all the paths at the start of the day and store them locally, ready to be read when needed. This led to a sizeable performance increase in the game as a whole.

[Figure 3]:

Customers take different paths based on which screen they are visiting

# Testing

Despite the fact that I was developing an iOS game, the majority of testing for my game took place on my (Windows) laptop. This was purely due to convenience – there was only one Apple Mac (needed to connect to the target iPad device) that I had the necessary admin permissions and/or software downloaded on, and this Mac was in the Labs in the Queen Mother Building. Obviously, it was easier to for me to just work on my laptop at home.

In terms of actual testing procedure, I tended to fully code a feature/element of the game, then test that one feature, recording any bugs found. I then fixed these bugs and repeated this process until no bugs remained. After implementing three or four features, I would do a system test and check that these new features integrated with the rest of the game correctly, recording and hence fixing any bugs as I went.

However, I did of course test on the iPad as well. These tests were a lot less frequent at times (due to the reasons mentioned earlier concerning the Mac), but were very important as there were bugs which occurred on the iPad which did not occur on the laptop version. To begin with, I would test the game on the iPad every ten features or so but nearer the deadline, it became a lot more frequent as I tried to fix various iOS bugs. One bug in particular repeatedly caused me problems – the feature to allow the player to select which colour of carpet to use. For some reason, when I ran this feature on the iPad, it caused the game to crash, despite working fine on the version on my laptop. I kept changing the code in the hope it work then re-deploying it to the iPad for testing, only for it to crash very time. This was not a quick process either as putting the game on to the iPad took a long time (usually around 15 minutes).

When testing on my laptop, I was able to use Visual Studio’s Debugger to examine the code line by line as it run, which was very helpful for finding bugs – particularly with moving customers to the correct points along their paths, where it was impossible to see what was going wrong without using the Debugger! However, this caused problems as multiple customers were running the same script simultaneously so the debugging showed for all the customers at once – which made checking the movement very hard. I got around this problem by tweaking the code so that the game temporarily only spawned one customer per film showing so it was a lot easier to examine what the customer was doing and hence find what was going wrong.

The results from my testing varied: some bugs were picked up straight away and hence fixed quickly; others did not show up until several system tests later. I think the main reason for this is that there are so many different features and elements of my game/genre that it is very hard to test every combination of actions to check that nothing goes wrong. So when I ran the game and perhaps tried a different sequence of tasks that I had not tried before, it uncovered some previously unfound bugs or errors. As far as I am aware, there is no real, systematic way of testing all possible sequence of actions in this type of game – other than just testing absolutely every combination! I did, with help from one of my classmates, conduct what we referred to as the “150 day Stress-Test”, in which we ran the game for 150 in-game ‘days’, trying every possible action that we could think of and trying to break the game! This unearthed several bugs and performance issues which had not previously been found by my normal system tests. I recorded these bugs and looked into way of fixing them. Issues found by this testing include the slowing down of processing if too many customers spawned (prompting the introduction of an Object Pool) and various customer movement bugs/glitches which caused them to walk through walls! I have since carried out similar tests and they too have been successful.

# In hindsight…

If I were to tackle this project again, I would certainly create separate controllers a lot earlier in the process – before more features were added and the one central one becomes swamped. Although the structure of my code ended up being okay, starting with one big controller (as mentioned earlier) meant that I had a lot of refactoring to do later on in the project which was not easy and caused several errors which I had to spend time fixing. If I was to start again, I would create a new controller when a new ‘group’ of features is started. For example, if I started working on customers, I would create a controller for customers straight away, instead of putting the functionality into the main controller and having to separate it later in the process.

I would also try and develop on the Mac a lot more often where possible. Even when I was in the labs, I tended to develop my game on my laptop as it is what I am most comfortable with. However, when it came to getting the game onto the iPad, using my laptop caused the process to be even slower as I first had to commit and push the source code to GitHub and then (on the Mac), download the .zip file and extract it. This added roughly 5-10 minutes of time to the process which slowed everything else down – it was often the case that I could not continue with a certain feature until I had tested the current version on the iPad. If I had developed straight on the Mac, this would have saved a lot of time over the course of the project.

In conclusion, I am very happy with the outcome of my project and I had fun all the way through (with the occasional, stressful exception)! I am happy with my use of Design Patterns, particularly Object Pool, and I am delighted with how the Queue system works with concurrency. I am also very happy with the Facebook integration part of my game. I look forward to continuing to work on my game and hopefully releasing it onto the App Store.

**Word Count: 3111**