EXTENSION REPORT

TEAM INI – ASSESSMENT 3.

Introduction

Aims of Assessment 3

The team's aim throughout this assessment was to achieve the following:

To continue with the implementation of the Single Statement of Need created in assessment 1:

> *"The goal of this software project is to provide a game that allows the player to control and manage the flight plans of several aircraft from the perspective of an air traffic controller presented using a graphical user interface. The game must take into account safe distances between aircraft, avoiding collisions and must calculate a score for the player."*

2. To extend the architecture provided by the previous team to accommodate our statement, with the implementation of a suitable scoring system and the implementation of the rest of the specification (supporting the landing and take-off instructions).

We chose to extend the code provided by team FLR, the game being Bear Traffic Controller (BTC). The reason we did so was that their game was most similar to Controller Concern (CC), in that the aim was not to guide aircraft individually from waypoint to waypoint as seen in the majority of games, but to guide self-steering aircraft through the airspace ensuring that no collisions occur. As early as the first assessment we had come to the conclusion that this was much more suitable for a game when upwards of 10 aircraft were present in the airspace. As a result, instead of concerning ourselves with converting the game to suit our intended play style we could concern ourselves with the implementation of the required features for this assessment.

Throughout the document, we have referenced the test report to support our implementation. *[T§2.1.1–HoldingWaypoint; T§2.2]* indicates a reference to sections 2.1.1 and 2.2 of the test report. Reference that include *T§2.3.1—ID(s): NX* are also references to the new system requirements.

General challenges encountered.

One immediate problem we encountered was the length of the two main classes; both Demo and Aircraft were over 1000 lines long and contained the vast majority of the games functionality. The number of variables and functions within these classes made working with these rather difficult, even frustrating at times when attempting to find the required functionality that was to be modified. One of the more annoying aspects was working with a flight plan generator that was inside the aircraft class.

Another 'feature' of BTC is the inability to modify aircraft velocities once the game had begun. Instead, this was done through the difficulty selection menu, with greater difficulty corresponding to greater velocity. This removed one of the control methods used in our previous game from being available for the player to use when managing aircraft.

While working on the original code, we had to modify multiple values in order to fit the game to our vision. However, this was not as trivial as expected as there were basically no named constants in the code beside very low-level ones such as ALTIMETER_X (which indicates the position of the aircraft's altimeter).

Simple modifications such as changing a repeat colour proved to be difficult as it required changing the colour value in almost all classes (more about this in the GUI section). In order to increase maintainability we added many named constants even though it was not necessary in respect to the requirements of assessment 3.+

Fulfilling requirements: Modifications and Justification

This section involves discussion of how we decided to fulfil the requirements set out by this assessment, namely the land, take off and scoring mechanisms.

Implementation of the airport

[T§2.3.1—ID: N4]

In our version of BTC, we chose to implement the airport as a single point, where aircraft would enter and leave the airspace. As a result, we extended the existing Waypoint class while adding the ability to contain a number of aircraft implemented using an ArrayList<Aircraft>. This represents the aircraft present at the airport. It was decided that this should be limited to five aircraft, requiring that the player pays attention thus adding a little more challenge to play, in addition to managing visible aircraft. If the number of aircraft within the airport was greater than the limit of five set, the game will end. While at first we thought simply to reduce the score, it was decided this was not harsh enough a penalty as we wanted the user to need to pay attention to this aspect of play.

We originally intended that aircraft which had entered the airport would be given a new flight plan and score, but the instance would remain. In the event new aircraft properties were required in the fourth assessment they too would remain, allowing the same aircraft which had entered the airport to leave again. However, the code provided by the previous team had flight plan generation logic within the aircraft class, which was called upon creation of a new aircraft. As a result, it would have been tricky to do what we desired, and deemed it not worth the effort. Therefore, when an aircraft instance enters the airport it is stored in aircraftList, but later on not used generally not used as a new instance is created upon take-off (mentioned in detail later).

Implementation of landing mechanism

Given the way BTC is meant to be played (which is to say that the aim is to ensure that an aircraft reaches its correct exit point, no matter the route), the game required a landing mechanism that would be fun yet challenging for the player to manage that did not have the aircraft landing automatically upon arrival (which would have made a land button pointless too) which would present the airport as being no different to any other exit waypoint, adding nothing new, fun or challenging to the game.

This problem had been given prior thought in assessment one, and the answer was to create holding patterns around the airport that the aircraft automatically followed until explicitly instructed by the player to land. The player would have to manage the altitudes of aircraft in the holding pattern to prevent collisions whilst having aircraft descend to or below a certain altitude (5000ft in this case) before the order to land can be given. This idea was also endorsed by the customer when we met with him.

Holding Patterns.

[T§2.1.1—HoldingWaypoint; T§2.2]

After reworking the way altitude control was implemented (see 'Other Modifications'), we then needed a way of having aircraft circle the airport continuously until instructed otherwise. This problem was solved by use of a new class HoldingWaypoint which extends Waypoint.

- Each holding waypoint holds a pointer to another holding waypoint, nextWaypoint.
- When an aircraft arrives at a holding waypoint, its currentTarget is updated to target that holding waypoints nextWaypoint.
- This process repeats indefinitely until the aircraft is directed to land; the only other way an aircraft can leave the holding pattern is if manual control of the aircraft is taken, though once removed it will return to the pattern. Also, the flight plan can no longer be dragged and dropped once the aircraft has entered the holding pattern.

If a generated aircraft is designated the airport as its exit waypoint, it selects the nearest holding waypoint to its entry waypoint and uses that as its entrance to the holding pattern.

Landing the aircraft.

[T§2.2—Landing, T§2.3.1—ID: N7]

Aircraft can be instructed to land by pressing F on the keyboard, or the land button in the GUI. These will only respond when the aircraft has altitude of 5000ft. Upon issuing this command, the aircraft reduces its velocity and altitude (now targetAltitudeIndex == 0) and turns towards the first of a series of landing waypoints. These act exactly as holding waypoints, but are not drawn. These waypoints guide the aircraft from one to the next and finally to the airport. Just as when an aircraft leaves through an exit waypoint, it is removed from the game and its points are added to the players score.

In order to improve playability we added a text indicate which aircraft "want" to land – this is discussed in detail in section the GUI Modifications – Textual information.

Take-off.

[T§2.2—Takeoff, T§2.3.1—ID: N6]

The implementation of the take-off function was considered of first-class importance and thus was discussed back while we were creating our game for Assessment 2. Despite the fact that we did not implement it then, we already had a rough idea how it should work together with the whole airport. In general, due to FLR's code, little was required to change in order for the take-off to function as follows:

As the airport is visible as a waypoint on the screen, once the user clicks it and there are aircraft in it, simply randomly choose one from the list (which is stored in the Airport class, as mentioned before), remove it from the list and return it back to Demo class which handles what happens next.

Even though in our code the Airport returns a specific instance of an Aircraft object, for the sake of simplicity, we generate a new Aircraft with a few special conditions:

- The origin point is always the airport.
- The destination point must not be the airport.
- It starts with a set altitude of 100 ft. and immediately starts climbing (thus making it more

realistic).

- The first waypoint is a specific waypoint takeoffWaypoint declared in Demo.java through which an aircraft must pass before continuing on its route (making it appear as if the aircraft is actually taking off).

As the Airport class was written by us specifically for this assessment, we made it as simple as possible. Even so, implementing this required us to change Demo and Aircraft classes:

- In Demo.java - altering the generateFlight() and createAircraft() functions to make sure they allow custom rules for aircraft coming from the airport.
- In Aircraft.java - a simple if statement in the constructors which swapped the first waypoint in the route of the aircraft if the origin point of the aircraft was the airport.

Lastly, in order to increase the difficulty level, we decided to add a timer for take-offs – that means that aircraft can be landed every 5 seconds. This is customisable in the Airport class as a named constant. Implementation of this timer was simple as time between each frame was already being counted, thus it required us only remember when is the next take-off (basically nextTakeoff = timeElapsed + TAKEOFF_DELAY) and making a check whether the time has come. We also added a visual timer which is discussed later in this document.

Scoring.

[T§2.2—High Score, T§2.2—Airport, T§2.3.1—ID: N1]

This feature was considered in the first phase of the project when as a team we decided that "we assign certain number of game points for every plane that exits the airspace and the score increases as more number of planes exit the airspace." Refining this idea further at the beginning of this assessment phase, we decided upon a scoring system that functions in the following way:

- Aircraft are each assigned a set number of points upon entering the airspace.
- Points are removed from an aircraft if it and another aircraft breach the separation rules.
- The remaining points held by the aircraft are added to the players' score upon successfully reaching its designated exit waypoint. If an aircraft has a negative number of points, 0 points are added to the users' score (else the user could simply steer the aircraft out of bounds, creating an exploit. See next point).
- If the aircraft goes out of bounds of the airspace, no points are added to the users score.

With BTC's implementation, the first three points were simple to implement and worked as anticipated. However, unlike in CC aircraft return to their flight plan after manual control had been released, meaning that the likelihood of an aircraft going out of bounds was very low.

While this would be simple to rectify, we decided that with the added complexity of managing airport its holding pattern, looking after aircraft which had lost their way after manual control was taken would be too overwhelming for the player.

Balancing scoring.

[T§2.3.1—IDs: N2, N3]

There are a number of variables that can be modified to affect the balancing of scores dependent on selected difficulty:

- Separation rule radius
- Points lost upon separation rule breach
- Points held by the aircraft upon entry into the airspace.

We also have to take the spawning frequency and speed of aircraft, which will both affect how many aircraft exit the airspace per unit time.

All of these attributes exist in the Aircraft class. Originally the intent was that across all difficulties, the only variable to change would be the separation rule radius, with the others being constant. On harder difficulties, the separation rule would be smaller than on easy, making it more difficult to lose points but providing less of a warning to players. The effect of this would be that while a player would lose fewer points due to breaches of the rules, they would receive less warning should a collision being about to occur and more likely to lose the game as a result.

The client quickly pointed out that with this point scheme, someone playing on a lower difficulty could easily collect as many if not more points than someone playing on hard, as they would be less likely to lose and therefore would play for longer. To overcome this, we decided to implement the following instead:

- Separation radiuses would be uniform across difficulty
- Aircraft would have an increasing number of beginning points assigned to them when created, respective to increasing difficulty. We decided upon values of 10, 15 and 20.
- In order to punish separation rule breaches more harshly aircraft would lose all held points upon doing so.

With this implementation, playing on harder difficulty will yield a higher score, both because of the more frequent spawn rates of aircraft and also their increased base score. On easier difficulties, both of these will be reduced making reaching a higher score much slower to compensate for the less intense experience.

Out of bounds aircraft.

The final component of implementing scoring was ensuring out of bounds aircraft were removed from play. In BTC, aircraft that went out of bounds would simply become deselected (if they were selected by the player) and manual control would revert to automatic control, with the aircraft entering the airspace and returning to its intended waypoint. Implementing the desired effect was straightforward, involving the modification of Demo.update().

GUI Modifications and Justification.

This section talks about all the modifications which were related to the GUI and were not covered in previous sections. This includes not only new buttons and other changes which were done in order to fit the new requirements, but as well as changes we necessary in order to improve the playability of the game.

Dominant colour and background.

After seeing the game for the first time, we immediately noticed how visually dark it was. This problem this especially exaggerated on different screens like laptops - as they often have darker screens. As we wanted to follow our last assessment's requirements at least to a certain degree, we decided to brighten basically every element of the game - such as text and assets.

The main thing we change - the green lines and text all around the game. This, despite seeming trivial, proved to be problematic with old implementation as every class and every small part of code that drew any part of graphics had its own "local" colour defined - meaning it was not a global constant, despite actually being the same colour. This is the first thing we did - having introduced a global named constant (which can be referenced as *Main.GREEN*), we just replaced all local colour references with the created constant - this made change of theme/colour much simpler.

The background was the second thing we had to change for two main reasons:

- As mentioned earlier - it was really dark, thus making the airport barely visible
- We believed the airport was too big for the scale, especially compared to the size of aircraft

Due to these reasons, we decided on creating our own, new background which would fit our earlier requirements as well as make help us implement the new requirements.

Creating the background itself was trivial - we essentially used the background from our original came and added the airport to it. However, changing the original background introduced a few problems - mainly that the size of the background was actually much bigger than visible. We decided to change this as we believe it was a wasted of resources and actually resized the background to fit the actual visible size.

Resolution.

Starting and playing the game proved to be challenging on practically all mainstream laptops. The reason for this was the inappropriate resolution chosen – 1280 by 960. Despite the original requirements saying that the game was supposed to work on the Software Laboratory's computers, we agreed that using a more standard resolution will not only open the game to more players but simplify development as our whole team mainly develop on laptops.

As for our original game, we decided on using a slightly lower resolution of 1280 by 720. The only downside to this was the fact that the physical airspace shrunk. However, we decided this reason was not enough to retain the old resolution.

Changing the resolution prove to be extremely simple and introduced practically no problems. In order to change it we only had to edit one final static variable. The problem after this change was the fact that some aircraft spawned with flight plans which made them leave the visible airspace. The solution to this was also simple - we only had to change the location of some of the waypoints.

New and modified buttons.

[T§2.3—ID: N7]

In order to comply with the new requirements a land button had to be implemented. We decided that placing it in an obvious place was essential. After a short discussion it was agreed placing it near the "Take Control" button would be sensible and will not increase interface clutter. Furthermore, since not all aircraft need to land at the airport (only those whose destination is the airport), for these aircraft the button will be hidden to avoid confusion. Lastly, in order to introduce some challenge and a bit of realism, the land button was enabled only when the aircraft's altitude was 5000ft. Up until then the button reads as "Lower Altitude!"

Adding the button in terms of code was not as simple as expected. In order to do so, one had to declare the button, declare its action (i.e. what happens upon clicking it), explicitly call the button's *`draw()`* method and finally, explicitly call the button's *`act()`* method.

Changed position of waypoints.

[T§2.3.2—ID: O13]

As mentioned earlier, some waypoints had to be reposition after changing the resolution of the game. However, due to the addition of the airport, it was decided that repositioning all of the waypoints made sense in terms of playability. Doing so was extremely simple as it required only changing the x and y coordinate of the older waypoints.

Displaying the score.

[T§2.5—Alpha/Beta Testing]

Through testing it was noted that seeing the score after the end of the game was difficult. As a result, we modified by using a bigger, clearer text for the scores. Furthermore, we also made sure one can skip the end game screen only by click space instead of any button as in the occasion a user had manual control over the aircraft using the keyboard at the time of the collision, the user will release the key that was pressed once.

Textual information.

New textual information was added throughout the game, specifically in the two following places:

- Near the airport – its name, number of aircraft currently landed and countdown timer until the next take-off.
- Take above every aircraft that "want" to land and those that are currently landing.

Adding the information near the airport was necessary as it had to stand out of the other waypoints. Information like the number of aircraft that landed was also mandatory as the game ends upon overflowing the airport.

The text above the aircraft was added in order to make aircraft that want to land stand out, generally making the player pay more attention to them, as otherwise may not understand why the aircraft is circling the airport – we made it explicit that the aircraft is has to land.

Adding all of these textual information was very simple mainly due to the fact that we already had an example of how to do so – text appeared near the entry and exit points at the corners of the screen, thus adding our own text was essentially copying over what was already written and changing position.

Other Modifications.

This section describes smaller alterations made to the game that either changed the existing implementation of a procedure to facilitate implementation of new features or to bring the game more in line to our desired style of play, implementing features present in Controller Concern and removing features from BTC that was saw as unnecessary or counterintuitive.

Control altitudes

In FLR's version of the game they had the concept of "Control Altitudes" which worked as follows:

- Every aircraft flies in one of the two altitudes.
- The player can change "Control Altitude" by using the mouse wheel which corresponded to the aircraft's altitudes.
- Once the Control Altitude is set to (for example) 30000 ft, only aircraft in that altitude can be controlled.
- All aircraft which are in a different altitude are represented as a different shade of grey.

In our opinion, this concept seemed counterintuitive. It introduced unnecessary complexity to the game, preventing the user from being able to quickly select any aircraft. Thus, following the spirit of our original game, we decided to remove this feature. Doing so was very simple - as the implementation of this was a simple check of the current control altitude and the aircraft's altitude (in Demo.java), we only had to remove this check. The same check (however in a different place - Aircraft.java) was conducted in order to darken the aircraft, thus we had to remove it from there as well.

Changes to altitude (rate of change and new set values)

The way in which FLR had implemented aircraft cruising altitudes and changing between them only supported two altitudes, with no easy way of extending this. Therefore it was required that the altitude system was replaced with the following:

- Each aircraft is initialised with an integer array list of cruising altitudes [100, 5000, 10000, 15000]*, and a random altitude is selected from indexes 1 to 3, representing the aircraft instances cruising height. Existing values can be modified and new values added to this list as desired. Index 0 is only used when the aircraft is landing.
- Altitude is altered by setting an integer value targetAltitudeIndex to the index of the desired altitude. The aircrafts current altitude remains stored in the aircrafts position.z.
- Every time the aircraft is called to update, if altitudeList.get(targetAltitudeIndex) equals or almost equals the position().z() of the aircraft, set the latter to the former. The "almost equals" is required as when the aircraft is ascending/descending, position().z() is very unlikely to ever equal the required value for that check to be true.
- If the aircraft's altitude is greater than the target altitude, the fall() method is called which decrements the altitude by a set amount per update, specified by altitudeChangeSpeed. Similarly,

if the aircraft's altitude is lower than the target altitude the climb() method is called, which increments the altitude by altitudeChangeSpeed.

- targetAltitudeIndex is modified directly by the user through the GUI allowing them to increment/decrement.

Aircraft destination/target logic.

[T§2.1.4—Aircraft]

Originally, each aircraft had a currentTarget and destinationTarget attributes of class Vector. The problem with this is that we could not know if the aircraft has reached the airport (as it is just an extension of a Waypoint, as mentioned earlier) without doing some comparisons of positions between the currentTarget's position and the airport's position. In order to avoid this, we changed the aforementioned attributes to be Waypoints. Afterwards we had to simply update a few places in the Aircraft class to make sure it gets the currentTarget's position (essentially adding position() calls).

This change allowed us to simply check whether the aircraft has reached a Waypoint of type Airport and call its appropriate land method.

New controls and faster 'direct control'.

[T§2.2—Keybinds, T§2.2—Landing]

One of the issues we found with the original game were the controls. As it stood before our changes, one could control the aircraft in the following manners:

- Press "Take Control" (or space) and using ← and → turn the aircraft
- After selecting an aircraft, clicking and drag from one waypoint to another altered the aircraft's path

Even though the second method was very intuitive and well done, we decided we wanted to make the first method much more flexible and as intuitive as the second method.

In order to do so, we used the same requirements we did for our own game in Assessment 2:

- Control the aircraft's heading with ←/→ or A/D.
- Control the aircraft's altitude with ↑/↓ or W/S.
- Take "direct control" as soon as the player changes the aircraft's heading, without having to press the on-screen button or any key.

Changing these proved to be rather simple, even though they required changes in both Demo.java and Aircraft.java.

Software engineering solutions and approaches

While doing assessment one, we researched different engineering approaches and came to conclusion to use an Agile method, more specifically the Scrum methodology. We decided Scrum would perfectly fit our project due to "the short development cycles" and in order to "suit the frequent assessment deadlines". Furthermore, for assessment 2, we decided more conduct weekly meetings as well as introduced an engineering method we did not mention in the report of assessment 1 - Pair Programming.

Even though we gained some knowledge and new experience during assessment 2 as we had time to use

Scrum and Pair Programming, we faced a major problem in assessment 3 - lack of time. In order to deal with this, we decided to change our Scrum plan - instead of meeting once per week, we agreed upon meeting at least 2 times per week (on Mondays and Fridays, to discuss what is going to be done and what was done) in addition to quick chats (essentially “sitreps”) after lectures to make sure everything is on track.

Furthermore, as assessment 3 required relatively little coding, we decided on splitting our group differently than earlier - this time we had 2 pairs of programmers and a pair that focused on the testing report. We had a dedicated pair on testing as we knew from Assessment 2 it was a major undertaking requiring a lot of time, thus doing the report from the beginning was a necessity.

As mentioned before, for this assessment we used Pair Programming as much as possible - this method, even though generally decreases efficiency, greatly increases code quality. Every pair had their own dedicated tasks which were assigned through *Github Issues*. Each pair decided at what time they can meet in order to retain flexibility.