Documentation to the Air Traffic Control Game -project

Anton Podlozny, 100558915
Tietotekniikka, 2023, 26.04.2023
anton.podlozny@aalto.fi

General description

The program is an Air Traffic Control Game. The game is centered around instructing commercial airplanes on their approach and departure from an airport. The planes are directed through a graphical user interface, with a game map in the center and two sidebars containing text information as well as controls. The map is randomly generated within predefined bounds. The arrival of the airplanes to the map is also randomized.

The player is tasked with directing airplanes to different runways, either for landing or takeoff. A collision may occur on the runway, ending the game. The plane may also run out of fuel while waiting for landing, causing a crash and a loss for the player. Points are awarded for successfully delivering and departing passengers.

Due to heavier course load than expected this spring, I wasn't able to implement as ambitious features as I had hoped. For example more detailed taxiing to gates and game saving were not feasible. All the core functionality and some additions are however there. The UI also changed a bit, with plane direction buttons moving to a sidebar.

The graphics are very simple and a bit pixelated, but graphic design was not the focus here. Graphic modding however could easily be done just by changing the files.

I estimate the final difficulty as medium and possibly demanding. The game implements all the features mentioned in the standard spec, and a few of the additional ones. The program is unpolished in some parts, but has solid algorithms in the backend. This was also a very visual project, and combined with the very little GUI instruction we received and poor documentation of Scala visual libraries the GUI was a very challenging task.

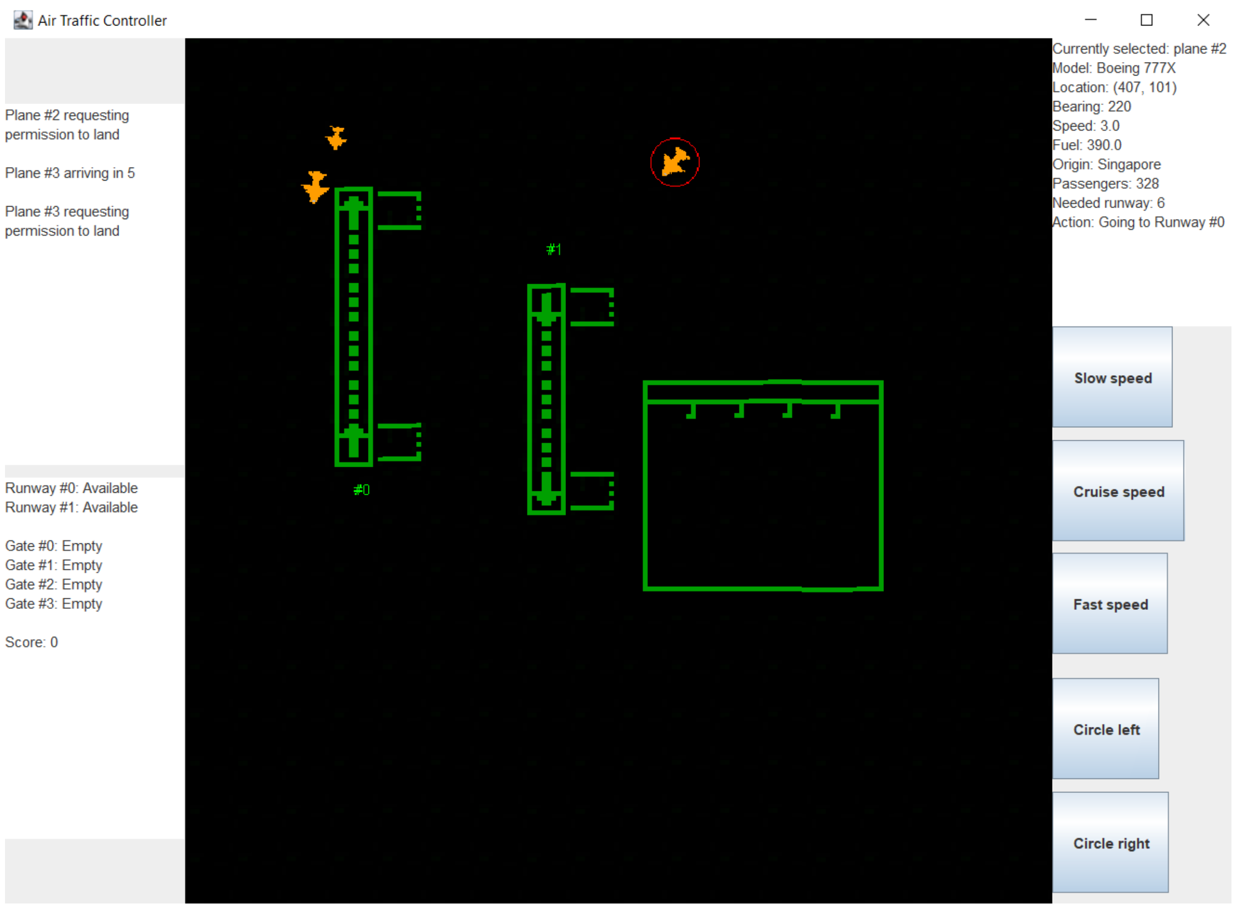
User interface

When starting the game, you will be presented with a short intro and some initial instructions. After reading those and closing the window, you will be asked to enter the numbers for gates and runways. These are then used in map generation.

After answering the prompts, you will be presented with the main playing window (illustrated below). It contains a game map and two sidebars. Shortly the planes will start coming from different directions. They will by default choose a random runway, and go for landing. You can however select a plane by left-clicking it, so that a red circle appears around it. The right sidebar will now present you with information about the plane, and several buttons which can be used to direct the plane. The buttons are quite self-explanatory. In addition to this, you can click a suitable runway start (the arrow pointing into the runway) and redirect the plane there. The Action of the airplane should reflect this, and the plane will modify its course to land there. Be vary however: the pilots are over cautious, and may take a while to land. Too many reroutings could cause the plane to run out of fuel with no way to help it! You should also check the plane's needed runway, and choose the runway appropriately. Small differences can be mitigated by slowing the plane down, but be careful about that.

The buttons in the sidebar change with the state of the selected plane. After landing it will automatically move to a gate if one is available, or await a free one if not. The plane will disappear from the map for the duration of taxiing. After completing the taxi, it will proceed to board and refuel. Here the plane will require the attention of the player, as you need to order it to a suitable runway. Again, check the needed runway length. Lengths can be checked from the left sidebar.

After receiving the order to taxi the plane will disappear again, appearing at the waiting area for the runway (not yet blocking it). There it again needs to be ordered to take off. The revving of the engines will take its time, so don't plan to land a plane as soon as you order one to take off. If a plane takes off successfully, it will proceed to leave the map (and award you points). While in the air, the planes can not collide, as they will be at different altitudes. However you should take care that no planes intersect on the runways as that will end your game (and be really bad for the passengers).



Air Traffic Controller

Plane #2 requesting permission to land

Plane #3 arriving in 5

Plane #3 requesting permission to land

Runway #0: Available
Runway #1: Available

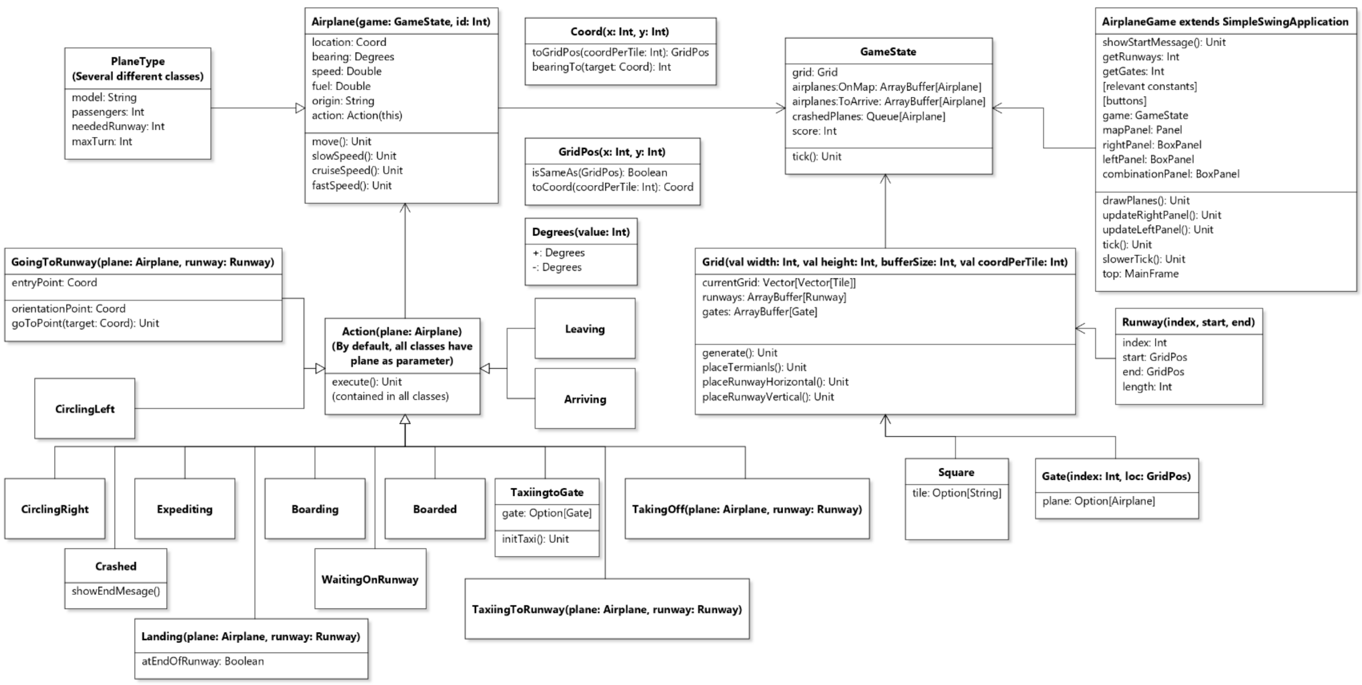
Gate #0: Empty
Gate #1: Empty
Gate #2: Empty
Gate #3: Empty

Score: 0

Currently selected: plane #2
Model: Boeing 777X
Location: (407, 101)
Bearing: 220
Speed: 3.0
Fuel: 390.0
Origin: Singapore
Passengers: 328
Needed runway: 6
Action: Going to Runway #0

Program structure

The final class structure is demonstrated in the diagram below (also attached). For simplicity only the most relevant details are shown.



(The diagram does not follow all UML-standard and is instead meant as an illustrative example. Variables and methods shown are mostly public.)

The structure of the program mainly follows natural lines. The code is broken roughly into following blocks: grid and map, airplanes and actions, unifying GameState and GUI. In addition there are several helper classes for positions and such.

The grid is contained mostly inside its own class. It oversees a grid of Squares and generates runways and gates for the GameState. Its most important methods are generate() and the planceX() methods it calls. These modify the value of Squares to form a map according to a set of rules. The Square-class is currently a bit redundant due to its code being transferred to grid-class. There were not that many ways to structure this part of the program, and it remained mostly unchanged from the original plan. More significant changes were made to its internal generation algorithm.

The next part is the Airplane-cluster. The Airplane itself is a robust trait that provides an easy way to create new plane variations. It is currently inherited by three different classes that define three different plane models. The brain of the whole program is however the Action-trait and especially its inheriting classes. These are behavior models that define what the plane does at the given point in execution. The Action of an Airplane is changed either by internal logic of the previous Action or by player input. The latter is implemented in the GUI-class. The logic of Actions is contained mostly in their execute()-methods, that access the grid-class when necessary. Execute() may also call helper methods inside the Class or the one defined in the trait itself.

This part remained relatively resemblant of the original plan. The main change is the absence of flag variables. Some of their tasks were replaced with methods, some implemented by timers and some deprecated. There are also a couple new Actions that were found necessary in the process.

Again I can't think of too many different ways to structure this part. One way could have been making a single big Action-class and using flags & conditionals to process the behavior. I however opted for separate Classes due to clarity. This proved a bit challenging to integrate with GUI however, and I had to resort to non optimal solutions. Perhaps creating some flag variables in the Action-trait itself would have helped.

The GameState is the unifying component between the Grid and Airplanes. It has a reference to the Grid-object and contains the Airplanes in a Buffer. It is a necessary class to store the state of the program at a given moment, and would be crucial if the saving feature was to be implemented. For what could have been done differently, it could have contained methods for accessing the Grid. Currently the Actions access the Grid-object quite directly, if through the GameState.

Lastly we have the GUI-class of AirplaneGame. This is a single class that derives itself from the Scala Swing SimpleSwingApplication. Due to being relatively inexperienced with Scala and programming in general, I was at a bit of a loss for how to implement the UI. I had used Swing before and this course encouraged it, so I decided to go with it. This might have been a mistake in hindsight, as it was really troublesome to find any material or good documentation for Swing. I also encountered many limitations and seemingly efficiency problems that were not easy to solve with Swing . As such, the GUI of the application is definitely not its strongest part, and could have been implemented more efficiently by, for example, ScalaFX.

I also had several auxiliary classes for storing and processing values, such as GridPos, Coord and Degrees. The former two represent the two coordinate systems of the program: large Squares and pixels. The classes provide methods to convert between the two. Coord also provides the crucial bearingTo-method, that gives the angle from a point to another. This method is the base for GoingToRunway-Actions pathfinding. Meanwhile Degrees offer a simple way to represent and manipulate 360 degrees, offering over- and underflows when around 0.

The diagrams also omitted the displayedTexts-file that contains several classes that are used for generating text shown by UI. These could have been dispersed through the classes, but I found it clearer to bring them together into one file.

Algorithms

The final program has a surprisingly large amount of different algorithms. Here I have described the most important of them, omitting ones like choosing the right buttons for a given selected plane or queuing up & dequeuing planes for takeoff. These are however mostly well-commented or self-explanatory in the code itself.

The game starts with map generation, where the first algorithm lies. This is a sequential generation, where first the terminal and then the runways are generated.

All of these follow the same pattern. First a “random” Square is chosen in the allowed placement area (edge + 2 GridPos for terminal and … + 3 for runways). It is then checked if the Square is valid (if the given object fits there), by checking the values of Squares in the area and vicinity of the placeable object. This is performed by iterating through Squares in a particular direction for a particular amount. If the Square is not valid, the method is called again. This can at worst result in maps where the next object cannot be placed due to crowding, but this is mitigated into non-existence through controlling the number of objects and map size.

These validity conditions are lightest for the terminal (generated first), and strictest for the vertical runway (generated last). The terminal just needs to fit there. The runways however can not have any object within one GridPos of them (unless the runways are crossed).

There would be many ways to implement this map generation. The one I originally considered was procedural generation through wave-function-collapse This however proved to be significant overkill for my needs, as it is best when producing complicated shapes and many objects.

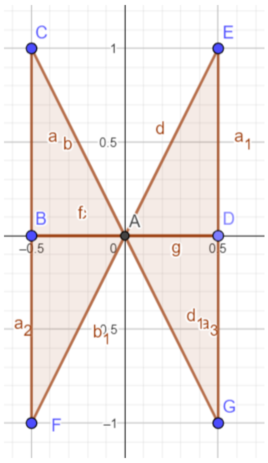
The validity checking and regeneration could also use a rework. Particularly to not end up in an irresolvable situation. This tail-recursive solution however was the quickest, simplest and not too inefficient.

The next important algorithm is the pathfinding for a plane going to a runway.

This algorithm is based on an orientation point and an entry point to a runway, The latter is located just outside the runway entrance, while the former is farther away and not on the centerline. The orientation point is determined by the location of the plane at the moment of calculation. The point is placed at constant distance from the runway, but the side in regards to the centerline is the same as of the plane. If the runway is horizontal, the orientation point is placed -2 GridPos in the runway directions on the x-axis and 2 GridPos to the side of the plane. The same is reversed for vertical runways.

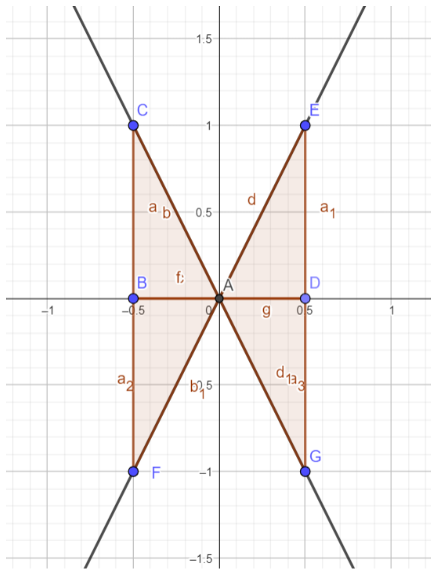
The plane is then guided first into the orientation point, where it receives permission to proceed to the entry point, after which it can go for landing. If the plane's bearing differs too much from the direction of the runway, the landing is rejected, and the plane must orient again. Here the curve could have been formed in a variety of ways, but the one I chose was probably the most straightforward, while preserving an appropriate amount of realism. The tool and methods used for this algorithm can also be adapted to other solutions.

Another algorithm that is crucial for the preceding one, is the one calculating the bearing from a point to another. For ease-of-use I decided to use a standard form, and state bearings in regards to the game's North (or up). As such 90 corresponds to East, 180 to south and so on. However finding the bearing to a point was not a trivial task. Here I turned to trigonometry.



I separated the problem into 4 situations, and formed 4 triangles (shown below)
Here, C, E, F and G would be the points from which I would calculate the bearing to point A. The coordinates of the given points are obviously known, so the logical choice is to use tan(a) = Δx/Δy. This gives the angle closest to the point of origin (eg C). What remains is to normalize the angles in regards to the North. Which can be done by adding or subtracting the angle to 180 or 360 degrees.
The above can however fail when Δx or Δy is equal to zero. This is handled by adding 4 exception cases, handling each direction.
Here the solution could have also been achieved by using analytical geometry, or perhaps an existing library. I however opted for an intuitive, self-made and good-enough way.

The missing piece between the two algorithms described above is the goToPoint algorithm that actually handles the steering of the Airplane into the desired direction. It starts by calculating the aforementioned bearing to target and separating into 4 familiar looking scenarios. Each of those also separates into two, that form the crux of the question: should the plane turn clock- or counter-clockwise? This can in fact be determined by comparing the plane's current bearing to the lines drawn here. For example a plane located at point C should turn clockwise, if its current bearing lies on the right side of the line. This can be put into a surprisingly compact conditional for every situation:



```
if plane.location.x < target.x AND ((plane.bearing.value < bearingToTarget) OR (360 - (180 - bearingToTarget) < plane.bearing.value)) then
  clockwise
else if plane.location.x > target.x AND (plane.bearing.value < bearingToTarget AND plane.bearing.value > 180 - bearingToTarget) then
  clockise
else
  counter-clockwise
```

Here the algorithm checks if the plane is on the left side of the target and if the current bearing is simply less than needed bearing to target or if the bearing is more but still falls onto the right side of the line. This check works also for point F, where the latter situation covers most of the area.
Then the same kind of check is performed for x > targetX, where the conditions are that the plane bearing must always be less than bearingToTarget and additionally be more that “180 - bearingToTarget” (the slope of the line).
All other situations require the plane to turn counter-clockwise.

The final part handles the amount, by which the plane turns. This “turn” is actually just rotating the airplane by modifying its bearing, but done frequently enough resembles turning. The amount by which the plane should be rotated is found by taking the plane's bearing and making it proportional to the aforementioned bearing to target. This ratio is then multiplied by planes specific maxTurnRate, resulting in a lower number, the closer the plane's bearing is to a right one. This prevents sudden movements when the plane is nearly on the right course.

The last algorithm I will describe here is the one for braking on landing. Otherwise the landing process is quite straightforward. If the plane overlaps with another plane, it crashes. If the plane is at the end of the runway and at required speed, it stops and changes actions. If it is at the end but speed is too high, it crashes. Otherwise it either slows down or maintains speed. The way to find the needed lowdown amount however was quite peculiar.

Here I used the max speed for a given plane and calculated the acceleration needed to slow down on a predefined neededRunway -length. The complete formulation is below:

$$d = \frac{1}{2} v \cdot t$$

$$\frac{2d}{v} = t$$

$$a = \frac{v}{t} = v \cdot \frac{v}{2d} = \frac{v^2}{2d}$$

Here d is the runway length, v is the initial velocity, t is time and a is acceleration. We model acceleration as constant, so the first equation represents distance traveled by plane with initial (linearly decreasing) velocity v until it comes to a stop. We then solve it for t.
This t can be substituted in an equation for acceleration, where we get the final result. This result is then used to linearly decrease a plane's speed, such that it mostly lands safely, but can crash at insufficient runways due to too high of a speed.

However it turned out that this algorithm failed for planes with short needed runways, due to the plane ending its landing not at runway end, but at center of GridPos. This could have been fixed with sizable changes to other parts of the program, but it was more straightforward to just modify the algorithm to allow landings if the needed runway matched.

Data structures

The program uses different Scala collections quite extensively. The default immutable collection is Vector. It is used to store the grid (where the vars in the Squares provide the mutability), store some of the displayed strings and id returned by helper functions. The choice fell on Vector because it was most familiar to me and the most efficient collection for almost any task. However nearly any other immutable collection short of maps and queues would have probably worked just as well.

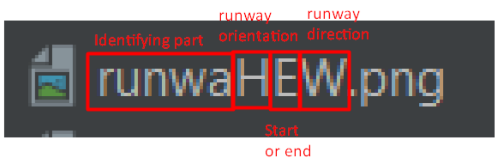
The default mutable collection for me was ArrayBuffer. These store for example Airplanes in the game. A Buffer was a natural choice, because the planes are mostly appended and iterated through. A Buffer however is not that efficient for random access, which the program sometimes does. The result was using an ArrayBuffer - sort of combination of both. Here a different choice could have been possible - a map. Mapping the index of a plane to the object itself would have satisfied the requirement, maybe even more efficiently, and cleaned up some code. The difference of using ArrayBuffer however is probably negligible.

Another mutable collection that I used is the Queue. It was the perfect abstraction for queuing planes up to gates or accumulating messages to show. In these cases the mutability needed is purely first-in-first-out matching the Queue perfectly. I also considered storing planes waiting for takeoff in one, but encountered problems if the user decided to launch them in a different order than which they were queued in. Here a Buffer or a map would probably have also worked, but Queue was the clearest choice.

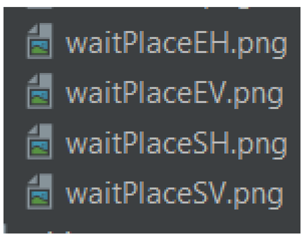
Files and Internet access

The files that the program needs to function properly are the graphics for tiles and planes. These are located in the root directory folder “Tiles”. My graphics are primarily 10x10p png squares with two colors. The png format is particularly important for plane images (that are actually also 20x20p), as they contain an alpha channel and are drawn on top of grid tiles. Using jpegs would cause some parts of the map to be overdrawn.

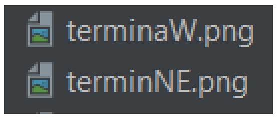
The graphics can easily be replaced just by substituting the existing files for ones with the same name. The program will even scale them into the right size automatically. The naming scheme of the runway tiles is as follows:



The orientation can be H or V for horizontal or vertical. The second letter (E or S) points to start or end, and the last is a letter of direction N - north, E- East, S - South, W - West (direction that the planes land or take off). The presented tile is the end of a horizontal runway with direction West.


waitPlaceEH.png
waitPlaceEV.png
waitPlaceSH.png
waitPlaceSV.png

The wait places are named similarly, with E or S standing for end or start, and V or H for vertical or horizontal.


terminaW.png
terminNE.png

For the terminals the format is a bit different with a single letter meaning the tile is part of an edge, and two letters representing a corner tile. Compass directions are used, with the two tiles presented being West edge and North-East corner.
If not sure about a tile name, its identity can be quite easily derived from looking at its graphic.

The slightly unusual names of the tile graphics are a remnant of the testing process, in which I printed the map into the console. This was made easier by tiles having equal-length names.

The program does not use the internet or save anything to disk.

Testing

The testing advanced roughly as I had anticipated in the technical plan. Most of the testing was done as unit tests for single functions or testing combinations of multiple related ones through the UI.

When working on the grid generation, I created a test class for generating the grid into the console as a simple printout. This allowed me to polish the map generation before going into GUI.

For other testing I followed the feedback I received on the initial plan, and first built a working graphic visual of the map and planes, before testing the Actions. From there it was quite straightforward iteration of Action algorithms and observing them on the map. Here addition of sporadic println()-calls in testable functions proved very useful, as it sometimes uncovered some unexpected behavior that was not observable in detail from the UI.

Other ways of testing could have included more command-line printouts before building the GUI. This however proved a bit challenging when I attempted it with the pathfinding algorithm.

The unit tests could have been done in a variety of ways. I preferred to build a small collection of functions and test them together, as sometimes getting to a relevant point in the game could take some time. For example testing a part of the pathfinding together with landing and gate-taxiing was a lot more efficient than waiting through already working functions to get to the taxiing part. Of course it would have been possible to build specific test situations, but I rather focused the efforts on upgrading the program itself. Testing through actual gameplay (focusing on relevant parts) also allowed for more edge-cases to be detected. For example I discovered that the program had a (suspected) memory leak when it caused my computer to blue screen after running for an extended duration. The suspected cause was an undisposed graphics object in an update method of the GUI-class.

The program passes all the current tests well enough. Due to a vast multitude of situations all possible cases are however difficult to test. For example the collision detection works less than perfectly, and both undetected overlapping and non-overlapped crashes have sometimes been observed. The failing of some functions can also go undetected as it can appear quite natural on the map. This has for example been the case with the pathfinding algorithms.

Known bugs and missing features

There are many small deficiencies in the game, such as unrealistic movement, general roughness of the UI, unintuitively of the plane direction and perhaps a lack of variety in games. There however are no clear game-stopping bugs in the current program that I would be aware of. There might be a very very small chance that with the largest amount of runways and gates the map fails to generate. This however is theoretical, as I have not yet encountered such a game.

There are also minor problems with movement to the runway, such as the flight path looking a bit weird or the plane failing to land on the first pass. These are however resolved gracefully by another attempt. In previous iterations the plane would sometimes get stuck in a circle. I am not aware of that happening in the current version, and even then it can be resolved with player input.

If the gates are full, the planes will stay to wait on the runway waiting area. When a plane dequeues, the rest do not move, which can cause planes moving inside each other. This however does not cause a crash. The same may happen for planes waiting for takeoff. This would be mitigated with the implementation of the planned taxiway network and more realistic waiting.

The plane's needed runway represents one needed to land at full speed. As such the planes landing on shorter runways at slower speeds may seem counterintuitive. This could be fixed with some other landing algorithm.

The redirection of planes to different runways may not seem intuitive due to lack of visual confirmation. This can be mitigated by checking the plane's current Action, and would be fixed by manipulating graphics and for example painting the selected runway or adding an arrow. The redirection is also sometimes problematic, as clicking a plane located at runway start, while another plane is selected, will cause the selected plane to be rerouted to the runway. Fixing this would require reformatting the plane selection.

Another possible problem with the UI is that the way to draw it is very inefficient. This has been mitigated by concurrency, but things may sometimes require a couple of clicks.

As for the code, there are a few small style inconsistencies and nonoptimal practices. Especially the Actions often modify objects outside of them directly and lack a concrete style of what handles what. They are also not that well modularized. There is also often usage of get-method for Option-objects. This is always accompanied by .isEmpty or similar check, but is still a fast, nonoptimal solution. I have already largely used foreach and forall with Options, and they could be possibly adapted instead of gets also elsewhere.

The features planned but not implemented are the following: more detailed taxiing, game saving and unusual situations. Of these the last was left out due to being relatively straightforward to implement, and me wanting to focus on more foreign features. The saving feature I deemed a bit unnecessary, as the games will probably be short. I believe however that the game's object-based structure could work well with json-base saving. The first one was a bit too ambitious of a goal. I had a few ideas on how to implement it, but decided to focus on more crucial features that I had clear plans for.

3 best sides and 3 weaknesses

One of the successes of this project is the pathfinding of the planes. It was a challenging but also a fun problem to solve, and the result is a relatively smooth and in many cases realistic movement for a variety of even unusual situations. The code for the algorithm is also relatively compact and modular, which could allow for further improvements, iterations and additional behavior models.

Another good part is the map generation algorithm. The inherent randomness brings a fun angle to the game and tweaking the generation options allows for different challenges and situations.

The third good thing I want to highlight is the structure of the program. Nearly everything is built with expandability and tweakability in mind. Graphics are replaceable, magic numbers are not used and are instead passed as parameters, new airplane models could be added effortlessly, new Actions implemented if needed and whole class-combinations could be decoupled from the program and reused elsewhere. For example the grid and its helper-functions could be used to generate different kinds of objects outside the Airport-context (this would however probably need rewriting of the generate()-functions or borrowing code from them).

For the main weakness of this program, I would point out the general roughness. Nearly every part could have been polished and improved more. This relates especially to the GUI, pathfinding and other movement. The plane movement is not 100% realistic, and in some parts shortcuts have been taken, such as the plane disappearing from the map to taxi. Improving these would have been relatively straightforward, but have required a significant time investment. For example the GUI would probably have to be rewritten in some other library and movement made more realistic by simply adding more lines of code and micromanaging the positions of the plane in different actions. I decided to focus my limited time on more fruitful features.

Another weak point is a bit of a contrast to the modularity of some parts. Especially the GUI and some of the Actions are quite purpose-built, and contain nonoptimal solutions that have to be taken into account if the code is to be expanded and reused. For example using the class-typeNames for UI button filtering was a bit of a band-aid solution to privateness of Actions and needs to be taken into account (or resolved with proper flags). The Actions also access and modify the grid quite carelessly, which in hindsight should have been through a clear, designed interface. Also, while the bigger wholes are indeed more modular, the Action-classes execute()-methods contain an outsized amount of logic. This causes them to be less readable, and a lot less adaptable. Later in the project I started building the helper-functions first, and including them into the execute() if-else structure later.

A third weakness is the actual playability of the game. The features, though up to spec, are still relatively limited; and the graphics, being a non-central part of the project, do not make up for the experience. As such the game can get quite repetitive and with time, boring. The randomness of every game helps this a bit, but for a true fun experience the either the features or the visual side should be significantly expanded. However, the true purpose of the project was to be a learning experience, and I focused my efforts accordingly on learning new things and handling a larger project.

Deviations from the plan, realized process and schedule

The final schedule looked quite different from the estimate, even though the work order did not.

The main issue was that I misestimated the amount of time my other activities and extra courses would take. As such, I wasn't really able to start working on the project in the first weeks, and there is really nothing to write about for those. The next misstep was choosing an ill-suited algorithm for the map generation, which caused me to waste many hours of early work.

As such the vast majority of work was done in the last two weeks, after the exams of the fourth period. This was a time of very intensive work that really tested my skills as a programmer. Although the final result would have been undoubtedly better had I started earlier, this was a really important learning experience on the limits of my productivity and simplifications that should be made in a time crunch. Armed with this knowledge I will be able to schedule my future work more efficiently and realistically, as well as being able to better anticipate the quality of my code with different time constraints.

As mentioned the order of implementation was mostly the same as originally planned. The major difference was that I moved on more quickly, finishing and polishing classes later, instead of finishing them before starting work on something else. The order of work was roughly: Grid -> Airplane stub -> Actions -> GUI -> more detailed Actions -> more detailed Airplane -> general polishing.

Final evaluation

The final quality is decent, but less than it could be. However, considering the time limitations, and my prior programming experience (none before this autumn), I am satisfied with the result. The program has a strong base in the backend and algorithms, but struggles to utilize the built framework to its full potential and present it graphically. This however means easy upgradability.

For example the polishing discussed earlier could be performed, with graphics substituted for more modern ones and more behavior models added to Actions. The frontend could also be rebuilt, increasing performance and responsiveness

The structure in my opinion is good for this kind of program, but some used collections could be reconsidered in favor of maps. The logic in Action-classes execute()-methods should also be broken into helper functions. Overall the structure should be very receiving for modifications, especially after the discussed refactoring is made.

If I had to make the project again, I would definitely start earlier as there would be less for me to learn by working under time constraints. I would also give more consideration for the GUI and for how the backend interacts with it. I also would make the Action-classes clearer and with a more defined structure and generally clean up the interfaces of classes. This didn't actually hamper my work during the project, but would increase the overall quality of the code and make further development easier.

References

CS-C2100 material by Seppälä, read 2022, chapter 8

CS-A1110 material by Sorva, read 2022, chapter 2.6 upwards

Inspiration for plane movement was taken from Airport Madness 4 by Big Fat Simulations, https://bigfatsimulations.com/game/airportmadness4

docs.scala-lang.org

stackoverflow.com

My past work in O1 and OS1

https://otfried.org/scala/drawing.html

Appendixes

Source code

Structure diagram

Game screenshots