**Design Proposal**

**Project Description**

This project is an “Air Traffic Control Simulator” that simulates the real-world air traffic control radar from towers use to manage flights near airports. As a simulator, it mimics the verbal commands controllers use to direct aircraft while having the game aspect of having objectives of directing arrivals and departures.

**Competitive Analysis**

There are instances of air traffic control games such as “Air Control” on the app store, which are very basic “drawing games” where the user draws a path to the objective runway. This project differs from this game in that it tries to simulate real-world procedures through a more complex command handling and execution model. Other instances of online projects include complex air traffic control simulators, which let users select an airport from a list of four or five. However, this project uses intelligent airport generation to create different runway layouts for each run. This helps create more variety and randomness to the game to help the user get better at “air traffic control,” not the airport itself. Also, this project features unique scenarios such as runway closures, weather, go-arounds.

**Structural Plan**

**Files**

The project is divided into the main file (atc\_simulator.py), the graphics file (draw\_functions.py), the classes file (objects.py), the generation files (flight\_generation.py, map\_generation.py, airport\_generation.py, perlin\_noise.py, weather.py), the command handling and execution module (commands.py), and the data files which handle airline and aircraft data (airline\_data.py, aircraft\_data.py).

**Object-Oriented Programming**

The project itself is highly dependent on objects that are set in the objects.py file. These objects include Flights (which contain subclasses of Departures and Arrivals), Airport, Runways, Airline, Aircraft, Weather. These objects help the generation features of the project have dependencies on one another based on the parameters of the object. For example, weather parameters affect airports and flights, airlines affect flight routes, airport size affects the number of runways or overall traffic. This also allows objects to have individual assignments to objects such as aircraft being set to a specific runway to land or depart from.

**Functions**

Most of the app-related functions are contained in the main file, where it will call external functions and objects from other files. Also, many functions are object-specific to simplify the structure where they are stored as methods for individual actions. With this structure, it is easier to add features without them negatively affecting the game or other objects.

**Algorithmic Plan**

Some important features of this project are its command handling and execution, intelligent map generation, and flight generation.

**Command Handling and Execution**

For the command handling, this is done with string searching. With a list of keywords that will recognize a specific action to be made, the game scans the command string inputted by the user for keywords and the parameters corresponding to the keyword. For example, in the command “Delta 123 turn right heading 220”, there are two primary components of the command. The callsign, which is used to identify the flight, and the heading command directs to change its direction to a magnetic heading of 220 degrees. For most English sentence structures, it is normal for a command to have the numerical parameter behind the actual keyword due to the subject and verb order. In this case, the number 220 is behind the word heading. The program uses this linguistic feature to simplify searching the parameter behind the keyword.

Also, the command handling model can identify both verbal and technical expressions of the callsign. For example, “DAL123” or “DAL 123” (with spaces), which represents the flight number, will be treated the same as “Delta 123.” This is possible due to the project’s overall object structure as well as the callsign structure. Airline data taken from real-world data is associated with the airline code part of the callsign. Also, callsigns can be split into letter and number parts. Thus, the flight numbers can be converted into its airline name counterpart and vice versa.

**\*Typo Recognition**

Additionally, the command handling model, also features typo recognition. This feature was inspired from nearestWords problem in week 5. However, in this model, a typo is compared to each keyword and is given an index based on its differences. The model searches for substrings of the word in the typo. 1 is added to the difference value if the substring is not in the word. Also, a dictionary with letter counts is created and compared with each other. 1 is added again to the difference value per difference of letter counts. This overall difference value is divided by the length of the word and typo combined to produce a dissimilarity ratio. The limit for the dissimilarity depends on the average length of keywords but is usually 0.4 or 0.5.

**Flight, Map, Weather Generation**

**Airport and Runway Generation**

For map generation, the components of the map that need to be generated include the airport, which is simply a placeholder for where the runways will be, the runways, and the waypoints. Airports are generated by a random four-letter string generator which will give its airport code. With the airport position centered on the map, the runway positions are generated by creating a radius vector from the airport position and setting the runway like a normal line to that vector. This creates runways that are tangent lines of different distances which stay within airport range and airport heading constraints.

**Future**: With a weather engine that generates winds, I plan to make airport runway generation dependent on winds to minimize crosswinds. Real airports actually set runway headings with the average wind direction of the area!

**Flight Generation**

As mentioned above, this project utilizes a significant number of different objects. Flights objects consist of Departures and Arrivals, which have different methods and parameters. To generate flights, random airlines, aircraft types are taken from existing data and random numbers for flight numbers. The airline and aircraft type that was generated affects the flights’ overall generation process. The flights generate routes that are either from or to the game-generated airport. This is done by calling an airport from a list of the airline’s most operated airports and will have a size and fuel parameters based on the aircraft type. This allows for size, fuel constraints for airports and the flight itself. Individual objects immensely helped in this generation feature to have greater realism and complexity.

**Weather Generation**

This part is arguably the most challenging part of this project. Weather has many features, mainly wind which affect flights. However, winds are vectors that have a direction and magnitude. Thus, generating individual values for this will be very difficult. Instead, I thought of generating what caused winds in the first place: air pressures. Since winds result from pressure differences, a direction and magnitude can be generated from a grid of different air pressures. Winds can be calculated by average air pressure differences in each direction like the wordSearch problem. However, to generate this air pressure grid in the first place, I utilized one line of NumPy code to generate a two-dimensional noise map to use.

**Timeline Plan**

This project has been able to progress quickly with features set by the MVP and other features that increase the simulator's realism. As of April 26, most of the command handling and execution module has been completed with tweaks to the typo correction under work. Intelligent generation of the airport, runways, flights, waypoints have been completed with preparation for future implementation of local weather systems. Constraints have also been modeled into the game though resulting game screens are still a work in progress. With most of the game currently solid, I have started working on a weather generation algorithm that will change the states of the airport, flights, and runways themselves.

**TP2 Update**

For the weather generation algorithm, for complexity as well as no external module usage, I have changed the way weather is generated. However, this time I created and implemented a modified Perlin noise algorithm which uses a paraboloid function to give the weather map a circular void effect and thus a cloud. Using a normal Perlin noise map proved too noisy for it to represent even similar to real-world weather.

**TP3 Update**

Changed weather display from grid drawing to image drawing. This method speeds up the overall gameplay. Improved constraints system to include warnings now with the type of constraint violation. The game over screen also features this cause system. Included new start screen before game as well as shortcuts to improve gameplay. Removed redrawAllWrapper after keyPressed in cmu\_graphics to speed up typing computation. Overall quality of life improvements for TP3 submission.

**Version Control Plan -**[**https://github.com/ay0503/Air-Traffic-Control-Simulator**](https://github.com/ay0503/Air-Traffic-Control-Simulator)

Using a personal repository on **Github**, I can make commits to changes I’ve made. Also, I have created a changelog that logs changes to the project every day with the code only time commitment for the day as well. Also, the local history extension in VSCode automatically backs up changed code to a **local folder** and stores the entire project in a folder in **Box**.

**Module List**

This project currently does not use any external modules apart from PIL and Python Default used in the 15-112 coursenotes.