Candidate Name	Candidate Number	Center Name	Center Number

PyATC

OCR Computer Science Project

Carlos Lagoa

INDEX

Topic Page

Analysis

Brief Description

In this section I will discuss a variety of topics regarding my project. Firstly I will talk about the project I'm doing.

My project is called PyATC, it is an air traffic control simulator written in Python with major graphical support from the Kivy library. Although there are already existing solutions I hope that mine will bring different usability options and therefore be a good alternative to existing air traffic control simulators.

The problem and the ease to solve it using a computational approach

The problem that I am aiming to solve is to provide an offline, free of charge software with learning capabilities that will teach anyone about the methods of handling the approaching aircraft in an airport. This problem can be solved using computational methods, starting from the ease of creating different environments to practice down to the complex task of having the same learning device and method as you progress (thanks to being able to change the difficulty) – which is more efficient than, say, having a book, which changes it's teaching and learning style as the user gathers more knowledge and he has to change books and therefore authors.

Identification of stakeholders

For the project I will have a certain number of stakeholders, the stakeholders will guide me in the process of creating, goal setting and evaluating my software. They will also have either problems with the current software available or with their knowledge, or lack thereof, regarding air traffic control.

My stakeholders will be members of the student body at my school who, for one reason or another, have decided to use my software (once it's finished). Below we can see a list of the stakeholders and their reason to be involved with the project.

Name	Involvement

Alan Harvey	Getting a flying license and uses air traffic control simulators
Matthew Beimborn	Interested in aviation but rather limited knowledge about it
Harry Dalton	Interest in how the software could be easier to learn than alternatives
Elliott Attew	Intrigued by aviation and interested in the friendliness of the project
Marlowe Ballan	Will try PyATC and see if he likes it, plays other computer games

What are the stakeholders interested in? - interviews to decide missing features and other problems with existing solutions

Whilst researching why other games where not the best option and why PyATC was needed it was decided to carry on a set of multiple choice questions for the 5 stakeholders in order to see where development of PyATC should be most focused on and what parts could not be given such a big priority.

1. Have you ever been interested in trying an air traffic control simulator?

*Yes **(4)*** *No **(1, Marlowe)***

2. If you have been interested in air traffic control simulators, why haven't you played them more? (please note, only 4 and not 5 can answer this question)

*Too easy/hard **(2)*** *Could of used the internet connection for something different **(1)***

*Other **(1)** - "my computer was too slow and getting to the website and playing it took time"*

3. If you haven't played PyATC, what was the main reason? (only Marlowe could answer this)

"The game seemed to complicated to begin with and tutorials were hard to follow"

4. Would you ever pay in order to have more functionality or tutorials?

*No **(3)** Yes **(0)** Only if the tutorials where very extensive and comprehensive and the price was very low **(1)***

5. How much do you value graphics when compared to a more or less engaging game play?

*As long as game is engaging graphics don't matter **(3)** Graphics are a top priority **(0)***

*I like graphics to be one of the priorities, but not the top one **(1)***

6. What computer do you typically use?

> *Desktop **(1)** Desktop and laptop **(2)** Laptop **(1)** Other **(1)***
>
> *Other... **(Harry Dalton)** "I use a laptop but when I'm home I also tend to use a Raspberry Pi; hooked up to my TV for tasks such as low spec games (Doom, N64 MarioKart...) and media purposes (films and music) - Over the last year I have used it so much that I have to give it credit as a daily computer"*

7. What operating system do you use?

> *Windows **(3)** Mac **(1)** Windows and Mac **(0)** Linux **(1)** Linux and Mac **(0)** Linux and Windows **(0)***

Why other software solutions will not fix the problem – Researching other solutions

There are a number of reasons PyATC was decided as the project and the large majority of them is to outline some of the lacking features that other simulators have.

Firstly, there are a number of ATC (air traffic control) simulators which require internet connection to operate and that can be seen as an inconvenience. Personally, I own a small computer and a carry it with me everywhere and so do most of the other PyATC stakeholders. There are parts of the day where the computer has no internet connection and the user has free time (perhaps waiting in an airport or being in a bus journey) – in those cases the laptop is there to be used but certain ATC simulators cannot be played simply because there isn't a stable internet connection.

Furthermore, and linking to the internet problem; nowadays an internet browsing software can be very demanding, for example Google Chrome has a tendency to use as much RAM as possible and that leaves the other tasks of the system almost unusable in some cases. A lower spec computer, such as the one being used to develop PyATC needs to divert to alternatives such as running Firefox or Opera, which don't use as many resources.

It is however not only the browsing software, if we decide to download an air traffic control simulator and have it as an executable file in our computers we will find that the file responds to the extension .exe, thus providing only Microsoft Windows computers with the ability to launch the software. Low spec computers (again, such as the one developing PyATC) have to run a Linux operating system in order to be usable to the extend of productivity and therefore cannot run any ATC simulator using an .exe - not even MacOS will be able to run an .exe file.

Secondly, there are a number of ATC simulators which are available offline. This solves the above problem however creates a new one – they cost money, an example can be seen in the following image.

The need to pay creates the problem whereby the user doesn't know if paying that amount of money is worth it and therefore might decide not to undergo the purchase, especially as he has never tried an ATC simulator before (a case which could determine whether or not the user plays the game at all)

Furthermore and perhaps the greatest problem is for the 'not-perfect' users. We could define a perfect user as a user who finds the game play engaging and challenging, this perfect player plays the game and by the time he has finished the session he gets a score which is considered as good but could be improved, for example an 75%. Unluckily the normal player is far away from a perfect user. The most typical players will either be a beginner and could find a session too difficult or an experienced player who needs something very challenging.

What does the research tell us about how PyATC should be

With this variety of problems that the stakeholders are experiencing a first design for PyATC was done;

- PyATC should be offline, so that it can be used anywhere

- PyATC should be free, so that people don't have any second thoughts about trying the software even if they think they won't like it

- PyATC should have the ability to change difficulty to keep a large pool of users interested and engaged.

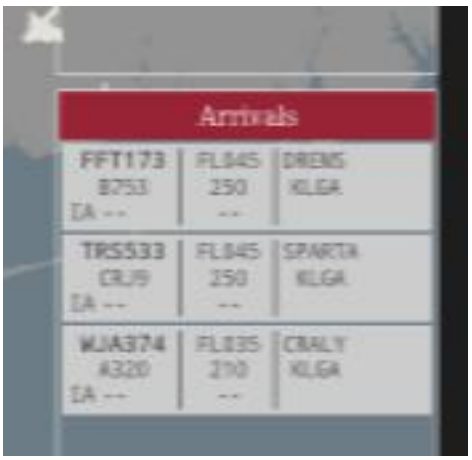
- PyATC should be multiplatform being able to run on multiple devices such as 11” touch screen Windows ultrabook, a powerful 32” desktop Mac Pro or even a Raspberry Pi

What will PyATC inherit regarding design and functionality from other solutions

Thanks to the research done, some items that other solutions had can be implemented to PyATC. For example, in the website www.atc-sim.com/simulator, we can see that there is very efficient user interface.

On the right hand side we can see that there is something which could be thought of as the control panel, with information regarding planes and their positions and also other items, such as a text box to communicate with planes. This design could be adapted to PyATC as it is a very intuitive solution round the problem of how to lay out and differentiate the real-time visual updates from other types of information and data intake.

Furthermore and regarding the way planes are classified we could take on from http://atcradarsim.com/simulator/klga/atc and lay them out based on how they have done it.



As we can see here, the aforementioned software solution has set out the information on boxes, with information such as the plane identifier, its speed, its altitude and heading and even information such as its arriving waypoint.

Limitations within PyATC

Now that a number of features have been determined based on lack of functions from other alternatives which have been researched we can define a framework for work so that we know what PyATC can and can't do and what the limitations might be.

PyATC is not aimed to be a realistic training software but a mere abstract simulation software so that people can understand and feel what it is like to manage arriving planes. Therefore we could say that one of the limitations of the project could be the realism it achieves. It should only be realistic to a certain extend; for example it should cover how planes need to maintain a safety distance, but it shouldn't be realist enough so that the distance depends on the type of planes interacting – if this was to be included the game play and learning process would be greatly increased and the valuable knowledge gotten out of the variable safety distance would be nothing compared to the time and effort put in initially.

Another fairly important limitation could be the cross platform availability. By developing with the Kivy library we can export the project to Windows, MacOS, Linux, Android and iOS – however due to the fact that the application will possibly need a keyboard as the communications interface that could limit the devices available to download the app. Therefore another problem for the application could be that it doesn't support all of the devices it could.

As a last point, PyATC will not count with a departing aircraft simulator, it will only have the approaching aircraft. This is because in reality a controller only takes care of either arriving or departing aircraft. However this can be seen as a limiting factor compared to other solutions, such as www.atc-sim.com/simulator which does in fact have both arriving and departing modes.

System Requirements

Now that the limitations of the project have been assessed we can discuss the minimum hardware and software requirements that the project will use-

Hardware	Software
Keyboard	Python 2.4
Mouse	Kivy 1.0
	Microsoft Windows XP, Mac OS X Puma or Linux 2.4.0

Why was Kivy chosen and not other alternatives

The reason why Kivy has been chosen, as opposed to any alternatives such as PyGame is because of two main reasons; the first one is the fact that while PyGame is purely for games, Kivy offers a varied UI library as well as the motion framework that you would expect to encounter in game development. Kivy allows for buttons, checked boxes, window transitions… whereas PyGame only allows for the creation of a rectangle and then one has to manually put text inside that rectangle and let it act as a button, this can be a problem when dynamically re-sizing a window.

Another reason for the choice of Kivy has to do with the UI and the way it renders and functions; it is built around the idea that touch input is very important and therefore supports and enhances its functionality when touch is used; furthermore since there are so many UI items and they are all so specific and well linked between each other and interact with its parent and child layouts the task of re-sizing a window is very dynamic and easy to handle. This will be good for modern smaller screen laptops.

What are the stakeholders interested in? - interviews to decide PyATC top priorities and other program features

Please note, all stakeholders have been informed about PyATC being offline, free and multi-platform, this set of interviews is purely based upon what should PyATC feature and how should it feature it.

1. In the title screen what should be displayed?

    Buttons taking you to places **(4)** A set of images that take you to other screens **(1)** The actual game, with the option to pause it and go to other screens **(0)**

2. Should you be able to detect incoming planes, departing planes or both?

    Only one of the options, whichever **(3)** Incoming planes **(2)** Departing **(0)** Both **(0)**

3. Would you prioritize having an extra airport to play on or an airport that served purely tutorial purposes?

    Extra airport **(1)** Tutorial based airport (simpler to understand; for practice) **(4)**

4. How would you keep track of difficulty within the game?

    Add countdown clock and see number of planes landed / departed **(3)** Have to land / depart a max numbers of airplanes **(2)** Unlimited time score, when plane has incident stop game and count points **(0)**

5. How many difficulties would you have?

    1**(0)** 2**(0)** 3**(4)** 4**(1)** 5**(0)**

6. Do you prioritize in-game aesthetics (colors, banners...) or an informative tutorial screen?

    In-game aesthetics **(0)** Tutorial Screen **(5)**

7. How long would you make a game, and how many levels would you set?

    Less than 10 mins, close to 30 mins and close to 1h **(4)**

    Less than 10 mins and close to 20 mins **(0)**

    10 mins, 20 mins and 30 mins **(1)**

8. If you have ever used an ATC simulator, what features did you like the most?

    It was very visual and I could see the planes moving **(2)**

    It was interesting to keep track of my process and see how I improved **(1)**

    I liked typing the input, rather than guiding a plane with my fingers or with voice **(0)**

9. What colors should be displayed?

    Contrasting colors; black background and red / yellow / orange / white items **(3)**

    Dual color palette; only white and black to aid simplicity **(1)**

    As real as possible; mostly green with red, white and black items **(1)**

10. Should the final score be composed of mistakes as well as successes?

    Yes **(4)** No **(1)**

Success Criteria

Based upon what has been researched, analyzed and discussed with stakeholders we can design a clear success criteria for PyATC. The following list is a recollection of what the majority of stakeholders thought was the best answer to the questions asked, as well as a requirements in relation to what was learned from the research.

PyATC Success Criteria

From a design point of view

a) The design will be modular, each screen will serve a purpose

b) There will be a title page whereby all features can be accessed

c) The screen will have an easy to read feel to it, with black background and contrasting items such as the runway and planes

d) There will be a variety of airports that the user can choose from

e) The planes in the screen will randomly appear and the user must guide them

f) The users will count with a real time view of what is going on, however all the information will be reachable through the side bar which will also have the remaining time and a pause button

g) Although not graphically extraordinary, the game should still represent the modern looks the Kivy UI library

h) Although not very complex graphically, the game should still be intuitive, with special emphasis on navigating the menus and engaging with the simulation

From a portability and usability point of view

a) The game should be able to run on Windows, macOS and Linux

b) The game will be usable for a variety of computer types (workstations, touchscreen laptops...) as it will detect and interact with touch

c) The game will be playable from different sizes; as it will dynamically adjust to different screen sizes

d) The game will be able to run from low-powered hardware (i.e. Raspberry Pi)

From a process point of view

a) The final score will be composed of user changeable values

b) The timing and location of arriving planes will be randomized within a framework based upon the selected difficulty

c) The player will only control planes that are incoming

d) The player will choose from a variety of difficulties, that will change how often planes are generated

SIGNED;

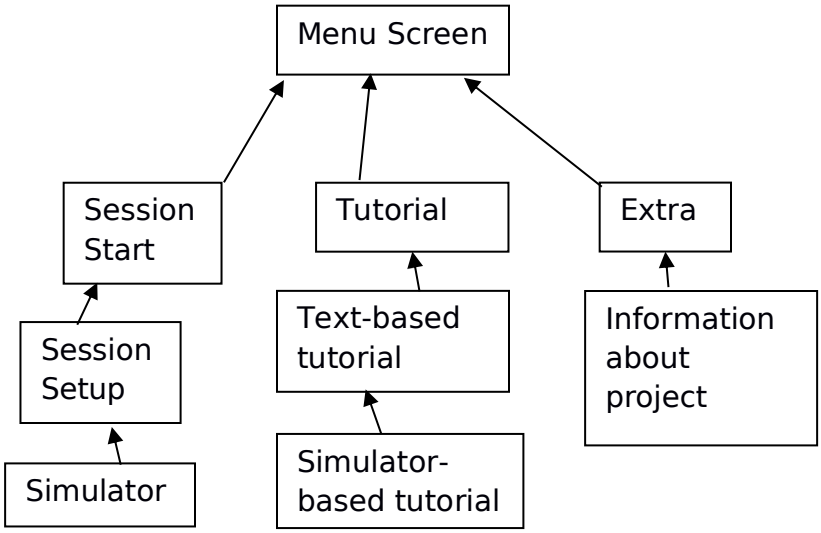
Alan Harvey, Matthew Beimborn, Harry Dalton, Elliott Attew and Marlowe Ballan

Design

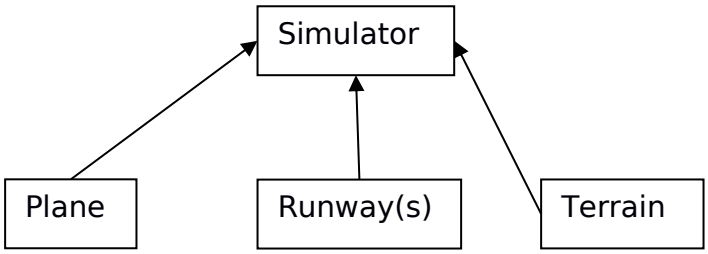
Design Methodology

Python can either be a procedural language or object orientated, because of the way Kivy works I will be programming in object orientated - and therefore I will have a variety of classes. Python defines not only items (perhaps a plane) as a class with certain properties; but also different screens, so as a screen moves from one to another so do the classes. In particular I will be using the following classes;

For the menus



For the simulation screen

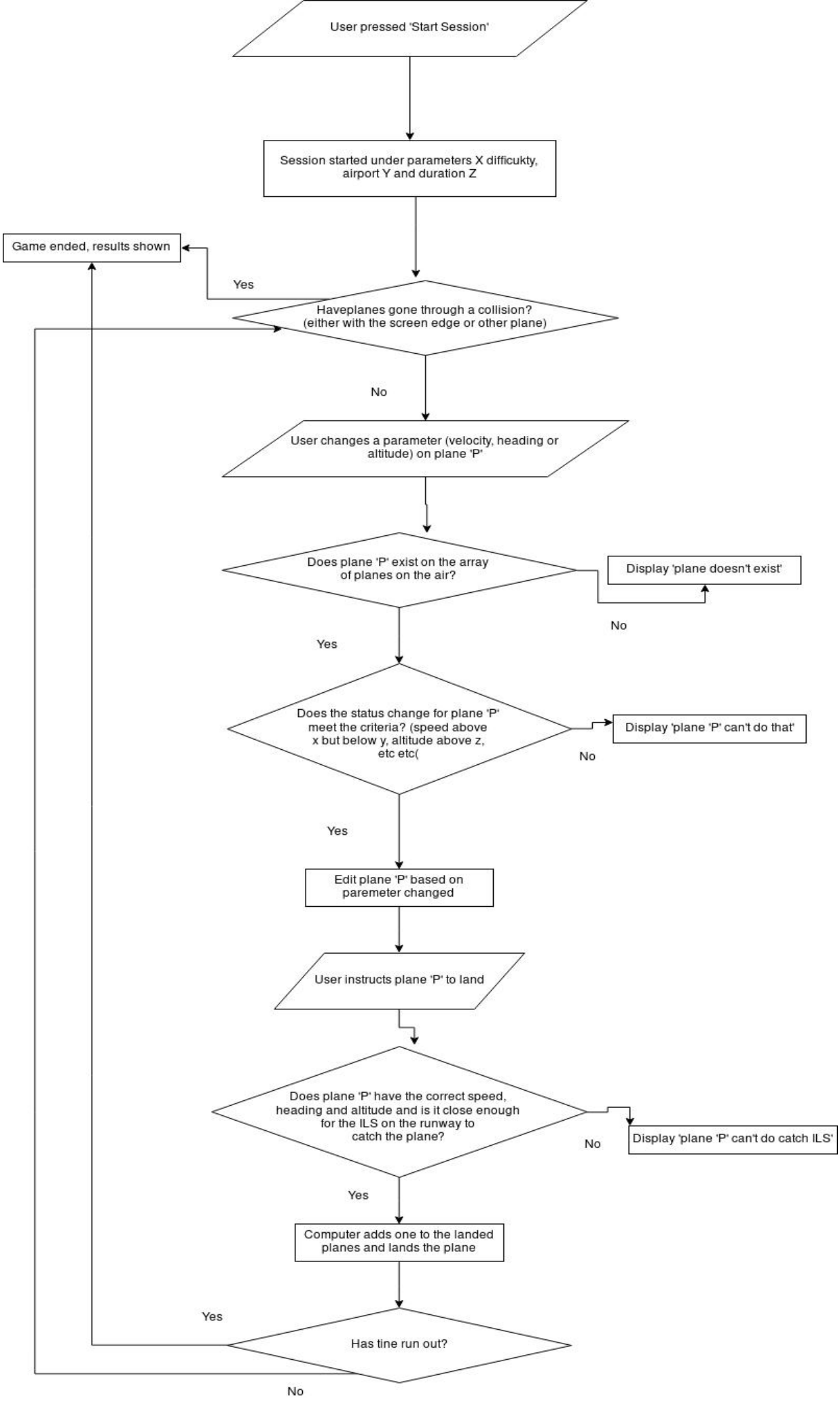


Main overview of code structure – pseudo code and flowchart

The following pseudo codes aim to describe the main functioning of the screens that will feature in PyATC.

I will now demonstrate the flow of the game using a flow chart diagram. This will only cover the button press ‘Start Session’ on the main screen and not the ‘Tutorial’ or ‘Extras’

On the next pages it will also be possible to see the pseudocode expected for the game logic and the way screens are structured in PyATC.


User pressed 'Start Session'
Session started under parameters X difficukty, airport Y and duration Z
Game ended, results shown
Yes
Haveplanes gone through a collision? (either with the screen edge or other plane)
No
User changes a parameter (velocity, heading or altitude) on plane 'P'
Does plane 'P' exist on the array of planes on the air?
Display 'plane doesn't exist'
No
Yes
Does the status change for plane 'P' meet the criteria? (speed above x but below y, altitude above z, etc etc(
Display 'plane 'P' can't do that'
No
Yes
Edit plane 'P' based on paremeter changed
User instructs plane 'P' to land
Does plane 'P' have the correct speed, heading and altitude and is it close enough for the ILS on the runway to catch the plane?
No
Display 'plane 'P' can't do catch ILS'
Yes
Computer adds one to the landed planes and lands the plane
Yes
Has tine run out?
No

```
#PyATC game screen, layout covered on IU layout section below, this wil lbe mostly logic
generate_plane:
        randomly generate a number and call it ‘x’, the range of numbers depends on difficulty
        if x = 1: generate new plane;      call it random(airline name, three digit number)
                                           give it speed random.range(200, 300)
                                           create it at screen edge at angle random.range(0, 360)
                                           give it altitude random.range(4000, 8000)
plane_landing:
        if plane is close to ILS catch point AND has speed 140 AND has altitude 1500:
                make computer take charge of plane
                print “plane caught ILS stream, will now land itself”

collision:
        if plane goes over screen edge OR plane and other plane are too close:
                go to game_over(Screen):

natural_game_over:
        if timer == ‘00:00’:
                go to game_over(Screen)
```

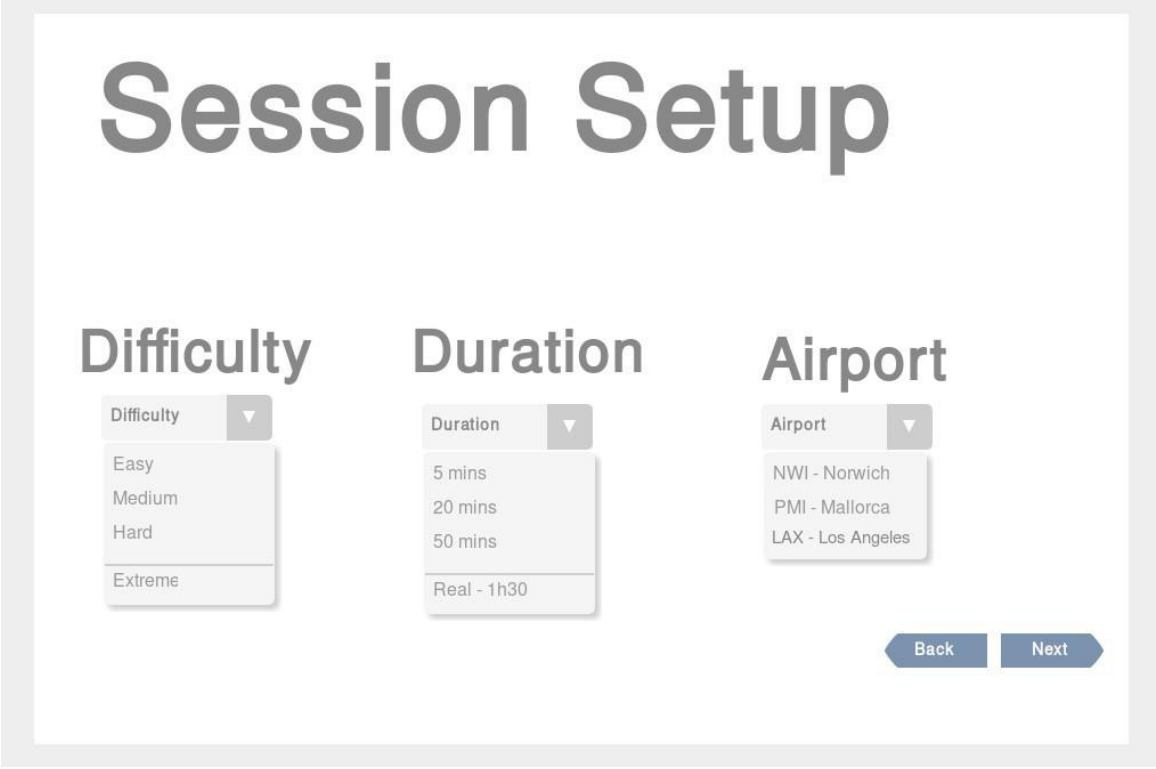
```
#PyATC pseudo code, main layout and linking between pages through buttons and program flow

main_menu(Screen)
        button = “Start Session”, when pressed go to session_setup(Screen)
        button = “Tutorial”, when pressed go to tutorial_text(Screen)
        button = “Extra”, when pressed go to extras(Screen)

session_setup(Screen)
        dropdown_menu = “Select airport…” - “Norwich Intl”, “Mallorca Intl”
        dropdown_menu = “Select difficulty” - “Easy”, “Medium”, “Hard”, “Extreme”
        dropdown_menu = “Select time” - “5mins”, “20 mins”, “1h”
        button = “Start Session”, when pressed go to simulation(Screen)

simulation(Screen)
        # due to the complexity of this screen I will devote a single pseudocode screen and UI
        # diagram later on

tutorial_text(Screen)
        text = “Welcome to PyATC, this program aims to help… the way to use it is… “
        button = “Practice in training airfield”, when pressed go to tutorial_simulation(Screen)

tutorial_simulation(Screen)
        # due to the complexity of this screen this has been covered before hand.

extras(Screen):
        text = “The highest scores achieved in each category so far are…”
        text = “This project is for the benefit of any user and in particular the stakeholders…”
        text = “The data structures used in this project are…”

game_over(Screen):
        # this screen appears when the simulation(Screen) ends, either through the user winning
        # or the user dying, The final score is composed of a number of variables discussed later.
        label = ”The game has finished, here is your score”, score, “\n and here your stats”, stats
        button = “Back to main menu”, when pressed go to main_menu(Screen)
```

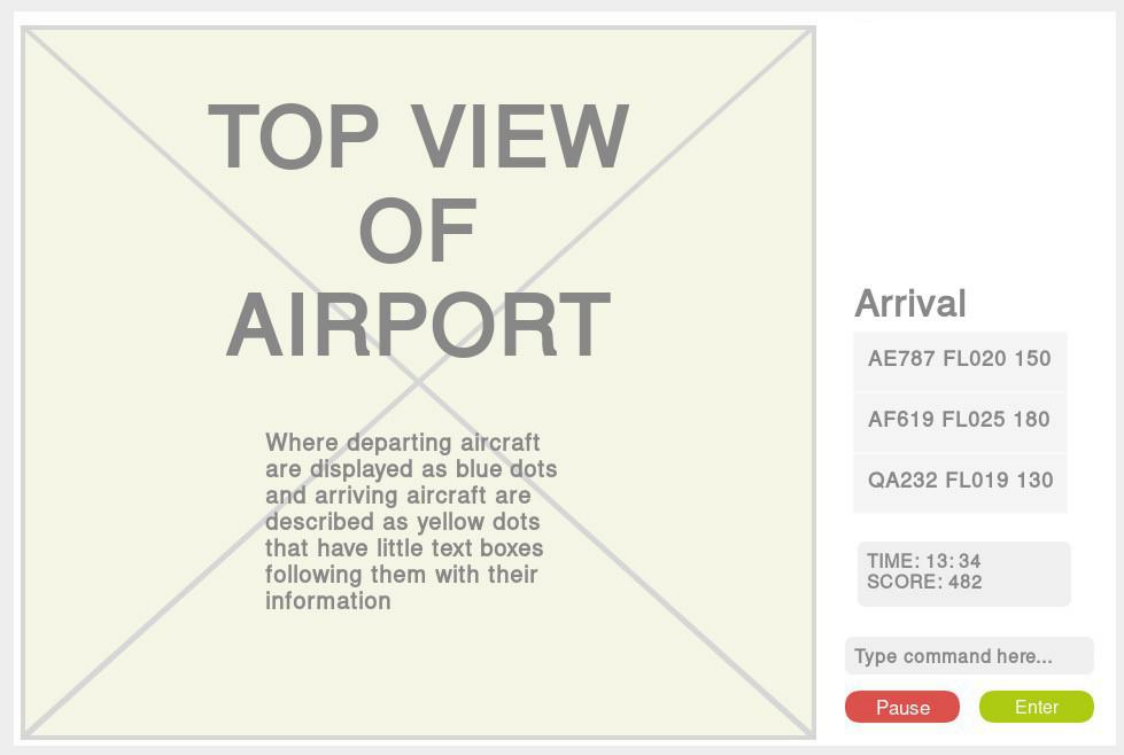
Main overview of code structure – visual

In the following pictures we will be able to see the prototypes for the UI, based on either self-made ideas or placing concepts based on research.

This is the PyATC main screen, when the program starts this screen is shown and all the possible actions are accessible right from the menu

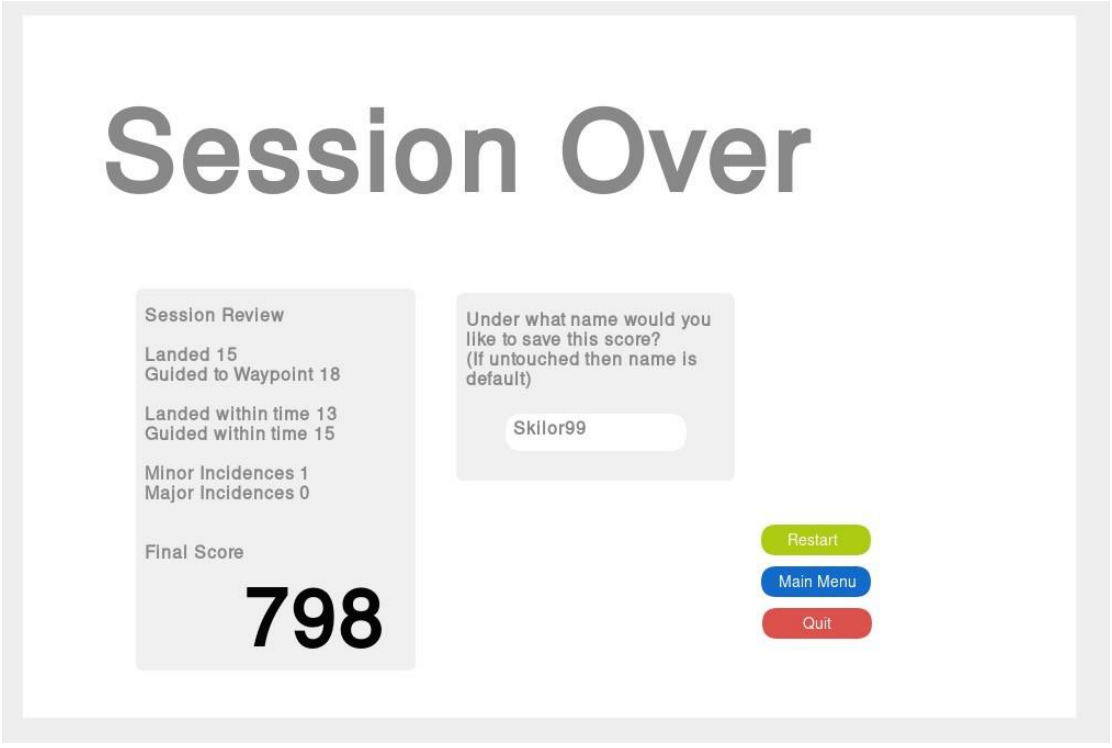
Once the user presses the Start Session button they are directed to this screen whereby they can choose the parameters for their session.





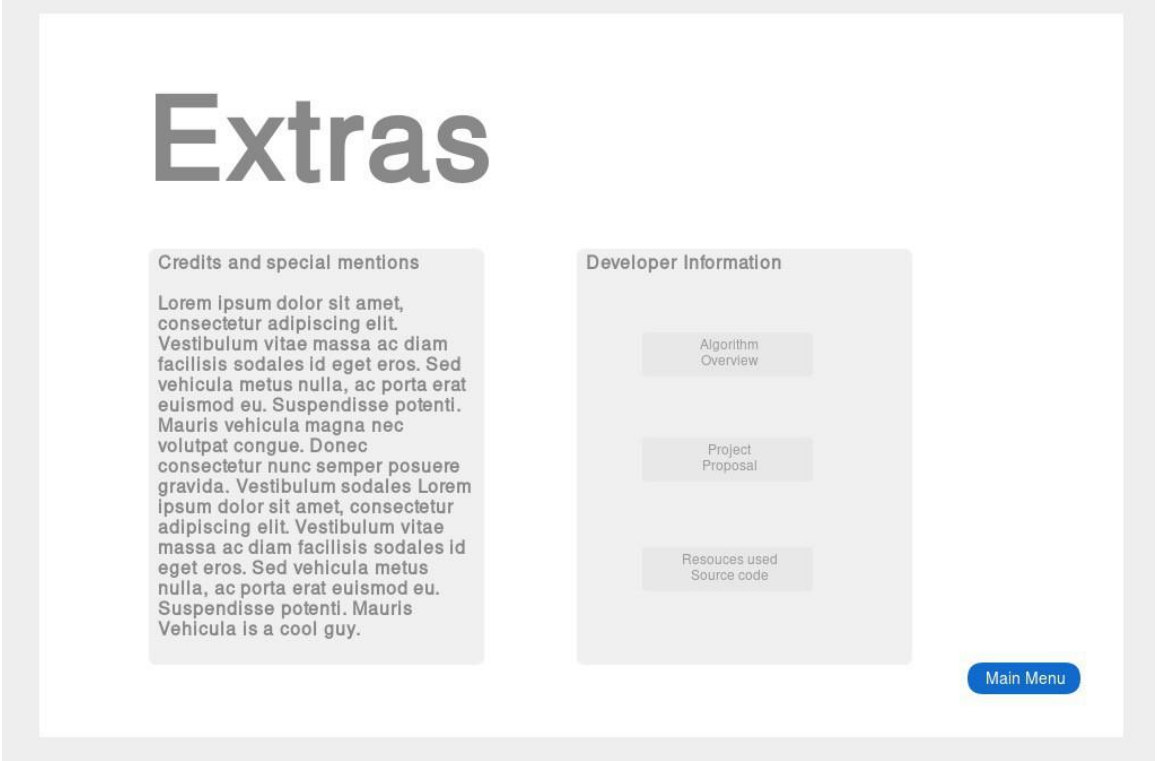
When the user starts the session he is presented with the simulation screen, on the left hand side the user counts with real time 2D visual aids of the position of the planes whereas on the right hand side the user can see the information and send messages back.

Either because the user has died or because he has successfully finished the session he will be taken to this screen, where he can see his information, final score and where he can save this



If the user however chooses to come to the tutorial screen and get guidance they can have lessons on how to perform certain tasks and then they can simulate those tasks and practice them in the practice airfield.

Furthermore, the user can choose to come here and get more information, the content of this page will be decided later on, but here is a basic overview of how it should look



Data Structures

The data that will be used for the proper utilization of the game can be divided into two lists; user-based and computer-created ones. The user-based ones would be ones that the need the interaction of the user to either be created or changed, whereas the computer-based one is simply data that the code steps through either sequentially or logically. For the planning purpose I will list examples of both data structures and then I will move on to give a general idea on how PyATC will use different data structures

User modifiable data structures

item	*data type*	*asked… /used…*	*basic description/use*
username	str	at end of session	to keep track of scores
game_length	array	session setup screen	ask how long the game is
game_difficulty	array	session setup screen	ask how difficult the game is
game_airport	array	session setup screen	ask where to simulate
ingame_landed	int	simulator screen	keep track of planes landed
ingame_warnings abnormalities	int	simulator screen	keeps track of traffic
ingame_crash	Boolean	simulation_screen	has the user made planes crash?

Software used static data structures

item	*data type*	*(suggested) value*	*basic description/use*
easy_plane_gen	array	range[100, 180]	how often planes will be generated
med_plane_gen	array	range[70, 140]	how often planes will be generated
hard_plane_gen	array	range[50, 100]	how often planes will be generated
xtreme_plane_gen	array	range[30, 70]	how often planes will be generated
easy_duration	int	300	seconds for one easy session (5mins)
med_duration	int	1200	seconds for one easy session (20mins)
hard_duration	int	3600	seconds for one easy session (1h)
xtreme_duration	int	5400	seconds for one easy session (1h30)

Generally PyATC would only contain the following data structures;

data structure	*basic information of usage*
integer	used for storing user score, random plane generation values etc..
string	used for storing texts such as tutorials or user names
array	used for storing the different difficulty levels
float	used for more extreme situations such as accurate layout

Development methodology

To develop PyATC I will use the Rapid Application Development methodology. This type of methodology features minimal planning in order to have rapid prototyping, thus allowing corrections to be made much quicker than if we spent time planning a version of the simulator.

RAD (Rapid Application Development) is beneficial because it aids the modular approach we are taking to the development of PyATC, whereby we develop abstract parts of the solution and then put them together once they work.

RAD is good for this project in particular because of the following reasons. Firstly, changing requirements can be accommodated, this means that if any decision or change where to be taken which would somehow change the content or aesthetics of PyATC this model could be integrated faster, thus furthermore giving a better image of whether the proposed change is of value or not.

Furthermore, the progress can be measured, this is beneficial as the project is very modular and could require the use the milestones in order to see where we are. The fact that the progress can be measured more easily also means that it will be simpler to go back to an early stage in case this is needed.

As one of the last points, RAD makes development time be reduced, this is because little planning is done and for certain projects (such as one with not many people working on it) this is beneficial as there is no need for communication upon planning or developing.

As a last point, by using this methodology we can encourage stakeholder feedback, this is extremely beneficial as it will be possible to always know whether or not the project has lost its initial aim.

How the project will be split into different parts - abstraction

PyATC development will be split into different smaller chunks to make the problem more approachable. This will aid the problem solving side of the development and furthermore will make troubleshooting easier.

The project can be abstracted into the following parts for development.

Item	Description
screen menu	This could focus on the development of the main screen
screen instruction	This could focus on the development of the instructions
screen extra	This could focus on the development of the extra screen
screen setup	This could focus on the development of the setup screen
screen game layout	This could focus on the layout aspect of the game screen
screen game levels	This could focus on making the time, difficulty & airports
screen game motion	This could focus on making the planes interactive

Dividing the development like that will ensure that any problem that comes up only appears in its section and therefore I can make sure that once its solved the simulator works just as it did before the error. The value of abstraction is essential in large projects.

Development and Version Testing

Version 0.1 - main aesthetics and menu navigation

As previously mentioned the development will be split into different sections. And therefore in this chapter each section will be separated. First we will start with the screen

menu. Because I am using the RAD model, I will be showing the progress I made on certain scenarios, but however in other scenarios (mostly GUI development) I will only show the final result, as there is no actual progress or extremely complex code to show or evolve

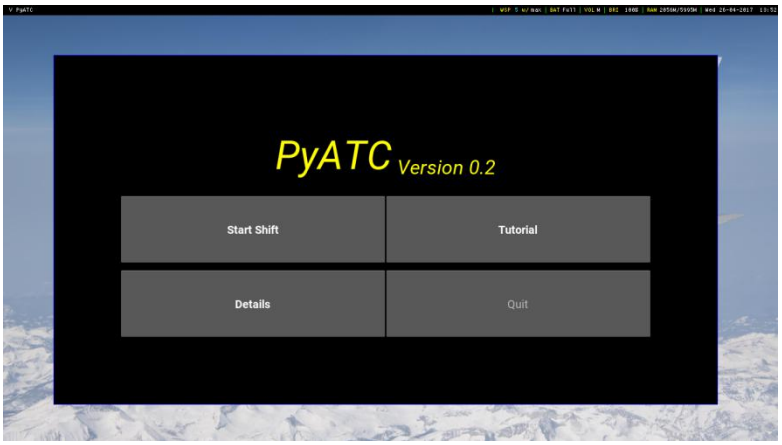
```
<MainMenu>
    BoxLayout:
        padding: self.width * 0.1
        spacing: self.height * 0.02
        orientation: 'vertical'
        Label:
            text: '[i][color=ffff00]PyATC[sub] Version 0.2[/sub][/color][/i]'
            font_size:70
            markup: True
        BoxLayout:
            Button:
                text: '[b][size=20]Start Shift[/size][/b]'
                markup: True
                on_release: root.manager.current = 'session_setup'
                on_release: root.manager.transition.direction = 'left'
            Button:
                text: '[b][size=20]Tutorial[/size][/b]'
                markup: True
                on_release: root.manager.current = 'tutorial_screen_text'
                on_release: root.manager.transition.direction = 'left'
        BoxLayout:
            Button:
                text: '[b][size=20]Details[/size][/b]'
                markup: True
                on_release: root.manager.current = 'about_screen'
                on_release: root.manager.transition.direction = 'left'
            Button:
                text: '[size=20][color=bbbbbb]Quit[/color][/size]'
                markup: True
                on_release: app.Exit()

```

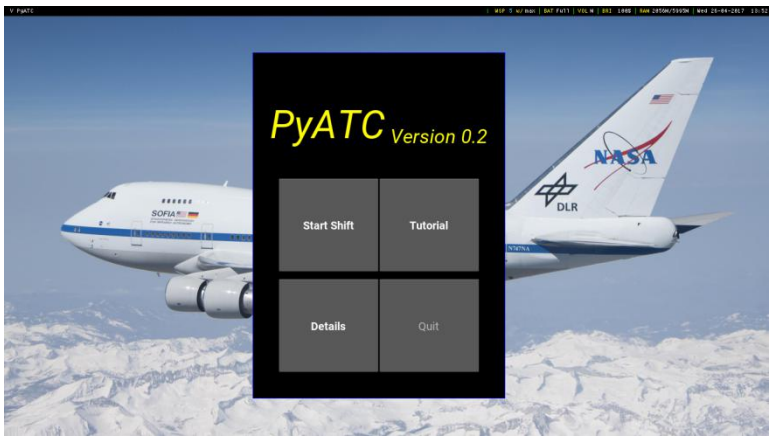
This is the styling code (Kivy separates styling and logic, how CSS and HTML works for example) here we can see how I use different indented layouts in order to add the correct spacing and layouts.

This class is called <MainMenu> and is later on referenced in the python code as a class, with a ‘pass’ as python doesn’t need to handle any logic from the screen.

This screen allows users to access different parts of the game through sleek window change animation. Here we can see how it looks.

As we can see, PyATC re-sizes automatically, this allows for a lot of different screen sizes to use PyATC.





Now I am going to talk about the instructions screen and how I did it, firstly I made a main layout which had three components (later on I reused this layout for other things). The three main components were:

- A button allowing to return to the main menu

- A button allowing to go to the next screen after a tutorial (trial run on easy airport)

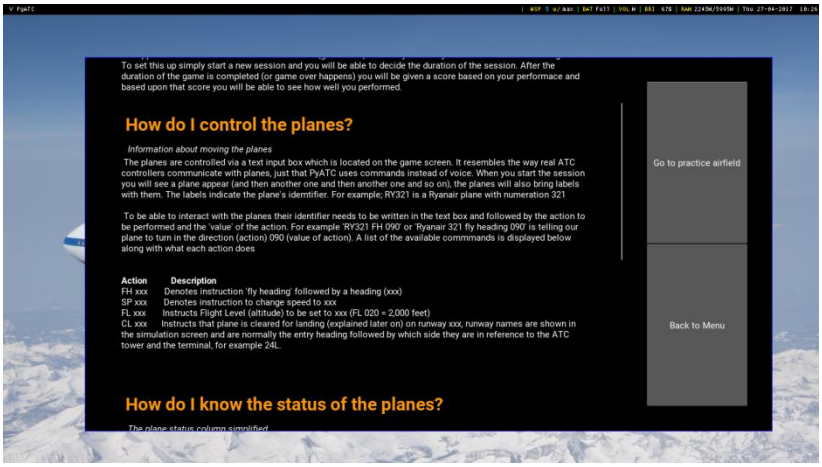
- A bigger section allowing me to put content on (the tutorial text in this case)

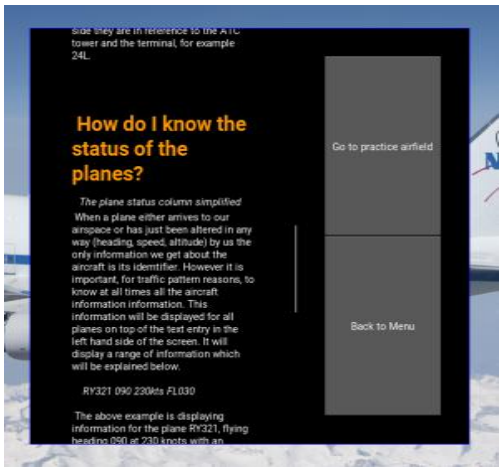
The way this was done was so that upon resizing the buttons wouldn't change shape, only the text would change shape in order to fit the new screen. Here is the code:-

```
<TutorialScreenText>
    GridLayout:
        rows: 1
        ScrollView:
            Label:
                size_hint_y: None
                height: self.texture_size[1]
                text_size: self.width, None
                id: paragraph_box
                markup: True
                padding: 60, 40

        BoxLayout:
            orientation: 'vertical'
            width: 250
            size_hint_x: None
            padding: 40, 40
            Button:
                text: 'Go to practice airfield'
                on_release: root.manager.current = 'session_screen'
                on_release: root.manager.transition.direction = 'left'
            Button:
                text: 'Back to Menu'
                on_release: root.manager.current = 'main_menu'
                on_release: root.manager.transition.direction = 'right'
```

Here we can see how we split up the layout into two parts, and then later on the second part of the layout was further split into two equally sized buttons. Initially the buttons where incredibly small but then, and with touchscreens in mind, I decided to make the buttons larger. Not only more aesthetically pleasing but also more efficient for any screen using this game.

Here is the result.



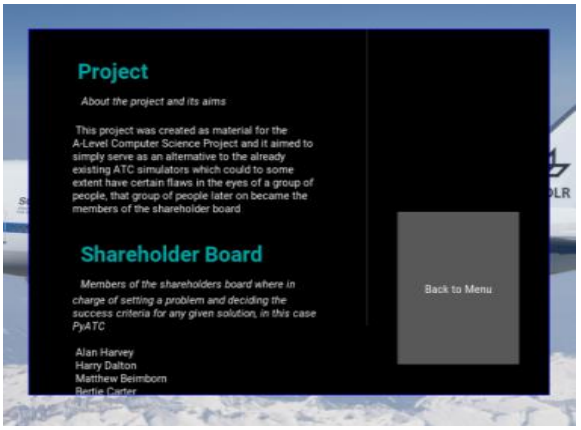


A very similar approach was done for the extras screen and the setup screen (where one chooses the parameters for the simulation session)

```
<AboutScreen>
    GridLayout:
        rows: 1
        ScrollView:
            Label:
                size_hint_y: None
                height: self.texture_size[1]
                text_size: self.width, None
                id: paragraph_box_about
                markup: True
                padding: 60, 40

        BoxLayout:
            orientation: 'vertical'
            width: 250
            size_hint_x: None
            padding: 40, 40
            Label:
                text:
            Button:
                text: 'Back to Menu'
                on_release: root.manager.current = 'main_menu'
                on_release: root.manager.transition.direction = 'right'
```

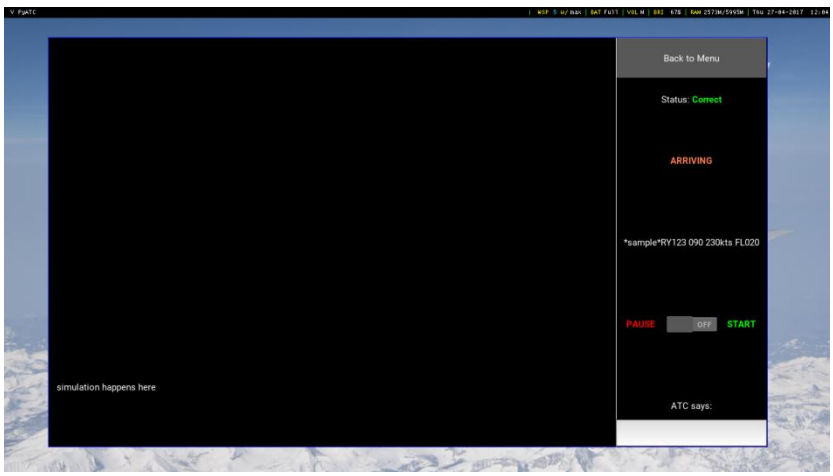
This is the extras screen, it displays other information such as who are the stakeholders and what the project is about. Because this screen can't take the user anywhere else, there is only a back button.

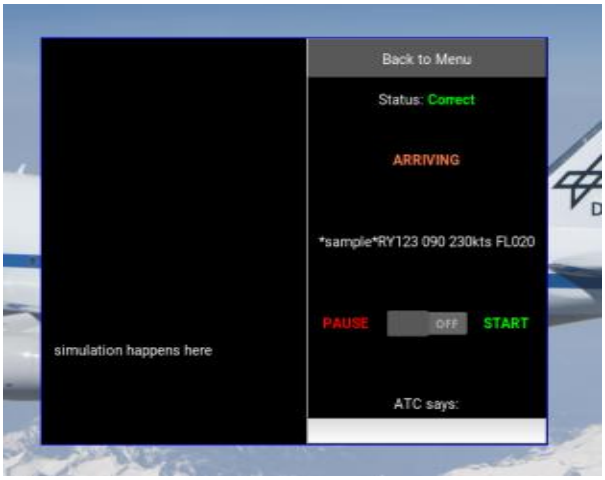
Here is how it looks:



After those main screen it was time to start developing the actual simulator, which is when the actual RAD development process started. To start with I made a very simple layout without any logic to it, this allowed me to prototype and then quickly change if I though things could look better. I vaguely based myself on the 'main-content-on-the-left, additional-content-on-the-right' design pattern I'd been following on the other screens. This was a good idea because it meant that design wise I didn't have to completely reinvent the wheel and furthermore, it was something that would look similar to the game users. Apart from using the same design as in other screens, I used the same ideology that the content pane on the right should be the one to stay the same size. This meant that I could confidently put all the important content on that right pane.

I started with a simple design:-






```
70      GridLayout:
71          rows: 1
72          Widget:
73              canvas:
74                  Rectangle:
75                      size: 1, self.height
76                      pos: self.width, 0
77              Label:
78                  text: 'simulation happens here'
79                  pos: (self.width/100*45), (self.height/100*48)
80
81          BoxLayout:
82              orientation: 'vertical'
83              width: 250
84              size_hint_x: None
85              BoxLayout:
86                  orientation: 'vertical'
87                  Button:
88                      text: 'Back to Menu'
89                      on_release: root.manager.current = 'main_menu'
90                      on_release: root.manager.transition.direction = 'right'
91                  Label:
92                      markup: True
93                      text: 'Status: [b][color=00ff00]Correct[/color][/b]'
94                      id: status_label
95              Label:
96                  text: '[color=ff7f50][b]ARRIVING[/b][/color]'
97                  markup: True
98              Label:
99                  text: "*sample*RY123 090 230kts FL020"
100
101                 id: departing
102             BoxLayout:
103                 orientation: 'horizontal'
104                 Label:
105                     markup: True
106                     text: '[b][color=ff0000]PAUSE[/color][/b]'
107                 Switch:
108                     id: pause_switch
109                     on_active: root.start_function()
110                 Label:
111                     markup: True
112                     text: '[b][color=00ff00]START[/color][/b]'
113             BoxLayout:
114                 orientation: 'vertical'
115                 Label:
116                     text: ''
117                 Label:
118                     text: 'ATC says:'
119                 TextInput:
120                     focus: True
121                     multiline: False
122                     id: input_box
```

Here we can see how the screen has been designed in the Kivy language.

At this point I started thinking of how the certain parts of the code could link into the python logic code, for example in line 98, where I linked the switch to a function in python so that the code would start a yet to be programmed clock cycle which made the planes move periodically.

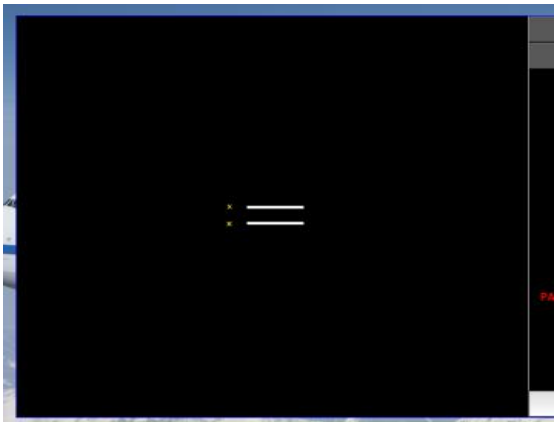
I also labeled with the ‘id: ‘ (example line 91) so that I would then be able to reference the Kivy styling sheet from Python and change its properties. In this case, my idea was to be able to change the label ‘departing’ depending on the planes that appear in the screen.

After designing this, I only needed to start working on the actual simulation part.

Beforehand however, the runways needed to be done; for this I will use a Kivy canvas (similar to an HTML canvas)

```
<SessionScreen>
    airplanes: planes_widget
    GridLayout:
        rows: 1
        Widget:
            canvas:
                Rectangle:
                    size: 80, 4
                    pos: (self.width/100*45), (self.height/100*48)
                Rectangle:
                    size: 8, 8
                    pos: (self.width/100*41), (self.height/100*47.5)
                    source: 'rsz_runway.png'
                Rectangle:
                    size: 9, 9
                    pos: (self.width/100*41), (self.height/100*51.68)
                    source: 'rsz_runway.png'
                Rectangle:
                    size: 80, 4
                    pos: (self.width/100*45), (self.height/100*52)
```

I also made the positions be according to the size of the screen so that when resizing the runways also resize, this is the result;



And with that we just have reached a milestone. Version 0.1 has been completed. It has a functioning menu and in-game screen layout, it is missing the logic and the addition of other airports and levels.

This was shown to the stakeholders, and a new small interview was carried on.

1. What do you think so far of the progress? Are you happy with the results and see the future of this project?

    Yes **(4)** I think what I see so far doesn't show me enough, I need to see the actual game **(1)**

2. Do you think the approach taken in terms of tutorials and user friendliness is acceptable?

    Yes **(3)** More work needs to be done on the tutorial screen **(1)** I don't think that matters, the user can choose how friendly he wants his game to be by choosing difficulty **(1)**

3. Is there anything that hasn't been discussed that you would like to see?

    The tutorial simulator should be far more approachable than any of the other screens **(1)** No **(4)**

After the meetings were done I was able to assess how the stakeholders felt about version 0.1 of the PyATC under RAD development methodology; overall positive and expecting to see progression in Version 0.2

Version 0.2 - initial motion testing

Version 0.2 focused mostly on the game dynamic and on creating the motion for the aircraft. To start on the motion, I started on a blank workspace, this blank workspace would allow me to experiment and create models, in this case there was a moving ball which would multiply once the screen was touched. The new appearing ball would be inside a class along with the other balls and have random values for direction and speed, once it touched the wall of the model the ball bounced back in the negative direction and speed.

```
class Ball(Widget):
    """Class for bouncing ball."""
    velocity_x = NumericProperty(0)
    velocity_y = NumericProperty(0)
    velocity = ReferenceListProperty(velocity_x, velocity_y)

    def update(self, dt):
        self.pos = Vector(*self.velocity) * dt + self.pos


class BallsContainer(Widget):
    """Class for balls container, a main widget."""

    def update(self, dt):
        balls = (c for c in self.children if isinstance(c, Ball))
        for ball in balls:
            ball.update(dt)

            # bounce of walls
            # (note: Y axis is pointing *up*)
            if ball.x < 0 or ball.right > self.width:
                ball.velocity_x *= -1
            if ball.y < 0 or ball.top > self.height:
                ball.velocity_y *= -1



    def on_touch_up(self, touch):
        """Touch (or click) 'up' event: releasing the mouse button
        or lifting finger.
        """
        ball = Ball()
        ball.center = (touch.x, touch.y)
        ball.velocity = (-MAX_BALL_SPEED + random() * (2 * MAX_BALL_SPEED),
                         -MAX_BALL_SPEED + random() * (2 * MAX_BALL_SPEED))
        self.add_widget(ball)
```

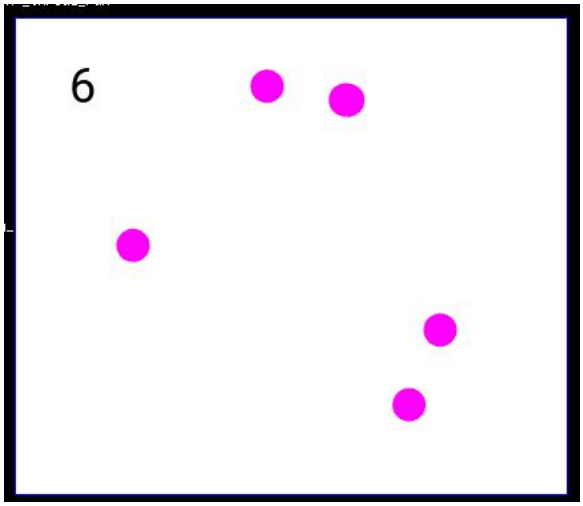
Here we can see a snapshot of the code I used, in it I have an update function which I will use in the PyATC simulator to have the planes moving like they would move in an actual Air Traffic Control booth (moving a couple of times per second and not contentiously; this happens because of the hardware refresh that the radar requires)

The on_touch function then adds a Ball of random properties. The ball class houses this functions and its properties if I ever wanted to display them - I used a class over an array as I found it easier to store a list of balls with certain properties this way.

After that and by using the Kivy language I was able to gather one of the ball class properties. Its overall number, furthermore I was able to put everything in a nice aesthetic thanks to the Kivy language.

The screen shots will be available next page.

```
<BallsContainer>:
    canvas:
        Color:
            rgb: 1, 1, 1
        Rectangle:
            pos: self.pos
            size: self.size
    Label:
        color: 0, 0, 0, 1
        top: root.top
        font_size: 36
        text: str(len(root.children) - 1)
```



After that was done I was able to give a certain direction and generate new balls under certain conditions, which meant I could translate this into the PyATC design.

Version 0.3 - motion implementation

Version 0.2 brought around the introduction to motion of the actual PyATC simulation screen, in version 0.3 I will work on the implementation into PyATC

For this I will use the original code I have and create a class called ‘SessionScreen’ and furthermore have implemented the movement we had on the blank workspace example.

```
def airspace_abandon(self):
    self.pause_switch.disabled = True
    Clock.unschedule(self.update_display)
    self.status_label.text = 'Status: [b][color=ff0000]Aircraft left radar[/color][/b]'

def update_display(self, dt):
    for plane in self.get_planes().values():
        plane.move(dt)
        if plane.x < 0 or plane.right > self.width - 250 or plane.y < 0 or plane.top > self.height: self.airspace_abandon()
```

In this example we can see how the update function has an update cycle where all the planes move, however this update loop can be broken with the conditional statement that appears below it; in it it doesn’t allow planes to leave certain boundaries; one of them not being the edge of the screen but rather the edge of the control center pane on the right hand side of the screen which, just as we planned it, will always be 250px wide and therefore we can set the condition with just 250px rather than actually referencing the

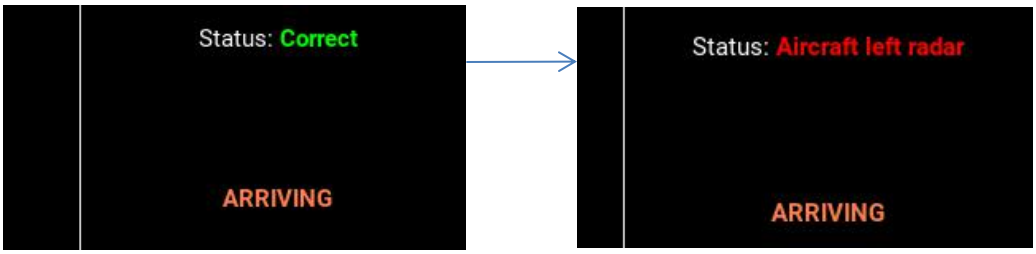
pane which is something that brought around problems. When one of the planes has one of these conditions the game clock (used in the update class) was stopped

As it happened in the blank workspace, there was also a class for the moving objects, but in this case the class was called ‘Airplanes’.

```
326 class Airplane(Widget):
327     angle = NumericProperty(0)
328     speed = NumericProperty(3)
329
330     def move(self, dt):
331         velocity = (math.sin(self.angle) * self.speed, math.cos(self.angle) * self.speed)
332         self.pos = (Vector(velocity) * dt) + self.pos
333
```

As we can see the in the screenshot above the plane determines its movement by multiplying its current position times its velocity at every cycle; and the velocity is determined by the angle we give, at the time being when planes render they all have the same angle, but in the future this angle will be different for each plane render.

Now we have the ability to have a plane moving under a control speed and within a boundary; below we can see a screenshot of how this works within the game



And with that Version 0.3 was finished, it was time to start Version 0.4

Version 0.4 - addition of functionality to game screen

Version 0.4 introduced the use of commands to interact with planes to a basic level, it added two new functions within the ‘SessionScreen’ class; ‘get_planes’ and ‘on_command’

The ‘on_command’ function gathered and split the string that the user had used as input once he pressed enter, furthermore the ‘get_planes’ command was used for being able to gather all the planes that appeared into screen.

After that I also added a ‘plane_creation’ function, which added the planes to the screen and gave them appropiate naming.

Here is the ‘on_command’ function :-

```
349     def on_command(self):
350         command = self.input_box.text
351         self.input_box.text = ''
352 
353         plane_referenced = command[:5]
354         plane_instruction = command[6:8]
355         plane_instruction_value = command[9:12]
356 
357         planes = self.get_planes()
358 
359         if not plane_referenced in planes:
360             self.status_label.text = "not on list"
361             time.sleep(3)
362 
363         plane = planes[plane_referenced]
364 
365         if plane_instruction == 'FH':
366             # flight heading
367             # TODO: handle them putting in not a number
368             angle = int(plane_instruction_value) * math.pi / 180
369             plane.angle = angle
370         elif plane_instruction == 'SP':
371             # speed of plane
372             # TODO: handle them putting in not a number
373             speed = int(plane_instruction_value)
374             plane.speed = speed
375         else:
376             # TODO: tell them that command doesn't exist
377             return
```

on_command is triggered when the input box in the game screen receives an enter as its input, when that happens on_command gets and stores the value (line 350), and clears the text box (line 351), after that it splits the text into which plane is being referenced, for what exactly and what is the value of change the airplane should undergo (lines 353, 354 and 355).

If the plane being referenced is not part of the list a message appears informing the user that there is no such plane (line 357 to 361).

After that a series of conditional statements convert the input into one of the possible instructions that the user could desire (lines 365 to 377). Please keep in mind that this code will be refined later on.

After working on the ‘on_command’ function I worked on the ‘get_planes’ function. This function fetches a plane from the class which stores the planes, it is incredibly useful when trying to reference planes like for example in the on_command class. The class returns nothing if the plane doesn’t exists. In the next page we will be able to see a screenshot of the code and the parts that need explaining.

```
392
393     def get_planes(self):
394         # IMPORTANT: assumes all planes have unique names
395         planes = {}
396         for child in self.airplanes.children:
397             if not isinstance(child, Airplane):
398                 # not a plane
399                 continue
400             planes[child.name] = child
401         return planes
```

So far we can see the one flaw is simply the fact that the random generator might create planes with exact names however this can be fixed later on. Here we can see how we make the planes into an array and we check if what we are looking for is in the plane list

After this function I also did was the ‘plane_creation’ function, this function allowed me to automatically create airplanes with a given random name within my desired parameters. Here is the code:

```
385     def plane_creation(self):
386         # create aircrafts
387         airplane = Airplane()
388         airplane.angle = math.pi / 6
389         airplane.speed = 2
390         airplane.name = random.choice(['AF', 'RY', 'VY']) + str(random.randrange(999)).zfill(3)
391         self.airplanes.add_widget(airplane)
```

`

Here we can see how this function simply gives a new plane the basic information it needs in order to be functional and interactive with the user, in the near future I will make this function run at a random time frame based on the difficulty selected by the user.

And with the addition of those 3 important functions Version 0.4 was finalized.

Version 0.5 - further game screen development and bug fix

Version 0.5 brought the creation of the ‘start_function’ function, these function added functionality to the START-STOP switch on the control pane on the right hand side of the game screen in PyATC.

```
379
380     def start_function(self):
381         if self.pause_switch.active:
382             Clock.schedule_interval(self.update_display, 1 / 3)
383         else:
384             Clock.unschedule(self.update_display)
385
```

After that I could make PyATC stop and start when ever I wanted. During the development of this I had an extremely hard to find issue with the code;

```
Switch:
    id: pause_switch
    on_active: root.start_function()
```

This is the kivy language code for the switch, which when changed of its current state, sends a callback function to the python logic code which then triggers it to either start or stop the game.

However up until Version 0.5 I used ‘on_release: root.start_function()’, this lead to the game starting and stopping immediately and I spent time troubleshooting. The way I found that the the problem was the action detector was by printing the action of the button pressed everytime it was pressed. When using ‘on_release’ it came up twice in one go (start and stop). This is a bug in Kivy. To solve it I used another action receiver; ‘on_active’

Output with single press on on_release

```
Start-Stop triggered
Start-Stop triggered
```

Output with single press on on_active

```
Start-Stop triggered
```

After that I also worked on the mid-air collision module, this meant the creation of a module similar to the ‘abandon_airspace’ module. Here are the two modules, and below we can see how they are triggered.

```
410     def airspace_abandon(self):
411         self.pause_switch.disabled = True
412         Clock.unschedule(self.update_display)
413         self.status_label.text = 'Status: [b][color=ff0000]Aircraft left radar[/color][/b]'
414
415     def airspace_collision(self):
416         self.pause_switch.disabled = True
417         Clock.unschedule(self.update_display)
418         self.status_label.text = 'Status: [b][color=ff0000]Mid-air collision[/color][/b]'
419
```

Here is the logic behind the trigger of the airspace_collision:-

```
420     def update_display(self, dt):
421
422         for plane in self.get_planes().values():
423             plane.move(dt)
424             if plane.x < 0 or plane.right > self.width - 250 or plane.y < 0 or plane.top > self.height:
425                 self.airspace_abandon()
426             for other_plane in self.get_planes().values():
427                 if other_plane == plane:
428                     continue
429                 diff_x = plane.x - other_plane.x
430                 diff_y = plane.y - other_plane.y
431                 diff_height = plane.height - other_plane.height
432                 distance = math.sqrt((diff_x**2) + (diff_y**2) + (diff_height**2))
433                 if distance < 50:
434                     self.airspace_collision()
```

As we can see here the game checks every update for a collision event. It runs through the list of planes (line 426) and discards checking the proximity with itself (lines 427 and 428) after that it follows Pythagoras theorem logic in order to have a 3 dimensional ‘personal space’ which when intruded by another aircraft jumps to the airplane_collision function. That leads to a game over.

Apart from that Version 0.5 introduced extensive error checking and user feedback when inputting characters. This was built on top of an existing piece of code.

```
369         plane = planes[plane_referenced]
370
371         if plane_instruction == 'FH':
372             if isinstance(plane_instruction_value, int) or (plane_instruction_value in range(360)):
373                 self.status_label.text = "Status: [b][color=ff0000]Heading given cannot be done[/color][/b]"
374                 return
375             angle = int(plane_instruction_value) * math.pi / 180
376             plane.angle = angle
377         elif plane_instruction == 'SP':
378             if isinstance(plane_instruction_value, int) or (plane_instruction_value in range(4)):
379                 self.status_label.text = "Status: [b][color=ff0000]Speed given cannot be done[/color][/b]"
380                 sreturn
381             speed = int(plane_instruction_value)
382             plane.speed = speed
```

Here we can see how there was been an addition to the on_command function, in here we check for the input given to be an integer and between some range of acceptable integers (0 to 360 for the heading for example) (lines 372 and 378). If that occurs then the user sees a notification message appear in red that informs him of the error that has occurred.

With those fixes the Version 0.5 was finalized and a new version was started. As the final project was starting to gain the final desired shape the stakeholders where called in for another meeting to access their opinions. The questions went overall well which shows how using RAD and diving the work into versions was a good idea.

1. What was your main opinion of the game?, knowing of course that bits were missing

    I like it and I see if its potential **(4)** I think the game needs more work **(1)**

2. Can you see the game as something that you don't understand and that you would like to learn about?

    When I look at the game it looks complex and I'm intrigued to learn about it **(3)** I understand what is going on but I thing I would still enjoy the game **(1)** I do not understand the game at all and I don't see how I can understand it **(1)**

3. How are the aesthetics of the game? Do you have any problem navigating the menus?

    No **(5)** Yes **(0)**

Version 0.6 - addition to game dynamic and aircraft referencing

Version 0.6 introduced a couple of minor additions to the code. The altitude for example was introduced as one of the planes attributes and it was given the boundaries it needed in order to be able to change it.

```
451             heading = str(int(plane.angle * 180 / math.pi)).zfill(3)
452             fixed_speed = str(plane.speed * 100).zfill(3)
453             fixed_altitude = str(plane.altitude).zfill(3)
454             line = "%s %s %s FL%s" % (plane_name, heading, fixed_speed, fixed_altitude)
455             plane_lines.append(line)
```

This is the code that is in charge of keeping the user informed with the information from the aircraft. The heading, speed and altitude are displayed in the form of a single string of text. In this case, line 453 shows the way the heading is 'fixed' so that the user can see it properly by always being displayed as a 3 number string (30 shows as 030 for example)

```
326  class Airplane(Widget):
327      angle = NumericProperty(0)
328      speed = NumericProperty(3)
329      altitude = NumericProperty(30)
```

This screenshot simply denotes the introduction of the altitude attribute to the Airplane class.

```
383      elif plane_instruction == 'FL':
384          if isinstance(plane_instruction_value, int) or (plane_instruction_value in range(2, 30)):
385              self.status_label.text = "Status: [b][color=ff0000]Altitude given cannot be done[/color][/b]"
386              return
387          altitude = int(plane_instruction_value)
388          plane.altitude = altitude
```

Furthermore here we can see how PyATC now treats the input of a FL (Flight Level, altitude) change. It again goes through the logic of determining the upper and lower boundaries of the possibilities of altitude change and confirms that the input given is in fact an integer value (line 384), if the user followed the correct criteria whilst writing the command then the written altitude is the new altitude.

After that, and expanding on what one of the screenshots was carrying I also worked on the addition of the side bar displaying all the pertinent airplane information.

```
448          plane_lines = []
449          for plane_name, plane in self.get_planes().items():
450              # convert from radians to bearing
451              heading = str(int(plane.angle * 180 / math.pi)).zfill(3)
452              fixed_speed = str(plane.speed * 100).zfill(3)
453              fixed_altitude = str(plane.altitude).zfill(3)
454              line = "%s %s %s FL%s" % (plane_name, heading, fixed_speed, fixed_altitude)
455              plane_lines.append(line)
456          self.departing.text = "\n".join(plane_lines)
457
```

Here we can see how we display a single line of string for every plane in the list and how we display its properties. This code is in the update function and therefore updates once the user has made a change. Here we can see it working.

As we can see the planes are displayed with their attributes; a heading of 029, a speed of 300kts and an altitude of 2000 ft.

After that the last minor thing I fixed was the ability to write the speed of the planes in kts and not in its relative pixel unit. Up to now the planes were moving at pixels*3 / second. This means that if the user wrote 2 then the planes moved at a comfortable rate, but of course if the user wrote 200kts which is what he should of been writing the game would be over instantaneously as the planes would of left the airspace. To fix this I changed the input that the user gave to the computer by diving it by 100

```
        return
    speed = int(plane_instruction_value) / 100
    plane.speed = speed
```

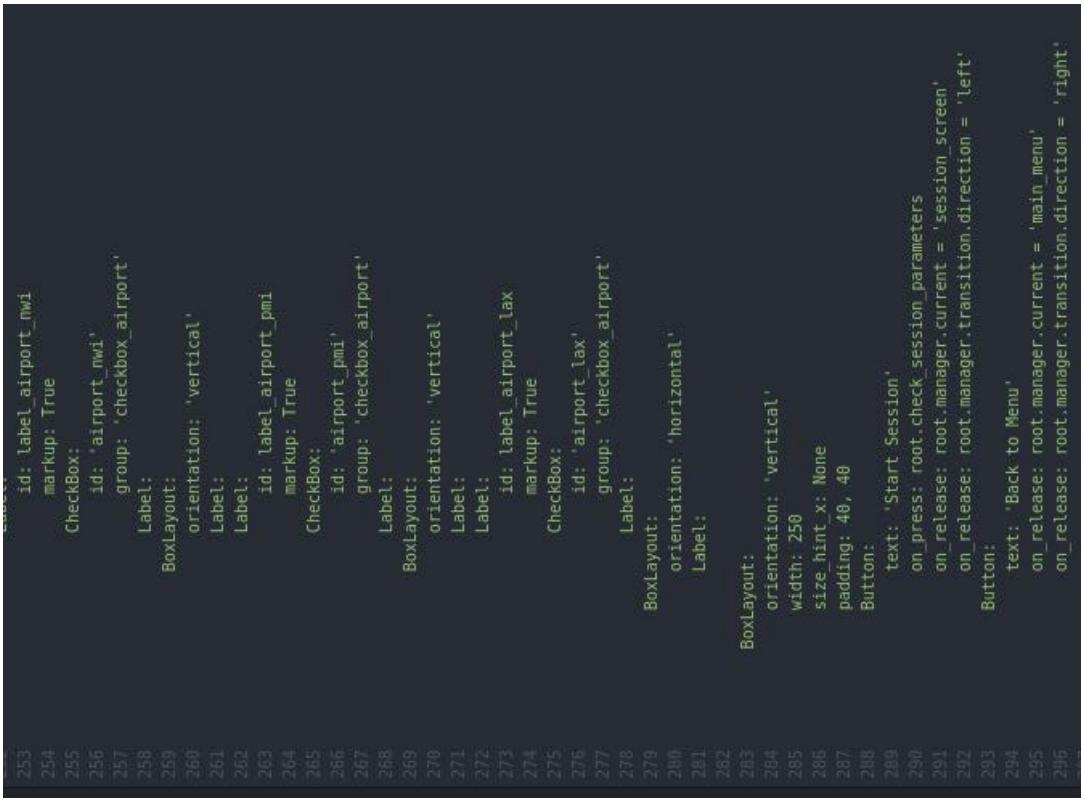
This example really outlines the usefulness of having a class to represent planes as changing the plane.speed changes it whereas in an array it would be harder to do.

The major change that occurred in PyATC was the introduction of the setup screen. This screen serves the sole purpose of giving the user the options of choosing his what he desires for the game that he is about to play. This is time, difficulty and location. For starters I will make the layout in the Kivy language and then in the next versions I will add its functionality.

The code is very simple to understand and will take a page worth of space. It will not be explained in detail as it simply layout code and it is easy to understand and read; but pretty much what it does is give the user a set of options for each category, as well as a side panel with access to buttons to continue to the game screen or to be able to go back to the menu.

```
<SessionSetup>
    GridLayout:
        rows: 1
        BoxLayout:
            orientation: 'vertical'
            BoxLayout:
                orientation: 'horizontal'
                Label:
                    id: label_duration
                    markup: True
                BoxLayout:
                    orientation: 'vertical'
                    Label:
                    Label:
                        id: label_duration_5mins
                        markup: True
                    CheckBox:
                        id: duration_5mins
                        group: 'checkbox_duration'
                    Label:
                BoxLayout:
                    orientation: 'vertical'
                    Label:
                    Label:
                        id: label_duration_20mins
                        markup: True
                    CheckBox:
                        id: 'duration_20mins'
                        group: 'checkbox_duration'
                    Label:
                BoxLayout:
                    orientation: 'vertical'
                    Label:
                    Label:
                        id: label_duration_1h
                        markup: True
                    CheckBox:
                        id: 'duration_1h'
                        group: 'checkbox_duration'
                    Label:
            BoxLayout:
                orientation: 'horizontal'
                Label:
```

```
            BoxLayout:
                orientation: 'horizontal'
                Label:
                    id: label_difficulty
                    markup: True
                BoxLayout:
                    orientation: 'vertical'
                    Label:
                    Label:
                        id: label_difficulty_easy
                        markup: True
                    CheckBox:
                        id: 'difficulty_easy'
                        group: 'checkbox_difficulty'
                    Label:
                BoxLayout:
                    orientation: 'vertical'
                    Label:
                    Label:
                        id: label_difficulty_medium
                        markup: True
                    CheckBox:
                        id: 'difficulty_medium'
                        group: 'checkbox_difficulty'
                    Label:
                BoxLayout:
                    orientation: 'vertical'
                    Label:
                    Label:
                        id: label_difficulty_hard
                        markup: True
                    CheckBox:
                        id: 'difficulty_hard'
                        group: 'checkbox_difficulty'
                    Label:
            BoxLayout:
                orientation: 'horizontal'
                Label:
                    id: label_airport
                    markup: True
                BoxLayout:
                    orientation: 'vertical'
                    Label:
                    Label:
                        id: label_airport_nwi
```



```
                        id: label_airport_nwi
                        markup: True
                    CheckBox:
                        id: 'airport_nwi'
                        group: 'checkbox_airport'
                    Label:
                BoxLayout:
                    orientation: 'vertical'
                    Label:
                    Label:
                        id: label_airport_pmi
                        markup: True
                    CheckBox:
                        id: 'airport_pmi'
                        group: 'checkbox_airport'
                    Label:
                BoxLayout:
                    orientation: 'vertical'
                    Label:
                    Label:
                        id: label_airport_lax
                        markup: True
                    CheckBox:
                        id: 'airport_lax'
                        group: 'checkbox_airport'
                    Label:
            BoxLayout:
                orientation: 'horizontal'
                Label:
            BoxLayout:
                orientation: 'vertical'
                width: 250
                size_hint_x: None
                padding: 40, 40
                Button:
                    text: 'Start Session'
                    on_press: root.check_session_parameters
                    on_release: root.manager.current = 'session_screen'
                    on_release: root.manager.transition.direction = 'left'
                Button:
                    text: 'Back to Menu'
                    on_release: root.manager.current = 'main_menu'
                    on_release: root.manager.transition.direction = 'right'
```

After this the screen looked like this.

And with that screen and the small fixes that was Version 0.6 done.

Version 0.7 - clock functionality, class inheritance and score screen

Version 0.7 brought the introduction of the clock to the PyATC simulator. Because the clock update function that I had until know updated 3 times per second I decided to make a new update function where I laid the foundation of the clock.

Firstly I made a clock layout by using a label on Kivy :-

```
96          BoxLayout:
97              orientation: 'vertical'
98              width: 250
99              size_hint_x: None
100             BoxLayout:
101                 orientation: 'vertical'
102                 Button:
103                     text: 'Back to Menu'
104                     on_release: root.manager.current = 'main_menu'
105                     on_release: root.manager.transition.direction = 'right'
106                 Label:
107                     text: '00:00'
108                     id: countup_label
109                 Label:
110                     markup: True
111                     text: 'Status: [b][color=00ff00]Correct[/color][/b]'
112                     id: status_label
113             Label:
114                 text: '[color=ff7f50][b] A R R I V I N G [/b][/color]'
115                 markup: True
116             Label:
117                 text: 'no arriving flights'
118                 markup: True
119                 id: departing
```

Here we can see how in line 106, 107 and 108 we are creating a label which will originally have the text of ‘00:00’ but which later on will of course change into the clock.

```
393     def start_function(self):
394         if self.pause_switch.active:
395             Clock.schedule_interval(self.update_display, 1 / 3)
396             Clock.schedule_interval(self.update_display_clock, 1 / 1)
397             self.status_label.text = 'Status: [b][color=00ff00]Correct[/color][/b]'
398         else:
399             Clock.unschedule(self.update_display)
400             Clock.unschedule(self.update_display_clock)
```

Here we can see in line 396 how we start the clock function for the game (3 updates per sec) and the clock function (1 update per sec)

On the next page we will be able to see how the code is used to make the clock count upwards in the clock update function

```
def update_display_clock(self, dt):
    # clock
            if self.pause_switch.active == True: # If the switch is off
                if self.zero < 60 and len(str(self.zero)) == 1: ## '00:0'
                    self.countup_label.text = '0' + str(self.mins) + ':0' + str(self.zero)
                    self.zero += 1

                elif self.zero < 60 and len(str(self.zero)) == 2: ## '00:'
                    self.countup_label.text = '0' + str(self.mins) + ':' + str(self.zero)
                    self.zero += 1

                elif self.zero == 60:
                    self.mins += 1 # Add a minute
                    self.countup_label.text = '0' + str(self.mins) + ':00' # EG '01:00'
                    self.zero = 1 # Reset zero to start again for a new minute
```

This is how the code looks, as we can see it is fairly straight forward; we update each value of the clock when the value to its right reaches a certain number. for example

x1	x2		x3	x4
0	0	:	0	0

00:01... 00:02... 00:03.. (when x4 reaches 9 update x3 to 1 and x4 to 0)... 00:09... 00:10...

After that I made the input from the user be defined as the maximum time before the game ended.

```
def update_display(self, dt):

        if self.session_setup.duration_5mins.state == 'down':
                self.duration_for_session = 5
        elif self.session_setup.duration_20mins.state == 'down':
                self.duration_for_session = 20
        elif self.session_setup.duration_1h.state == 'down':
                self.duration_for_session = 60
```

Here we can see how the the Kivy buttons can either be down or not down; and only one at a time, therefore because of this we can simply use an easy conditional statement to

see which is the time limit, after that you can input the ‘duration_for_session’ as the time limit and the game will stop when the time limit is reached.

```
                if self.mins == self.duration_for_session:
                    self.pause_switch.disabled = True
                    Clock.unschedule(self.update_display)
                    Clock.unschedule(self.update_display_clock)
                    self.status_label.text = 'Status: [Label:b][color=00ffff]Shift has ended[/color][/b]'
```

I also used one of the classes as a global variable so that I could access the variable that was chosen in the setup screen to use in my clock in the session screen.

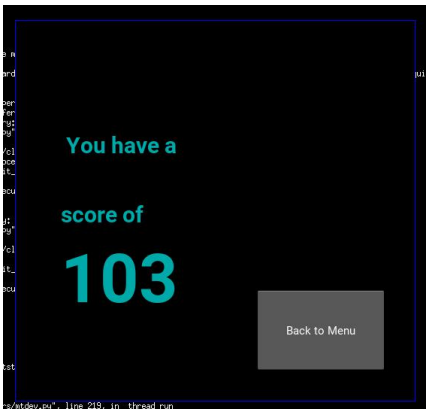
```
sm = ScreenManager()

session_setup = SessionSetup(name='session_setup')
session_screen = SessionScreen(session_setup, name='session_screen')

sm.add_widget(MainMenu(name='main_menu'))
sm.add_widget(session_screen)
sm.add_widget(TutorialScreenText(name='tutorial_screen_text'))
sm.add_widget(session_setup)
sm.add_widget(AboutScreen(name='about_screen'))


class PyATCApp(App):
    def build(self):
        return sm

if __name__ == '__main__':
    PyATCApp().run()
```

Here we can see how we declare each screen on the Kivy Screen manager and how furthermore for the Session Screen we also declare the Session Setup; which is the one that contains the user input when it refers to the time.

After work on the clock and its implementation was finished it was time to work on the final screen; the score screen. A similar look was given for this screen. Here is the Kivy code.

Here we can see how this layout follows the standard ‘left pane with button access’ and ‘right pane with content’, in this case the content is another paragraph which will have the final score after the session ends this score will be given even if the user crashes. The score is made of the planes he landed and based on the difficulty he had. This is a sample of it;

```
<ScoreScreen>
    GridLayout:
        rows: 1
        ScrollView:
            Label:
                size_hint_y: None
                height: self.texture_size[1]
                text_size: self.width, None
                id: paragraph_box
                markup: True
                padding: 60, 40

        BoxLayout:
            orientation: 'vertical'
            width: 250
            size_hint_x: None
            padding: 40, 40
            Label:
                text:
            Label:
                text:
            Label:
                text:
            Button:
                text: 'Back to Menu'
                on_release: root.manager.current = 'main_menu'
                on_release: root.manager.transition.direction = 'right'
```



After this was done it was time to work on the next version.

Version 0.8 - landing implementation and different airports

This version brought the implementation of landing into the game, to do this I used the same thinking and logic behind the plane mid air collision model. But however this time I used the runway as the ‘collision item’ instead of any other plane. This is how the code looked.

```
if (plane.x < 125 and plane.right > 130 and plane.y < 80 or plane.top > self.height - 90) and self.landing_request = 1:
        self.airplane_remove_widget(plane)
        self.overall_landed = self.overall_landed + 1
```

Here we can see how we are using the same proximity warning for an aircraft leaving the airspace as we use for the aircraft landing, after that we remove the aircraft from the airspace and add one to the landed aircraft counter, so that later on we can come up with a score.

This is how we call the landing command, we simply added to the commands list.

```
elif plane_instruction == 'CL':
    self.landing_request = 1
```

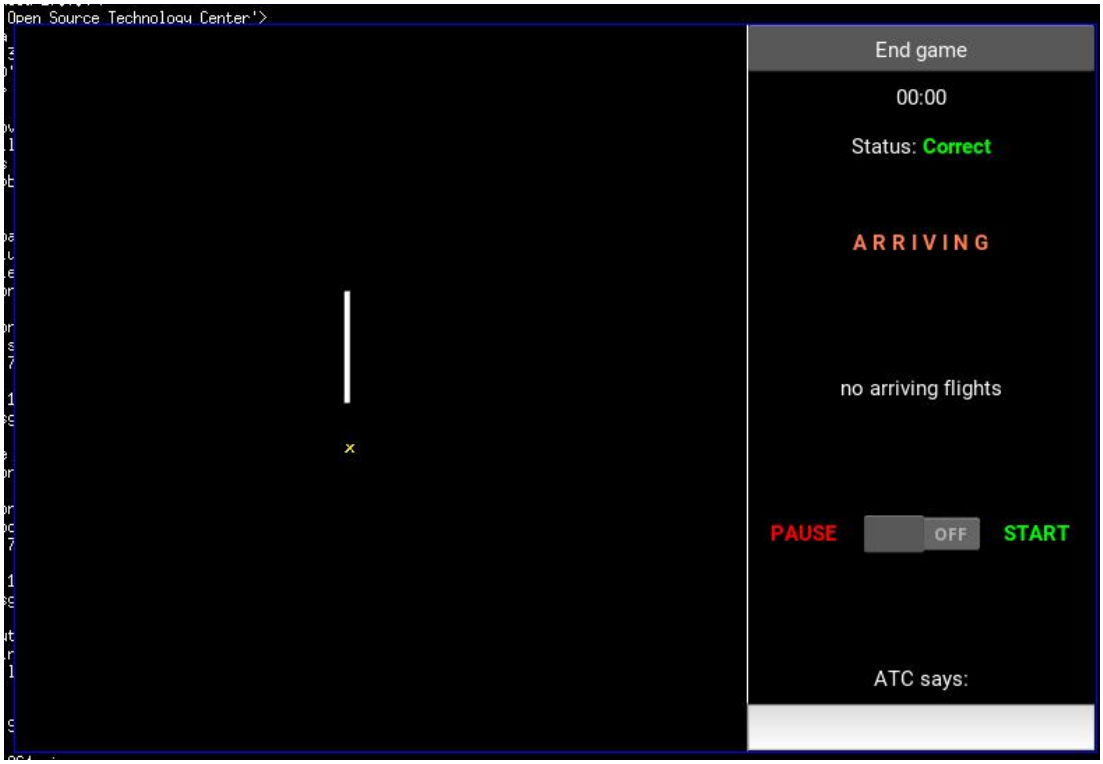
We don’t have to do anything else as the update part handles the removal of the aircraft from the screen and the list of planes in the left bar.

After that it was time to start work on the more airports; by default the airport I have is the Palma de Mallorca airport. Here are the three airports using the Kivy language as the layout creator; I won’t show the code as its the same that I use for the other airport.

Los Angeles Intl Airport (LAX)

Palma de Mallorca Intl Airport (PMI)

Norwich Intl Airport (NWI)



After that all I needed to do was do one airport for the tutorial mode; however for this airport I was going to use an overall simpler layout, one without timer and score screen. So i copied the code of a single runway airport and moved it into a new class (a new window in Kivy) called the TutorialScreen, which would be accessed from the ‘Go to practice airfield’ in the Tutorial screen.

This screen aims to help the user gain control and confidence by providing a more relaxed and user friendly atmosphere.

And with that screen finished PyATC was mostly finished.

Version 0.9 - final test and runs before delivery to stakeholders

I have tested the final game and I am very happy with the results, I think it covers everything that I wanted to do and I think it matches the project aims very well. Of course there are some minor bugs that I have fixed after extensive white box reviewing The first one was how sometimes the game created too many airplanes at once. Of course this happened because the planes are created randomly at a rate defined by the difficulty. And that meant that at any given point the number needed could be generated twice in a row.

To fix this I initialized a variable called since_last_plane and made it act as a count up method so that the wait would always have to be 6 updates (2 seconds) between one plane creation or another one. Here is the code;

```
    if self.session_setup.session_screen.difficulty_easy.state == 'down':
        difficulty_level = random.randrange(1, 150)
    elif self.session_setup.session_screen.difficulty_medium.state == 'down':
        difficulty_level = random.randrange(1, 100)
    elif self.session_setup.session_screen.difficulty_easy.state == 'down':
        difficulty_level = random.randrange(1, 75)

    if difficulty_level == 1 and self.since_last_plane >= 6:
        self.plane_creation()
        self.since_last_plane = 0
```

After that I also had some minor problems with the string handling and I realized that I had mistaken some of the logic; initially I had set that in order for a plane to turn heading the input had to be in a certain range;

```
if plane_instruction == 'FH':
    if isinstance(plane_instruction_value, int) or (plane_instruction_value in range(360)):
        self.status_label.text = "Status: [b][color=[/color][/b]"
        return
    angle = int(plane_instruction_value) * math.pi / 180
    plane.angle = angle
```

For some reason the circled part of the code didn't work, and after much testing I decided to simply implement another simple arithmetic operation. This was a quick fix, but after trying other methods which seemed better in the beginning none of them worked. So this was one of the best alternatives I could of used.

```
if plane_instruction == 'FH':
    if isinstance(plane_instruction_value, int) or (plane_instruction_value - 360 < 0):
        self.status_label.text = "Status: [b][color=[/color][/b]"
        return
    angle = int(plane_instruction_value) * math.pi / 180
    plane.angle = angle
```

Furthermore and after about 10 minutes of playing the major problem I found was how I wasn't completely comfortable with the way planes were shown in the screen. They were all white balls and I had to remember which plane was what, to fix this I added a text that would always follow the planes around; this text would have the plane's unique ID and therefore served great when playing with a large amount of planes.

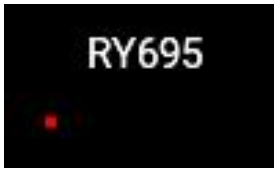
Here is the code and a screenshot of how it looks;

```
<Airplane>:
    size: 4, 4
    name: 'Default'
    canvas:
        Color:
            rgba: 1, 0, 0, 1
        Rectangle:
            pos: self.pos
            size: self.size
    Label:
        text: root.name
        pos: root.pos
```

As we can see the only thing we have done is added a label to each plane, but it immensely improves the game play, and it was something that could only be noticed when one played the game for some time.

Before

After



This were the only problems I could find with the code after 8 versions of RAD development whereby I also fixed bugs throughout the development.

The fact that I couldn't find any more problems meant that it was time to give release PyATC and to have the stakeholders perform Black Box testing.

PyATC 1.0 Evaluation

Does PyATC meet the criteria chosen whilst planning?

Let's review the criteria and then see if PyATC does meet it.

From a design point of view

The design will be modular, each screen will serve a purpose

> *OK > Classes used as different screens, each screen is different*

There will be a title page whereby all features can be accessed

> *OK > There is the main menu screen which shows all the places the user can go*

The screen will have an easy to read feel to it, with black background and contrasting items such as the runway and planes

> *OK > The background is black without any background images and with whites and strong colors to aid contrast and readability*

There will be a variety of airports that the user can choose from

> *OK > There are three airports (LAX, PMI and NWI) and the practice airport*

The planes in the screen will randomly appear and the user must guide them

> *OK > Depending on the difficulty planes appear at a rate; the user commands them to the landing strip*

The users will count with a real time view of what is going on, however all the information will be reachable through the side bar which will also have the remaining time and a pause button

> *OK > There is a real time version which shows the motion taking place, however all the important information is given by the left panel.*

Although not graphically extraordinary, the game should still represent the modern looks the Kivy UI library

> *OK > It scales and supports touchscreen interfaces.*

Although not very complex graphically, the game should still be intuitive, with special emphasis on navigating the menus and engaging with the simulation

> *OK > Each screen has the same layout with easy to read text and available buttons to navigate.*

From a portability and usability point of view

The game should be able to run on Windows, macOS and Linux

> *OK > The game is written in Python using the Kivy library, both supporting multiple operating systems*

The game will be usable for a variety of computer types (workstations, touchscreen laptops...) as it will detect and interact with touch

> *OK >The fact that is written in Python and Kivy (touch support) means that it can be run in any computer that has the ability to run Python*

The game will be playable from different sizes; as it will dynamically adjust to different screen sizes

> *OK >Kivy takes care of dynamically adjusting to different sizes, furthermore the way the screens are written ensures that upon resizing the important part of the screen always remains visible.*

The game will be able to run from low-powered hardware (i.e. Raspberry Pi)

> *OK > Because the game is written in Python it will be possible to run it from*

From a process point of view

The final score will be composed of user changeable values

> *OK >It is composed of the time and difficulty variables*

The timing and location of arriving planes will be randomized within a framework based upon the selected difficulty

> *OK > The timing and location have custom scripts which make them random*

The player will only control planes that are incoming

> *OK > The player only deals with incoming planes, not with outgoing or even landing the planes*

The player will choose from a variety of difficulties, that will change how often planes are generated

> *OK > There is a custom script which makes plane only appear once a random number is matched*

As we can see the final solution, PyATC 1.0, does meet all the criteria set by the stakeholders and I initially.

Limitation of PyATC 1.0

There are of course certain limitations that exist from the solution, obviously the game cannot share user data and therefore comparing scores between users can be difficult if its not done by the users telling each other what they score they got.

After that, the other limitation is the fact that although it allows touch displays, it only allows keyboard inputs; this can be fixed in two ways;

> - The short-term answer if you only have a touchscreen could be to use a built-in touch keyboard from the screen. Windows has this by default for example
>
> - The long-term answer could be to implement a voice recognition library into the game and use a microphone or a keyboard as the input devices.

Furthermore, there is no multi-player support, this means that if you want to play the game with a friend then you have to use two computers; one for each. However if I had the time and I had done the speech recognition project I would implement a multi-player mode; whereby one user can control incoming planes and another user can control departing planes.

Another limitation of the game, more in terms of design, is how this game cannot be re-sized as it is being played; this is because the speed and angle variables use the distance over the screen; and if the screen is re-sized this can lead to certain problems. The user can however pause the game, re-size and then carry on playing. I had an idea on how to fix this; the theory was that Kivy would detect a moving screen and would send a stop

signal to the update functions (motion and timer). However Kivy does not have the ability to detect this yet and therefore I wasn't able to complete it.

So, as a recap - the limitations we have are...

Problem	Fixable?	Difficulty
Network sharing support	Implement online game database	High
Only semi-functional touch support	a) Use touch-screen keyboard	Simple
	b) Speech recognition	Medium
Multi-player support	Dependent upon speech recog.	Medium
Re-sizing whilst playing game	Once Kivy releases movement sensor	Medium

Proving that the UI and main features are effective

The UI was mostly done in version 0.1, then as development went on things where added to it; however even at its most initial state (version 0,1) the UI had a clean look to it which made it easy to navigate. Furthermore the whole PyATC ecosystem follows a certain design guidelines;

- The UI has to be split into two main parts

- The right main part has to have be ability to edit the status of the current screen (close, go back to menu, go to score screen, etc..etc..) and therefore has to always be the same size upon re-sizing (typically this is circa 250 pixels

- The left hand side has to have the content of the current screen, with any main text in contrasting white to the black background (such as the tutorial screen), and any title or header of a soft color which also contrasts (blue and orange where used mostly).

- Each screen has to have the ability to take you somewhere directly.

- The motion of movements of screen has to follow the ideology that PyATC reads like a book, so that then when wanting to go from the main menu to anywhere else it goes from left to right; however when returning to the main menu the screen has to be animated the other way; from right to left.

After following this main guidelines to make sure that every screen had a similar feel to it I can happily say that the UI is effective and that based on what the stakeholders said; it was easy to navigate in it.

Furthermore and now talking about the features I can also happily say that the features that PyATC has are effective. This was proven when every item on the Success Criteria was marked as done. The only flaws of main features that PyATC could have are discussed in 'Limitations of PyATC' in page 53.

Of course there were features which ticked the box in the Success Criteria, but that could of course be improved. For example, "The final score will be composed of user changeable values", this is true as it is composed of a combination of the planes landed and the difficulty in the form of 'per every extra difficulty level add twice to the score' however it could also be improved by having the score composed of more things, here is an idea;

*score = landed * difficulty level * (inverse) incidents which were not crashes * time bonus*

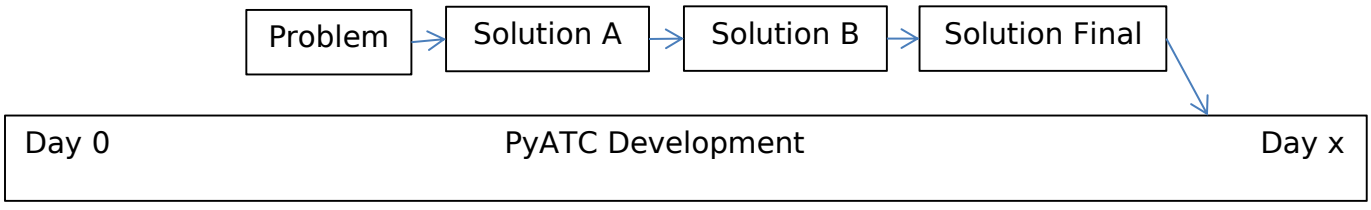
This score accounts for any incidents (such as having two planes too close, or asking planes to go faster than they can), it also has a time bonus so that a user is rewarded more points for playing a 20 min session rather than a 5 min session due to the concentration needed for the 20 min session.

Furthermore, another example is "There will be a variety of airports that the user can choose from", this, although true, doesn't have an exceptional number of airports, perhaps this could be improved by either adding more airports, or making a script that puts runways in a random order everytime (to avoid users getting used to one airport) or even to have the ability to make your own airports.

Was RAD development the correct development to use?

In my opinion RAD was without a doubt the best method I could of used. I found myself comfortable in an environment where I could prototype something and then try to fix it with a couple of tries to later on added to the actual build of the PyATC program.

The way I did this was...



Here we can see how what I did was prototype and then do improvements until the problem was solved to then added to the overall PyATC project thus removing the abstract layer I used during the development of the module.

Of course sometimes applying abstraction to the modules wasn't so beneficial; for example on the on_collision module I spent more time trying to remove the abstract layers rather than actually coding it.

But overall I am extremely happy with the result of using RAD. I enjoyed being able to improve by failing as it taught me about what is considered good practice and what is not; there was a point for example where I tried to use a large amount of global variables in my code and after doing some more versions of the fix I decided to use local variables.

Initially I was going to use waterfall but I ended up deciding on RAD as I was very exited about the large amount of prototyping that could be done; I find myself more comfortable with prototyping and planning than only planning.

PyATC maintenance

There are a wide variety of things that I need to look out for when trying to maintain PyATC and by using certain methods I tried to avoid making it very uncomfortable for anyone to try to maintain the code;

For example, apart from Kivy and maths I didn't use any other libraries; this is a good idea because if Python or Kivy gets updated by having less libraries you significantly lower your chances of having problems with the newer versions. An example of this is the clock; I could of used python.time or any other counting up library but instead I decided to make my own script using Kivy's in-built clock.

Furthermore I tried to keep my variables as self explanatory as possible, often avoiding naming things numerically but rather with names that I would remember, for example, 'difficulty_hard' instead of 'difficulty_3'.

By using this technique reading this piece of code becomes a much more comfortable task

```
def airspace_abandon(self):
    self.pause_switch.disabled = True
    Clock.unschedule(self.update_display)
    Clock.unschedule(self.update_display_clock)
    self.status_label.text = 'Status: [b][color=ff0000]Aircraft left radar[/color][/b]'
```

We can see how the variables make sense in relation to what is going on in the game.

Lastly, I used annotations on specially hard parts of the code, or parts with similar logic that one could get mixed in, an example of this is in the clock, where I used annotation to describe which if statement moved which part of the clock

```
def update_display_clock(self, dt):
    # clock
            if self.pause_switch.active == True: # If the switch is off
                if self.zero < 60 and len(str(self.zero)) == 1: ## '00:0'
                    self.countup_label.text = '0' + str(self.mins) + ':0' + str(self.zero)
                    self.zero += 1

                elif self.zero < 60 and len(str(self.zero)) == 2: ## '00:'
                    self.countup_label.text = '0' + str(self.mins) + ':' + str(self.zero)
                    self.zero += 1

                elif self.zero == 60:
                    self.mins += 1 # Add a minute
                    self.countup_label.text = '0' + str(self.mins) + ':00' # EG '01:00'
                    self.zero = 1 # Reset zero to start again for a new minute
```

By having those commands I can see which conditional statement does what in case I ever needed to change a single ‘hand’ of the clock.

On the other hand, and regarding code sustainability we could talk about how the game will be maintained for the users.

According to research I’ve done, air traffic control simulators; although not very popular, are niche and have had a steady customer line for years, furthermore the air traffic control simulator that are available online have been there for years and still have the same customer base. This means that PyATC will not be a *vuvuzela-piece-of-software*, this relating to the short period of time in 2012 where, due to the South African Football World Cups, vuvuzela sales raised by the millions for a short period of time.

Furthermore, and thanks to the fact that PyATC only uses Python and Kivy (both being continuously supported and update software) it is very probable that PyATC would still be able to run on computers for years to come.

Stakeholders final opinion of PyATC 1.0

After showing the game to all of my stakeholders they were all extremely happy with the result. This document therefore certifies that the software solution, PyATC, has met all of the success criteria.

Signed,

PyATC Stakeholders

Alan Harvey	Matthew Beimborn	Harry Dalton
Elliott Attew	Marlowe Buchannan	