**Final Project: Plane Tracking System and Tag Game**

Kailyn Isaac

John Brown University

December 15th, 2024

CS-3743-01

1. **INTRODUCTION**

Air tracking systems have been a huge development over the years, necessary for safety and traffic control. Predicting the paths of planes considering weather, wind flow, other planes and specific engine requirements or gas levels are some of many components to take into consideration.

As an avid flyer from the Tulsa airport, this project’s goal was to simplify this system but with that same security in mind: a GUI displaying planes within the specified location range tracked from an API given from a website [1]. This GUI would have multiple warning levels and a predicted path display for the planes. Creating multiple plane types, including timed updates while processing real-time user input for the warning system are a few of the aspects of programming that this project will be demonstrating.

The different levels of this project started with research into API, followed by the creation of data storage classes to be used and compiled into the GUI, notably the PlaneLocation class. When the API process revealed to be a longer process than feasible, the plan changed: to keep this warning and location tracking in mind, two planes were made to respond to user input and have visible path prediction. Curving was not an option anymore because of keyboard restrictions, but both predicted paths and warnings were introduced into a game-like version of (explosive) airplane tag. This introduced a small portion of threading for interrupts and game ends.

1. **METHODS**

The first part of this project, the original version, was working on API and lists of plane objects. The code for API processing was mostly borrowed from the internet [2], set in the LocationAPI class. The PlaneLocation class was added to be filled up with the given variables the website provided, which specifically needed to be named the same or be put through a specific JSON formatting change in the code. The UML class diagram for this part can be seen in Fig.1.

A screenshot of a computer screen

Description automatically generated

Fig. 1: Class Diagram of Version 1

The second part of this project came after multiple tests with the API led to errors that couldn’t be resolved after a couple of hours and questions for the instructor. Multiple alternative options came to mind: downloading multiple test locations to calculate the predicted path option (which was not a possibility after over an hour of research so the idea was dropped), or making another predicted path option that involved user input for more randomness and therefore testing accuracy of that predicted path. This second option was chosen and transformed into a game of sorts.

Here, the major parts of the paint events and basic movement were inspired by a video tutorial[3], making the Form1View class. To adapt it for two players, a Plane class was added to get and save all the input the users would need. This could hold more players than this decently easily, but for the sake of time only two were added. The predictive path option was added into that class once the appearance and movement of the planes were confirmed. Most of this was done through image uploads (some from the internet, some the result of my personal artistic skills), so different arrows and variations were added to make it more dynamic (GoingRight() function was mostly used), even if the combination of realistic plane and 2D drawings made an interesting combination. The timer that kept track of the movements also dictated how often the predictive path changed, which was made through basic coordinate subtraction and an array of x and y “next” coordinates within three steps. The tricky part was adding options for speed and levels of warnings that wouldn’t be affected by the arrow keys. To go around this issue, the two planes were controlled with the letter keys WASD and IJKL instead. The UML class diagram for this second version can be seen in Fig. 2.

A black screen with white text

Description automatically generated

Fig.2: Class Diagram of Version 2

The dynamic system for the images can be seen in Fig. 3.

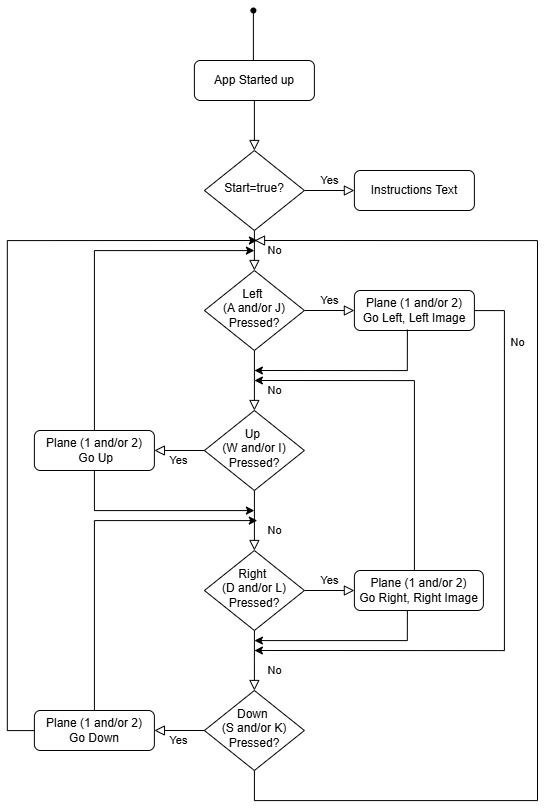


Fig. 3: Plane Movement State Diagram

The state diagram for the more specific event images can be seen in the following Fig. 4.

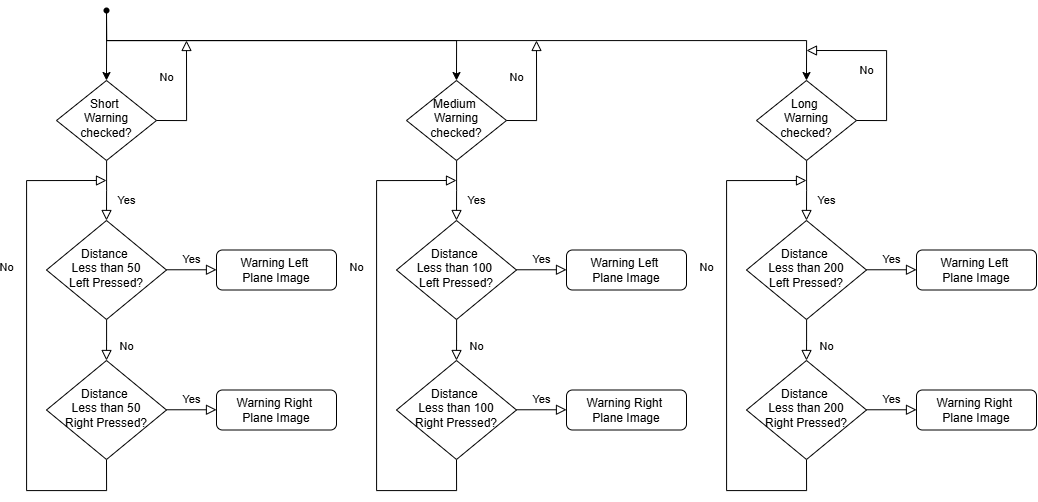


Fig. 4: Plane Prediction Event State Diagram

Speeds were also added in this game through a button. The following state diagram shows its events.

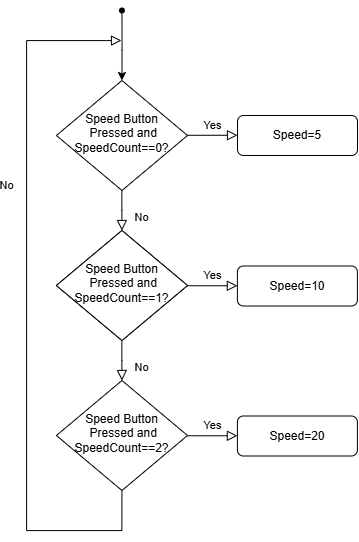


Fig. 5: Speed Button State Diagram

Throughout all the processes there is a check for whether the planes are within 10 digits of each other leading to a crash and end panel for WASD winner. The second timer is also its own event that finishes the whole game with an end panel of IJKL winner.

1. **RESULTS**

Taking creative liberty, this was made into a tag game between player WASD and player IJKL, where another timer was added in the background to challenge that first player to catch the second within the time frame (resulting in an explosive crash). The game starts out with rough instructions and would need more visual polishing, but overall is fully functional with slight catches (such as diagonal traversing is a lot faster than straight and down) and rough memory use. The whole game can be found and downloaded from GitHub at the link provided [4]. While testing was done intermittently through playing the game, two testing details were provided for extra reinforcement, testing the predictX() (it is exactly the same for predictY() and was not deemed necessary for the second function) and GoingRight() functions. These can be seen in the appendix.

What would have been done with more time and money would be the first part of the original project: buying an API pass for more than 100 tries. The location tracker once coordinates are set would have been doable using the timer, setting off an API and checking every once in a while for a new plane emerging in the area. The output value for each plane would have been a simple conversion from lateral and longitudinal coordinates to the integers within the given area of the GUI to appear there as close as possible to reality. The tracker, for the sake of the warnings, would not take altitude into consideration but could through adding a third coordinate to each plane. Since these would be actual planes with a possibility of curvature in their paths, a simple three point system to compare those locations to a possible curving radius would have been put in place, the distance between each point determining speed as time between points is constant, and through those values, a predicted curved path can be added, three, four or more points depending on the wanted value from the user. Instead, I made the small game, but I will try to work on the API in my free time as it is going to be very useful in my career this summer.

1. **CONCLUSION**

This project was very polarized. On one hand, I have a product that works and that does do predictive paths as well as crash warnings and an actual crash. On the other hand, making the API code work would have been a big deal and is a loss to the project that could have been. I learned a lot in the process, notably that API is a process that holds a lot of intricacies, and that coding is a lot of fun when gaming starts getting involved. The use of threads could have been exploited more as well as imaging classes for the paint event, but overall, the game worked. The goals were still obtained, while noticeably less hard to achieve in the new context. After spending over 20 hours on this project, this was time well spent that I will be taking into my future career as both a troubleshooter and problem-solver that thinks outside the box.

1. **REFERENCES**
2. <https://aviationstack.com/documentation>
3. <https://stackoverflow.com/questions/9620278/how-do-i-make-calls-to-a-rest-api-using-c>
4. [Move an image using the arrow keys in C# and windows form using paint event.](https://www.youtube.com/watch?v=5aBN3fp4uk0)
5. [kailyn-calypso/PlaneTracker: Plane tracker GUI with 1, 5, and 10 minute collision warning.](https://github.com/kailyn-calypso/PlaneTracker/tree/main)
6. **APPENDIX : UnitTest1.cs**

using Microsoft.VisualStudio.TestTools.UnitTesting;

using System.Numerics;

using PLaneTrackerGUI;

namespace PLaneTrackerGUI

{

[TestClass]

public class UnitTest1

{

[TestMethod]

public void TestLocationPredictX()

{

// Arrange

Plane plane = new PLaneTrackerGUI.Plane(10, 100, 200, 50, 50);

plane.AverageX = new int[] { 100, 110, 120, 130 };

// Simulate movement

int currentX = plane.PositionX;

// Act

int[] predictedX = plane.LocationPredictX(plane.AverageX,

currentX);

// Assert

int expectedStep = 10;

// Difference between consecutive AverageX entries

Assert.AreEqual(currentX + expectedStep, predictedX[0],

"Prediction step 1 is incorrect.");

Assert.AreEqual(currentX + 2 \* expectedStep, predictedX[1],

"Prediction step 2 is incorrect.");

Assert.AreEqual(currentX + 3 \* expectedStep, predictedX[2],

"Prediction step 3 is incorrect.");

}

[TestMethod]

public void TestGoingRight\_DetectsCorrectDirection()

{

// Arrange

Plane plane = new Plane(10, 100, 200, 50, 50);

int[] increasingX = new int[] { 100, 110, 120, 130 };

// Simulate moving to the right

int[] decreasingX = new int[] { 130, 120, 110, 100 };

// Simulate moving to the left

// Act

bool isGoingRight1 = plane.GoingRight(increasingX);

bool isGoingRight2 = plane.GoingRight(decreasingX);

// Assert

Assert.IsTrue(isGoingRight1,

"Plane should be detected as going right.");

Assert.IsFalse(isGoingRight2,

"Plane should be detected as going left.");

}

}

}