TWO PUBLISHED FLIGHT DYNAMICS MODELS REWRITTEN IN RUST AND STRUCTURED AS AN ECS

THESIS

Chad A. Willis, 2d Lt, USAF

AFIT-ENG-MS-21-M-095

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-21-M-095

TWO PUBLISHED FLIGHT DYNAMICS MODELS REWRITTEN IN RUST AND STRUCTURED AS AN ECS

THESIS

Presented to the Faculty

Department of Aeronautics and Astronautics

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Science

Chad A. Willis, BS

2d Lt, USAF

March 2021

AFIT-ENG-MS-21-M-095

TWO PUBLISHED FLIGHT DYNAMICS MODELS REWRITEN IN RUST AND STRUCTURED AS AN ECS

THESIS

Chad A. Willis, BS

2d Lt, USAF

Committee Membership:

Dr. Douglas D. Hodson, PhD
Chair

Maj. Richard A. Dill, PhD
Member

Dr. Scott L. Nykl, PhD
Member

AFIT-ENG-MS-21-M-095

Abstract

This thesis explores using the Entity-Component System (ECS) architecture to implement a Flight Dynamics Model (FDM) by re-implementing two published versions in the Rust programming language using the Specs Parallel ECS (SPECS) [1] for military simulation advancement. One FDM is based on Grant Palmer's published textbook titled *Physics for Game Programmers* [2], and another is based on David Bourg's textbook titled *Physics for Game Developers* [3]. Furthermore, this thesis uses these models within an interactive flight simulator.

The ECS architecture is based on the Data-Oriented Design (DOD) paradigm, where Components contain the data and the Systems implement the behavior which transforms data. The specific ECS selected for this research is the Rust-based SPECS framework. One goal of this research is to define the Components and the Systems of an ECS for the two FDMs and use them to build a flight simulator by creating additional Systems and Components to read keyboard inputs and send User Datagram Protocol (UDP) packets to the FlightGear application for visualization [4].

The ECS Systems used to calculate the equations of motion both for the Palmer-based and Bourg-based FDM's functionality were tested against the original code and were found to work as expected within a floating-point tolerance. A benchmark comparison between both FDMs determined that the Palmer model is faster on average than the

Bourg model. Moreover, the re-implemented FDMs were successfully used to create interactive flight simulators using FlightGear for visualization.

Acknowledgments

I would like to thank everyone who helped me complete this research. I want to thank my research advisor, Dr. Douglas Hodson, for his support and guidance in all aspects of this thesis. I want to thank my committee members, Maj Richard Dill and Dr. Scott Nykl, for their time reviewing this research along with their valuable feedback. I want to give a special thanks to a prior advisee of Dr. Hodson for his excellent feedback and time. I also want to thank my friend and recent AFIT graduate for helping me through some new material. Lastly, I want to thank all my friends and family for their support and motivation. I appreciate everyone dearly who had any inspiration or influence on this research.

Table of Contents

List of Figures

List of Tables

TWO PUBLISHED FLIGHT DYNAMICS MODELS REWRITTEN IN RUST AND STRUCTURED AS AN ECS

I. Introduction

Real-time computer simulations are important tools for the military [5]. The reasons to use a virtual computer simulation over real-life simulation are because it is less expensive, more accessible, more versatile, and completely safe. Virtual simulations are cost-effective as they do not require real-life vehicles and equipment, which require fuel, maintenance, and human resources to operate. They are more accessible since all that is needed is a computer to use them. They are more versatile because one can simulate scenarios that might not be possible to put together in real-life. And finally, they are entirely safe since users cannot hurt themselves. Today, military computer simulations exist to simulate everything that could occur at war, such as battle planning or individual pilot training [5].

Simulation software used is generally written using the Object-Oriented Programming (OOP) paradigm, which revolves around designing classes of data and its associated behavior. However, there is an alternative, more modern paradigm, called Data-Oriented Design (DOD), which focuses on the data itself and considers how it is laid out in memory. The DOD paradigm provides some performance advantages over OOP that is explored in this thesis as it relates to implementing two Flight Dynamics Models (FDM) and a flight simulator. One application of DOD is specifically discussed, called the Entity-Component-System (ECS) architecture. ECS is an architecture that is used in the video game industry within game engines. However, video games are similar to simulations, making them candidates to use the ECS architecture [6]. This thesis investigates the use of ECS for simulations.

In this thesis, the ECS design in a military-oriented simulation is of particular interest because it is a novel area of research. The ECS architecture is a way of organizing data, alternatively to using OOP to organize data. Where OOP relies on data hierarchies, ECS relies on the compartmentalization of data in which data content and its functionality on the data are decoupled [6]. Notably, Unity, a game engine used to create real-time 3-Dimensional (3D) games, uses an ECS architecture in its game engine [7]. The popular battle royale game *Fall Guys: Ultimate Knockout*, for example, was created with Unity, which relies on the ECS design [8]. The ECS architecture, which is discussed in more detail in Chapter II, leads to code that is more reusable, readable, and performant [9]. Although possible pitfalls in software created using an ECS are also presented.

In this thesis, the focus is on the research and development of FDMs. This important piece of flight simulation software models realistic flight behavior [10]. In a FDM, the physics calculations that mimic real-life flight are referred to as the equations of motion. The numerical integration of equations of motion determines how an airplane is translated and rotated, taking into account external forces, such as gravity and thrust, as well as the physical properties of the airplane. The Degrees of Freedom (DoF) classification of an FDM hints at how these translations and rotations are computed; three degrees of freedom are associated with translation (e.g., x, y, z), and three are associated with rotation (e.g., pitch, roll, yaw) for a total of six possible degrees that could be calculated [11]. This thesis describes translating two FDMs that are built using the DOD-based pattern, ECS. Subsequently, both FDMs are interfaced to a keyboard-driven input flight control system and output network packets to a graphics visualization system.

The Rust programming language was chosen to implement the ECS architecture in a FDM. Rust, created in 2015, introduces modern features not available in other languages. These features stem from two concepts that Rust focuses on memory safety and performance. Rust will always guarantee memory safety, avoiding issues in software such as data races, dangling references, and buffer overflows [12]. It accomplishes this safety by checking for these errors at compile time. Other languages, such as C++, rely on experienced programmers to be sure the software is memory-safe. Although some languages are also memory-safe like Rust, they sacrifice performance by using a garbage collector that manages memory at runtime [12]. Rust's performance is achieved because it does not need a garbage collector and checks for memory issues at compile time. However, Rust is not without drawbacks compared to other languages, which is discussed in Chapter II. The Specs Parallel ECS (SPECS) [1] is a Rust-based package that defines an ECS architecture.

1.1 Problem Statement

In real-time military flight simulations, it is common to see the software using an OOP design approach and be written in a systems programming language like C++. Typical scenarios might employ hundreds or even thousands of moving entities, such as airplanes or soldiers. These entities have many different functions acting on potentially large sets of data associated with them. Unfortunately, with OOP, the entanglement of functionality and data memory impedes potential performance and causes code dependencies – which complicates the addition or modification of functionality or data entities [9]. Also, memory-management in OOP software is not ideal for accessing data as efficiently as possible in cache memory [9]. Furthermore, when programming with a systems language, it is common for beginner and intermediate-level software developers to introduce memory vulnerabilities in code [9].

1.2 Research Objective

Typical simulation software could benefit from the compartmentalization that the ECS approach employs in game engines. With functionality and data decoupled, the software can be as performant as possible in terms of memory-management and parallelization [9]. Also, the decoupling nature of ECS improves ease of code development because pieces of code are not dependent on each other to work [9]. Furthermore, Rust solves the memory safety burden that plagues systems-level languages by enforcing strict rules at compile time. The goal of this research is to build easily extensible FDMs implemented as Rust-based, ECS-based architectures and use them in real-time interactive flight simulators visualized within FlightGear [4]. The first FDM is based on Grant Palmer's works in [2], while the second FDM is based on the works by David Bourg in [3].

1.3 Hypothesis

The thesis hypothesizes that a FDM implemented as a Rust-based, ECS design used in simulation improves upon typical OOP simulation design by introducing some powerful benefits that other FDMs do not possess. By using the ECS architecture, functionality is decoupled from the actual data in memory. This architecture is a powerful and popular strategy in designing complex video game engines, which need to process updates to large amounts of game entities dealing with large amounts of data [9]. This decoupling allows for the methods that operate on game entities to be easily parallelized when possible and use cache memory more efficiently. In addition to the performance benefits from decoupling, ECS eases code development. Due to decoupling, the ability to modify existing functionality or entities, or add new functionality or entities is a much smoother process than OOP. So, the same reasons game designers choose to use the ECS design in a video game translates over to FDMs for use in simulation.

1.4 Approach

The methodology of this research is explained in full detail in Chapter III, but an overview of the approach is given here.

This research focuses on building two Rust-based, ECS-based FDMs: the first is a published FDM based on the work by Grant Palmer in [2], the second is a published FDM based on the work by David Bourg in [3].

In terms of ECS compatible aspects, the Palmer-based FDM comprises one Component and one System, which play a part in the airplane Entity’s data storage and functionality, respectively. The System and Component serve to calculate the equations of motion of the airplane. The Component, which contains the airplane data used by the System is DataFDM. The System, which adds functionality to the Component data of an airplane Entity, is EquationsOfMotion. Once this FDM was built, to ensure its accuracy in modeling flight, it was tested if it replicates the same realistic flight data within a floating-point tolerance of the data generated by the original FDM written in the C programming language by Grant Palmer.

Similar to the Palmer-based FDM, the Bourg-based FDM is also composed of one Component and one System that serves to calculate the airplane's equations of motion. The Component is also called DataFDM. The System is also called EquationsOfMotion. Once again, to ensure its accuracy of modeling flight, it was tested if the realistic flight data is replicated within a small tolerance of the data generated by the original C++ FDM as described in Bourg's textbook.

With the two FDMs built, this thesis defines three additional Systems for each FDM to create two interactive flight simulators. The three Systems support flight control via keyboard input and networking to send User Datagram Protocol (UDP) packets to provide graphics within the FlightGear visualization system. The three Systems are FlightControl, MakePacket, and SendPacket. The Systems operate on the Components: KeyboardState, DataFDM, and FGNetFDM. When these Systems and Components that support keyboard input and graphics stimulation are added to the extensible ECS design in conjunction with the EquationsOfMotion System and the DataFDM Component for each FDM – flight simulators are created.

The FDMs were put through Rust integration tests to verify the FDMs work as expected compared to the original code. A series of flight scenarios were performed on both of the original FDMs for a set amount of time, and the flight data at the end of the simulation was recorded. That data was then given to an integration test that ran the same scenario but using the FDMs built in this thesis. At the end of the integration test, the results were compared to a floating-point tolerance. A successful test verifies that the

FDMs work as expected, and therefore the ECS-based FDMs created meet the goal of the hypothesis and work as expected.

1.5 Assumptions/Limitations

The following general assumptions and limitations are understood about the two re-implemented FDMs and their respective flight simulators:

- Both flight simulators are not standalone applications; a graphics generation program is needed to display the resulting simulation. The flight simulators are only configured to work with FlightGear as the visualization system. The simulators work with a specific packet structure required by FlightGear. However, the packet structure could be retrofit for different visualization systems.
- The flight simulators are only set up to fly interactively using a keyboard for input.
- It is assumed that the flight simulators are running on a modern computer. The example simulation loop is defaulted to run at a fixed frame rate of 30 Frames Per Second (FPS), and a slow computer may not be able to keep up with this FPS. However, the FPS could be reasonably lowered in code to work with a slower computer.
- Both flight simulators are set up to model the flight of a specific airplane configuration. The Palmer-based FDM is set up to model a Cessna 172P Skyhawk, whereas Bourg-based FDM is set up to model a fantasy airplane. The airplane properties in either FDM could be modified in the code, however.

- Both flight simulator examples are defaulted to fly at a predetermined location, specifically at Wright-Patterson Air Force Base. To fly in a new location, the user must update the origin position and general position variables within the `palmer.rs` and `bourg.rs` code files within the *examples* directory. The latitude, longitude, and altitude of the desired location must be plugged into both the origin position and general position variables.

1.6 Thesis Overview

This thesis is arranged into five chapters. Chapter II explains relevant background information used in this thesis. Chapter III provides the thesis methodology. Chapter IV is a results analysis of Chapter III. Finally, Chapter V summarizes the research and the impact of the research.

II. Background

This chapter provides an overview of the background information that has gone into building the FDMs and the flight simulators that use the FDMs. The information presented is important in understanding the inner workings of the research methodology in Chapter III. The first section of this chapter explains the basics of modeling flight physics in a Flight Dynamics Model. Next, the Entity-Component-System architecture, and an implementation of it, the Specs Parallel ECS (SPECS) [1], is described. Furthermore, the fundamentals of the Rust programming language are laid out. Next, the use of the FlightGear flight simulator [4] is explained. Lastly, networking and sending packets of data to FlightGear with Rust is described. All of this information is tied together to reach the result of a ECS/Rust-based interactive flight simulator using a published FDM that is displayed in FlightGear.

2.1 Physics of Flight Modeling

A Flight Dynamics Model (FDM) is defined as the set of math equations that compute all of the physical forces acting on a rigid-body airplane, like thrust and gravity [10]. A rigid-body can be defined as a body made up of particles that stay fixed and do not rotate or translate relative to the other particles. In this thesis, the physics of a rigid-body airplane is approximated through numerical integration of the equations of motion.

This thesis considers two different FDMs. The differences between each of the FDMs are pointed out because they have different considerations when determining the airplane's orientation. Although they are fundamentally similar in how they calculate the airplane’s position coordinates. Regardless, the final result is integrating the equations of motion, which describes the airplanes flight behavior with respect to time [3]. The equations of motion are summed up as determining the airplane's position, as well as the airplane’s velocity, which is the speed and direction of the airplane, over time [13]. However, before the equations of motion are solved, a basic understanding of Degrees of Freedom, forces, airplane components, rotations, and coordinate systems is needed.

2.1.1 Degrees of Freedom

Flight dynamics is the science of air vehicle orientation and control in three dimensions [14]. FMDs define equations of motions to account for the various forces (e.g., engine thrust, gravity) acting upon it to determine an airplane’s position and orientation (i.e., translation and rotation). Three Degrees of Freedom (DoF) are associated with translation (e.g., x, y, z), and three are associated with rotation (e.g., pitch, roll, yaw) for a total of six possible degrees that could be calculated starting with initial forces [11].

For example, the equations of motion for a particular FDM might only compute translational dynamics – which would classify it as a 3-DoF model. A complete model that computes all possible airplane translations and rotations associated with external forces would be classified as a 6-DoF model.

For this research, both an imperative programming-based FDM and an Object-Oriented Programming-based FDM are re-implemented using the Rust programming language and organized as Systems and Components in the ECS architecture. Both Palmer [2] and Bourg [3] define a flight model that can translate on all three axes and rotate on all three axes due to external forces. Therefore, our re-implemented versions are associated with what would be classified as 6-DoF.

2.1.2 Forces

Four major forces act on an airplane in flight: gravity, lift, thrust, and drag. These forces govern the motion of the airplane. Newton's Second Law of Motion states: the acceleration of a body is proportional to the resultant force acting on the body. The resulting equation is force equals mass times acceleration. Gravity pulls the airplane to the ground, and the airplane wings generate lift to counteract gravity to keep the airplane aloft. Thrust is generated by the airplane's jet engine or propeller to increase velocity and keep the airplane generating lift. Drag is the opposite of thrust; drag impedes an airplane's motion. Torque, also known as moment, is similar to force, but force causes linear acceleration, and torque causes rotational acceleration. An airplane must have a lift force acting upon it greater than or equal to gravity to fly [3]. Figure 1 is a visual of the four forces.

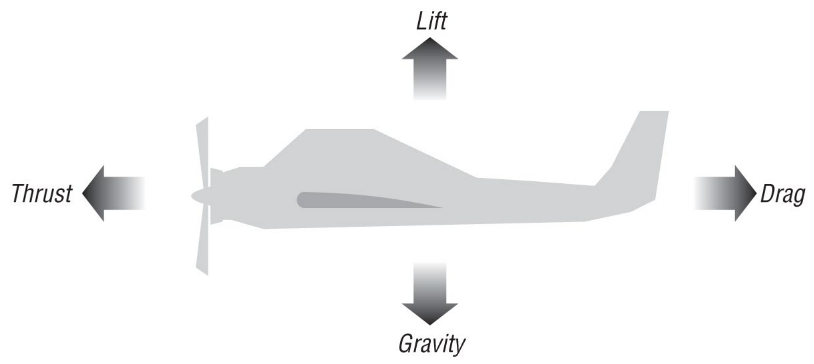

Figure 1: Forces Visualized [3]

2.1.3 Airplane Components

An airplane is composed of different components that affect the state of the airplane. The wings are the rectangular parts coming out of the main body. The span is the length of the wing, whereas the chord is the width of the wing. The ratio of span squared to wing area is called aspect ratio. The ailerons are on the outside end of the wings, and the flaps are on the inside of the wings. The elevators are on the tail and look like small flaps. The rudder is the vertical flap on the tail [3]. Figure 2 shows these airplane components.

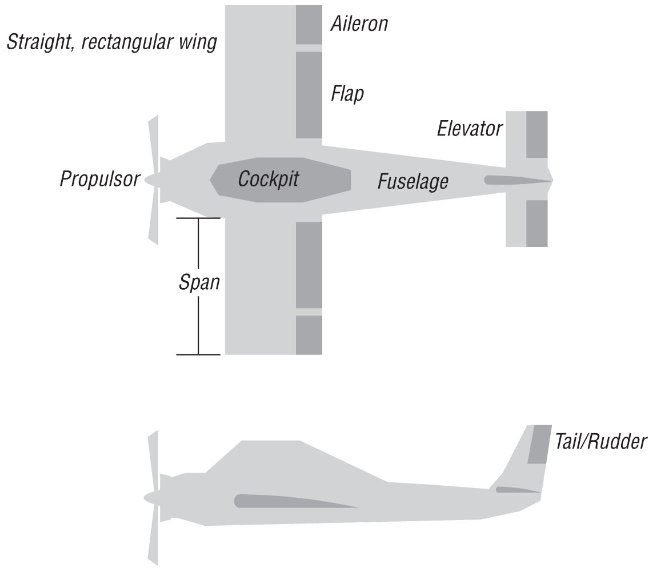

Figure 2: Airplane Components [3]

2.1.3.1 Airfoil

The cross-sectional structure of the wings component, called an airfoil, plays an important role in the performance data determining how the lift force acts on the airplane. Camber on the airfoil represents the asymmetry between the top surface and bottom surface of the airfoil, which creates a curvature. The angle of attack is the angle between the chord line and the direction of travel, or oncoming flow that the plane is moving through. The total lift force produced on the airfoil is caused by the camber and the angle of attack [3]. Figure 3 shows a labeled cross-section of an airfoil with an angle of attack.

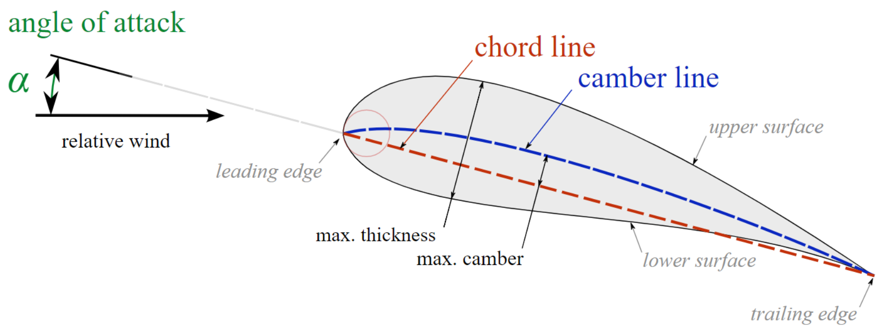

Figure 3: Airfoil Cross-section [15]

2.1.4 Rotations

The airplane's orientation when it is rotating can be described by three angles: pitch, roll, and yaw. These angles, measured in degrees, are called the Euler angles and represent the *xyz* axis on a coordinate plane, respectively [3]. The ailerons create the roll rotation. The elevators create the pitch rotation. The rudder creates the yaw rotation, and the ailerons create a secondary effect on yaw rotation [16]. The flaps do not rotate the plane, but they do affect lift by altering the chord and camber of the wing, which has to do with the shape of the cross-section of each wing [3]. Figure 4 shows the three types of airplane rotations.

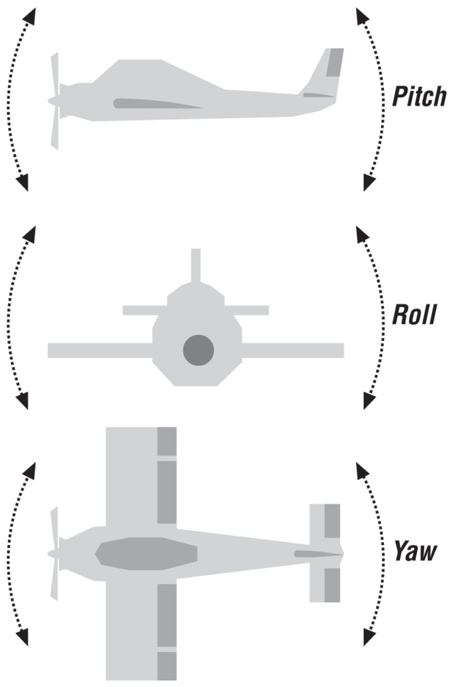

Figure 4: Airplane Rotations [3]

The distinction between the terms roll versus bank and pitch versus attack angle is worth noting. The motion about the axis controlling roll angle is called roll. Bank is another word for roll [14]. The pitch is the motion about the axis controlling pitch, whereas the angle of attack is the angle between the chord line and the direction of travel, or oncoming flow that the plane is moving through. The angle of attack, however, does affect the lift force on the airplane and thus affects the pitch motion [3].

2.1.5 Coordinate Systems

Coordinates are used to describe the location of an airplane and also the orientation of the airplane. A Cartesian coordinate system is used to describe an airplane's location on the Earth. Cartesian coordinates use the *xyz* axes to specify coordinate points on a plane

[2]. The Cartesian coordinates representing an airplane's location are sometimes referred to as the Earth, or world, space frame of reference [3]. Earth space coordinates are created when the Cartesian coordinates cross each other perpendicularly to create a position coordinate relative to an arbitrary fixed origin position [3]. The specific Cartesian coordinate system of interest is the East, North, Up (ENU) coordinate system. The ENU coordinate system is formed from a local tangent plane on the Earth's surface where the x-axis points North, the y-axis points East, and the z-axis points up [17].

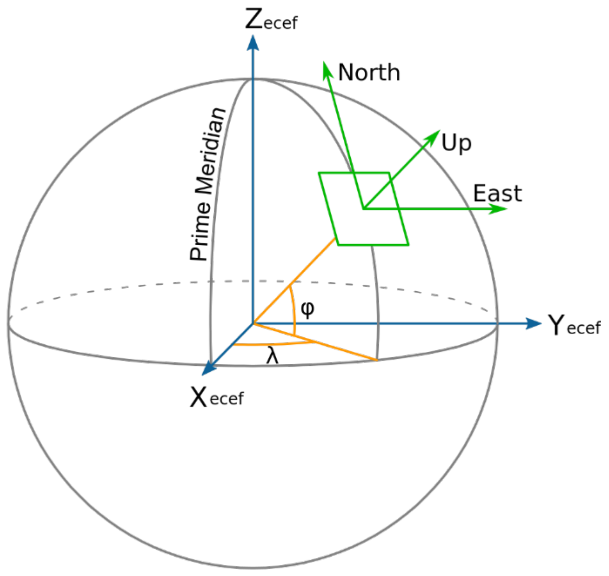

Figure 5: East, North, Up Coordinate System [17]

The Cartesian coordinate system is also used to describe an airplane's rotations about the *xyz* axes. The rotation is described by the three Euler angles (i.e., roll, pitch, and yaw) [2]. The rotational coordinate axes are fixed to the center of gravity of the rigid body airplane and moves with the airplane [2]. The rotational coordinate axes are referred to as the body, or model, frame of reference [3].

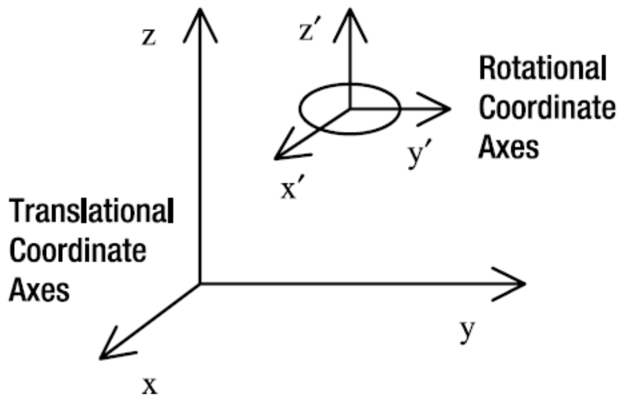

Figure 6: Rotational and Translation Coordinate Axes [2]

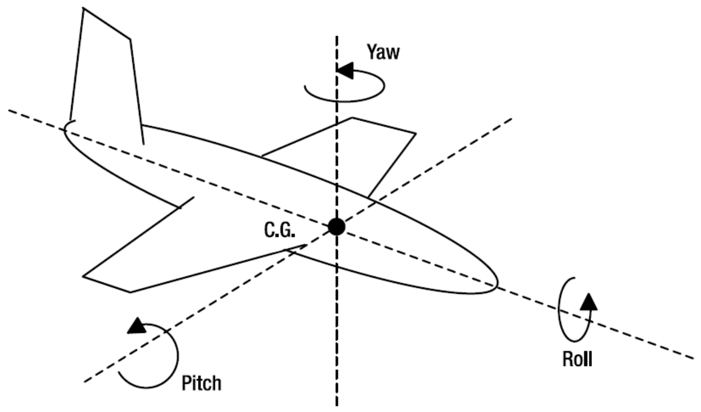

Figure 7: Rotational Coordinate Axes [2]

In the case of this thesis, ENU Cartesian coordinates describing the Earth frame of reference are converted to a geodetic coordinate representing latitude, longitude, and altitude. This conversion is performed because the FDMs use a Cartesian coordinate to represent position coordinates. However, FlightGear, which is being used as a visualization system, requires geodetic position coordinates to display the airplane on the Earth for simulation.

Xyz axes representing coordinate systems are sometimes oriented differently in different software. A Cartesian coordinate system can be defined as left-handed or right-handed. For example, the Cartesian coordinates used by Bourg's FDM in [3] uses a left-handed system with the x-axis pointing to the right, the y-axis pointing up, and the z-axis pointing into the screen. However, Palmer's FDM in [2] uses a right-handed system with the x-axis pointing into the screen, y-axis pointing to the right, and z-axis pointing up. Being aware that these differences exist is important because rotations on an airplane are expressed differently depending on the coordinate system. For example, an airplane that has a positive roll about the x-axis might need to be inversed to a different coordinate system representing a positive roll rotating in the opposite direction about the axis. So, for instance, when sending the Euler angles to FlightGear via UDP Packet to visualize the airplane on the screen, FlightGear may need the coordinates calculated expressed differently to properly depict the airplane flying how it is expected to be.

2.1.6 FDM Calculation Steps

With an understanding of the airplane components, airfoils, forces, rotations, and coordinate systems, the general steps that the FDMs take can be explained:

1. The mass properties of the airplane are calculated
2. The load, or forces and moments, on the airplane are calculated
3. The equations of motion are integrated

2.1.6.1 Mass Properties

The mass properties of an airplane's components are important because they affect how the airplane interacts with external forces. For example, the airfoil properties of a fighter jet have a different effect for creating lift than a propeller airplane. In Palmer's model, the Cessna 172P Skyhawk is being modeled. In Bourg's model, the airplane represented does not exist, but the model representation is shown in Figure 2 and Figure 4. With that, mass properties are chosen to represent that desired airplane. Mass properties include data about the airplane's mass, the airplane's surface areas of components that create lift, and the airplane's airfoil performance data [3].

Bourg's model is more complicated than Palmer's model because the mass properties are not simply chosen. In Bourg's model, the airfoil performance data, inertial moments of each component, and mass of each component are chosen, but that data is used to calculate the center of gravity, and the airplane's moment of inertia tensor. The center of gravity, or center of mass, is where the forces are applied on the airplane [3]. The moment of inertia is essentially the airplane's resistance to rotation. It can also be described as a distribution of mass. The amount of torque needed to make the airplane rotate about an axis relative to another point (i.e., the center of mass) [3]. The measure of how fast the airplane rotates is called angular velocity [18]. The moment of inertia tensor is the mathematical expression, represented as a 3x3 matrix, that has magnitude and direction and summarizes the moments of inertia values [3], [18].

2.1.6.2 Forces

With mass properties covered, the forces and moments acting on the airplane at all times are calculated. Lift and drag are not as simple to calculate compared to gravity and thrust, but the general steps will be explained. The lift and drag forces are calculated by first determining the direction in which they will act. Next, based on airspeed, angle of attack, and the airfoil performance data, the airplane's total lift force coefficient is computed. Although there are many components on the airplane that affect lift and drag, the total lift force on an airfoil is mainly composed of two components: the lift due to camber and the lift due to attack angle. The airplane’s flaps are used to the alter camber to create lift. Usually, the flaps are used at low speeds to increase lift. Drag increases with an increased angle of attack. If an airplane is flying at an angle of attack high enough, it will not be able to create any lift and will stall. Figure 8 is a visual that shows lift coefficient versus attack angle and that when attack angle is increased, lift increases. However, at some point, a stall occurs with lift dropping off rapidly [3]. Also, shown in Figure 8 is that with flaps lowered, the lift is increased.

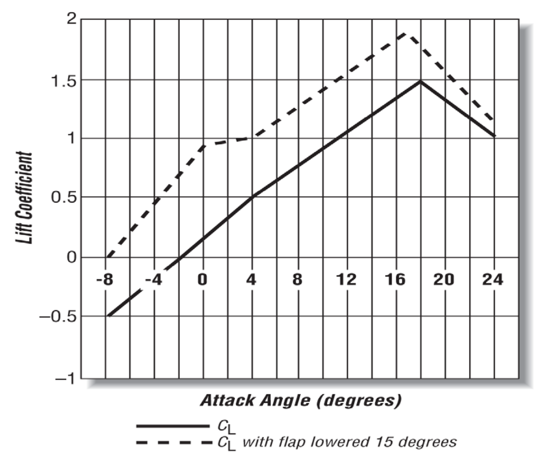

Figure 8: Lift Coefficient vs Angle of Attack [3]

Other variables are considered that affect lift in the FDMs, such as air density, component surface area, and wing aspect ratio. Wind is another external force that can also act on the airplane, but the FDMs in this thesis do not consider wind [2], [3].

Gravity is based on the airplane's mass and the acceleration of gravity, which is a constant -9.8 m/s^2. So, gravity can be calculated by the mass of the airplane multiplied by -9.8 m/s^2, which is added to the force acting on the z-axis (down). Similarly, thrust, expressed in pounds, is force added to the x-axis (forward). Without thrust propulsion, the airplane cannot make lift [3].

2.1.6.3 Equations of Motion

With forces considered, the equations of motion are integrated so that the simulation progresses through time. The equations of motion seek to calculate the airplane’s velocity in each step of time, which can be used to calculate the airplane's updated position. The velocity is the speed and direction of the airplane [3]. Bourg [3] lays out the steps to calculate the equations of motion, where first the mass properties must be calculated, and the forces and moments must be quantified and summed:

1. Calculate the body's mass properties
2. Identify and quantify all forces and moments
3. Take the vector sum of all forces and moments
4. Solve the equations of motion for linear and angular displacements
5. Integrate with respect to time to find linear and angular velocity
6. Integrate again with respect to time to find linear and angular displacement

In computer simulations, the equations of motion are approximated by solving nonlinear Ordinary Differential Equations (ODE) using numerical integration algorithms [13]. The integration of the equations of motion is considered an approximation because there is always some error introduced depending on the technique chosen [3]. Simply stated, the integration of the equations of motion, given everything else mentioned, approximates the future velocity and position based on the previous values from the last frame of the simulation [19].

The amount of finite time chosen to represent a frame (i.e., delta time) in the real-time simulation loop is important in numerical integration. A correctly chosen time step will result in stable numerical integration. Stability has to do with how closely an approximate solution converges to the exact solution [3]. A smaller time step will give a result closer to the exact solution, but this also means that there are more steps and a larger buildup of numerical precision error. On the other hand, there is a breaking point when the time step is too big where the simulation crashes [20]. So, there is a practical balance as to what the time step should be to achieve a stable solution and avoid a potential buildup of error [3].

There are multiple methods to implement numerical integration to calculate the equations of motion. Using Euler's method is by far the most common numerical integration technique [13]. In Bourg's model, the basic Euler's method for integration is applied [3]. In Palmer's model, a simplified and more accurate version of Euler's method called the 4th order Runge-Kutta method is used [2].

An important role of the equations of motion is that the orientation of the plane is tracked. Bourg's FDM uses a quaternion, which is a way to represent the orientation of something about three axes. Quaternions get updated with new Euler angles in each time step to reflect the new orientation using the angular velocities of that instant [3]. However, Palmer's FDM uses 3x3 rotation matrices to transform the external forces, which are evaluated parallel and normal to the velocity vector of the airplane, into an *xyz* axis direction [2].

2.1.6.3.1 Simulation Loop Methods

Another consideration in choosing a time step when integrating the equations of motion is that there are different design strategies to employing the simulation loop. The easiest way to manage the simulation loop is to loop as fast as possible with no time step. The problem with looping as fast as possible is that there is no control. If someone is using a fast computer, the simulation will go too fast, and if someone is on a slow computer, it will be too slow [21]. The next simplest way to manage the loop, and the strategy chosen for use in this thesis, is a fixed time step. A fixed time step is ideal when it is known that the update loop takes less than one frame worth of real-time [20]. For example, if the desired FPS of the simulation is chosen to be 30 and that equates to a time step of 33 milliseconds (1 / FPS = seconds delta time), then it must be reliably known that the real processing time will take less time than that. And for the remaining time, the program will go to sleep until it is time for the next update [21]. The more advanced way to employ a simulation loop is to have a variable time step. It involves choosing a time step based on how much real processing time the current update takes versus the previous loop's real processing time [21]. This strategy, however, introduces some issues, which are not important to explain here.

2.1.7 Fidelity

With the definition of an FDM stated, fidelity is the degree to which a flight simulation's characteristics match that of the real world [22]. In this research, the FDMs are characterized medium "fidelity," because the flight simulation characteristics match that of real life, but are not computationally expensive, and require minimal data [23]. One may think that a flight model with the highest fidelity, or accuracy, is the best – this is not always true. Advanced flight modeling requires large, detailed sets of data about the stability and performance of an aircraft that it is modeling. Wind tunnel data analysis is an example of the kind of data needed for a high-fidelity flight model. This data is not readily available and often costs hundreds of thousands of dollars [24].

2.2 Entity-Component-System

The FDMs are organized and re-implemented in code as a Data-Oriented Design (DOD). DOD is a programming paradigm that is focused on how data is oriented in memory. The Entity-Component-System architecture is a specific implementation of the DOD. ECS is a viable alternative to organizing code in software compared to the ubiquitous Object-Oriented Programming (OOP) design. The ECS design is popular in designing game engines. The Unity game engine, for example, uses the ECS design pattern [25]. ECS focuses on the composition of data, whereas OOP focuses on inheritance and hierarchies. ECS gets rid of the troubles that arise with inheritance, improves performance, and improves ease of code development [9].

2.2.1 Object-Oriented Problem

Inheritance is an important use-case for OOP. Inheritance allows the creation of new classes that inherit states and methods from another class with the ability to add states or methods to the new class [9]. The problem with inheritance is that it couples classes together and creates hierarchy chains. In large codebases used in video game engines, for example, hierarchies can become large and complex with code highly dependent on other code. If one were to alter a parent class that everything inherits from, there would be a ripple effect across the hierarchy [9]. Complex hierarchies make code readability, maintenance, and the addition of features sometimes a pain with OOP. However, this is where ECS shines.

Along with code flexibility, performance is an issue with the OOP design in terms of memory-management of the cache. When a class instance, or object, is instantiated, it is not most efficiently put into the cache. When an object is created, all its data is put into the cache next to all the other object's data. This storage is a problem because if a thread shares an object's data, there is not always enough room on the cache for all the data that the thread wants to work on. And when unneeded data is in the cache, the chance of a cache miss increases [9]. This cache usage is suboptimal for performance and ECS improves upon this.

2.2.2 ECS Explained

In an ECS architecture, because it is based on the DOD paradigm, there are no hierarchy levels – instead, there are these pieces called Entities, Components, and Systems. An Entity represents concrete objects in the world, like a bullet. Entities on their own are useless; Entities are simply unique IDs used to access data elsewhere [9]. Components hold the Entities data, commonly stored as a vector in memory, and represent a feature. For example, Position is a Component of an Entity. The Components called Position, Render, and Spawning may be added to the Bullet Entity. The mixing and matching of Components can create unique Entities. A Player Entity can be created by adding the Components: Position, Render, and Health. Systems are what give functionality to an ECS architecture. A System could be created called UpdateRender, which handles all of the rendering functionality of the world. Or a System called UpdateHealth could handle all of the health functions. Systems have something called a filter. Filters check for every Entity, which has some combination of specified

Components. For example, the UpdateRender System's filter wants to find every Entity with the Render Component and Position Component. Once these Entities are found, they are returned to the UpdateRender System to perform some job on it [26]. The example visualized:

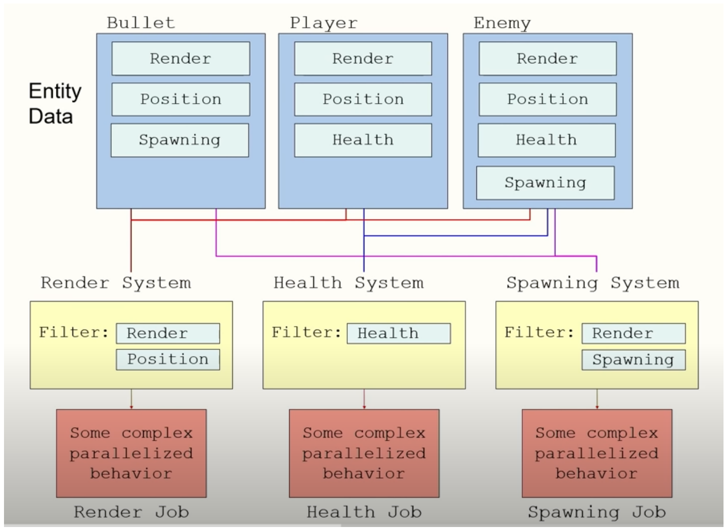

Figure 9: Entity-Component-System Visualized [26]

As seen in Figure 9, the Systems are separate from the Components. The Systems, when performing a job, does not care what Entity it is operating on. The whole point is that the data Components are completely separated, or decoupled, from the Systems functionality. By separating code dependencies, it is simple to add new Systems or Entities without tweaking anything else in code – a benefit that OOP does not offer. This decoupling also makes it simple to implement concurrency where multiple Systems can be working on an Entity concurrently. Also, because a System does a job on specified data Components, it only iterates over Entities that have the necessary Components [27].

The ECS design also optimizes performance by more efficiently managing memory. It does this by laying out the data in the cache more efficiently than the OOP design does. The goal for any software is to retrieve data from cache as quickly as possible, and this is an aspect that is easily overlooked when choosing a certain design approach to a program because it happens in the background. ECS minimizes cache misses by not having data organized by classes, and instead of having data broken into smaller chunks – the data Components, which are stored as a vector. For example, the Render System only needs the Render and Position data for all the Entities. Then, only that required Component data that is needed to perform the System is pulled into the cache for that Entity being worked on. Now the cache holds only what it needs, and does not bombard itself with all the data that an entire class in OOP holds. The vector which stores the Component data in the cache is guaranteed to hold data in each index. This usage of cache is good for performance because no time is wasted determining if an index has null data, and therefore pulling more data into the cache [9].

2.2.3 SPECS Framework

Specs Parallel ECS (SPECS) [1] is an implementation of an ECS for the Rust programming language. SPECS is developed by and used by Amethyst, which is an open-source game engine written in Rust [27]. SPECS is a Rust crate, which is a package that is imported into code for use. SPECS lays the code framework to create Entities, Components, and Systems and set up the ECS architecture.

With SPECS, Components are created similarly to how a regular Rust structure type is created, but the structure created is specified to be implemented as a Component using a vector as its storage type. Systems are created similarly to functions, but when created, it is specified what Components are needed by the System to access. This specification of Components to be used by the Systems determines what Entities the System operates on [27].

The Components specified for a System are given either WriteStorage access or ReadStorage access, designated if that data can be mutated or not, respectively. The ReadStorage or WriteStorage access type designation is important because it determines what Systems may run in parallel. Only one System may be accessing and writing to data at one time to avoid data races during parallel execution. So, if two Systems require the same WriteStorage access to a Component, they cannot be run in parallel to not write over the same memory [27]. This is called a write-write conflict. Two other conflicts exist, read-write and write-read, where two Systems require access to the same Component, but one needs to read data and the other needs to write data. In either case, the Systems cannot run in parallel because the data between the Systems would be inconsistent. However, SPECS can detect conflicts and automatically parallelizes execution based on which Systems may be run in tandem with each other. The dispatcher is the feature in code that is created to manage the System's execution order. Although in the case of a read-write or write-read conflict, the programmer can specify which System to execute first to avoid non-determinism. This specification is commanded to the dispatcher in code to schedule the System execution order [27].

SPECS has a feature called Resources, which allows shared data to be used between Systems [27]. Resources are not part of a Component; they are standalone data. Resources are useful to avoid global variables.

SPECS retains the attributes that make ECS attractive to implement in software. SPECS decouples code into Systems and Components – making software easily extensible and maintainable. Also, SPECS makes for highly performant code by improving cache efficiency by only bringing necessary data into the cache and utilizing parallelization when possible [27].

2.2.4 ECS Overall

The ECS architecture has proven itself as a dominant design strategy in improving game programming. It allows simple parallelization of complex Systems, simple code modification, and quick access to the data that matters. With all of these useful attributes, however, ECS does come with some downsides. To go from coding OOP to ECS, one must change their way of thinking because there is no inheritance, which most programmers are already familiar with. Also, planning ahead when coding in ECS is necessary; one must know what Systems use what Components, and changing Components after the fact is not ideal. Furthermore, since Systems are sometimes dependent on each other to complete, it may be difficult to add another System in the middle of two consecutive Systems. Planning could help to avoid that situation, however. Overall, ECS introduces some excellent benefits once the programmer wraps their head around the architecture.

2.3 Rust

Rust is the programming language being leveraged to build the Flight Dynamics Models. It is the language of choice given its advantageous features: memory safety and performance. Rust was released in 2015 by Mozilla, which has been behind the language's success [28]. Rust guarantees that a program will be memory-safe, with no invalid data accesses, while remaining as fast as other popular programming languages [12]. The C++ programming language, for example, is a systems-level language, giving developers complete access to data inside memory, with no memory safety precautions. This access level is only great when the programmer is an expert on writing code that is free of memory issues – this is hard to do. On the other hand, the Python programming language takes away a developer's access to low-level memory, but does provide complete memory safety by using a garbage collector to clean up unneeded memory – but this imposes a runtime cost [12]. Rust is unique because it is a systems-level language, it does not have a garbage collector, and it still guarantees memory safety. So, Rust provides control, is fast, and protects against memory issues such as data races and dangling pointers [12]. These features built-in to Rust make it the choice programming language for building a FDM.

The safety assurance of Rust is accomplished by its system of Ownership, Borrowing, and Lifetimes. All languages have a way to manage memory while a program is running. Some languages have a garbage collector that actively checks for memory no longer being used during runtime. Other languages make the programmer manually allocate and then free memory. Rust takes neither approach; it instead has something called a "borrow checker." At compile time, the Rust borrow checker will check that all data access is legal and valid [12]. The borrow checker deals with three important Rust concepts: Ownership, Borrowing, and Lifetimes.

2.3.1 Safe Rust and Unsafe Rust

Before discussing what specific features make Rust a “safe” programming language, it is important to know that there is an “unsafe” side of Rust, like C++. By safe, it is meant that the code is completely bug-free of memory issues, and undefined behavior such as data races (i.e., two processes writing data in the same memory location at the same time), and dangling references (i.e., accessing unintended data). The unsafe side of Rust will let the programmer do things that are not safe. And sometimes it is necessary to do unsafe things. For example, sometimes it is required to go into the unsafe mode to dereference raw pointers, which are possibly invalid, or out of scope at access [29]. In safe Rust, this is impossible to do because of the protection feature implemented called Lifetimes. The other implemented features, Borrowing and Ownership, also empower safe Rust. These features are discussed in this chapter now. Overall, a great aspect of Rust is that it allows the programmer the choice to write code in safe mode or unsafe mode, whereas in other languages, that mode is unchangeable.

2.3.2 Ownership

Ownership refers to the fact that every use of a value needs to be valid and cleaned up when it is done being used. Ownership is accomplished with three rules that the programmer must follow in Rust [28]:

1. Every value has a variable which is called its owner
2. There may be only one owner at a time
3. When an owner goes out of scope, the value is dropped

2.3.3 Borrowing

Borrowing is the next important concept in Rust. Borrowing allows data to be "borrowed" from one owner (variable binding) to another owner and accessed for a moment without changing the Ownership of that data. When Ownership of memory is transferred to another owner, the original owner cannot legally be used, so Borrowing is the solution. The borrow checker statically guarantees that references always point to valid objects. So, if a reference to an object exists, it cannot be destroyed. There are two rules to Borrowing [28]:

1. Any borrow must last for a scope no greater than that of the owner,
2. One may have one or the other of these two kinds of borrows, but not both at the same time:
    1. One or more references to a resource
    2. Exactly one mutable reference

These rules avoid the possibility of a data race because although there can be more than one pointer to a resource, only one can access and change that resource at a time. This is good for performance and uses less memory overhead [12].

2.3.4 Lifetimes

The next important Rust concept is Lifetimes. A value has a Lifetime, which is the period that accessing that value is valid to do [28]. A Lifetime starts when a value is created, and that Lifetime ends when it is destroyed. Rust's compiler, or borrow checker, is smart enough to know a value's Lifetime in many common circumstances [30]. In these common circumstances, the Lifetime of a value does not need to be explicitly written, but otherwise, a Lifetime parameter needs to be written. The borrow checker tries to limit the period of a Lifetime. A value should not be alive any longer than it needs to be because, otherwise, it could be referenced by accident and cause a dangling reference, which is a reference to unintended data [12]. So, to keep some piece of data alive for only as long as completely necessary, a Lifetime is specified for that data.

2.3.5 Testing

Rust supports features to write software test functions within the language. Rust supports unit testing and integration testing, both of which are used in this research to check code correctness. Rust tests set up a segment of code to be executed where expected results and compared to the actual results. Depending on the equivalency of the results, the test will pass or fail. Test functions are marked with the `#[test]` attribute [28].

Unit tests are written in the same code file as that of what they are testing. Unit tests are typically used to test units of code in isolation from the rest of the code to see where code is or is not working [28].

Integration tests are external to the library that it is testing. An integration test uses code in a project by importing and calling functions. They are used to verify that many parts and functions of the library work together correctly, whereas a unit test covers a smaller unit of code or a single function. Integration tests are necessary because, although all unit tests may pass, integrating a larger segment of code may fail. Integration tests are built in a *tests* directory of the software package [28].

2.3.6 Crates

Crates are essentially libraries added to a Rust project for additional functionality written by public users. Crates do not come pre-packaged when Rust is installed on a machine; they must be manually installed [28]. In this thesis, multiple crates are used to code the final result, but the most notable crate is SPECS, a framework to easily program the ECS design in Rust.

The Rust community is also active and quickly growing, with over 50,000 crates available to download [31]. Rust makes it easy to import these crates into code and begin using them, and the crate to implement ECS into code, SPECS, is no exception. Only two steps are needed to get started with a crate. First, add the crate dependency to the `cargo.toml` file, which is in every Rust project. Second, add a line at the top of the code file to indicate that this crate is being used. Now the desired crate can be utilized in code.

2.3.7 Rust Overall

Overall, Rust can hold its own against more popular languages, and the learning curve of the language is met with staggering benefits in safety and performance. And with safety dealt with by the borrow checker at compile time, the programmer does not need to be concerned with memory issues. Simultaneously, there is no cost in performance at runtime because memory is naturally checked at compile time without a garbage collector. The downsides to Rust are minimal, but they do exist. For one, compile time is longer because memory is checked at compile time. Also, the compiler is very strict. New Rust programmers often “fight with the borrow checker” to get their code to compile because it is more difficult to write code that abides by Rusts rules [12]. So less time is spent fixing bugs, and more time is spent getting the code to compile. However, the compiler is like a safety net because once the code compiles, the programmer knows the result is safe and free of memory errors.

2.4 FlightGear

FlightGear [4] is an open-source flight simulator application. However, in this thesis, it is being used as a standalone visual system. An advantage to using FlightGear’s visualization system is that no extra processing time is required from the flight simulator application to render graphics – this task is delegated to FlightGear [6].

Although it has its own built-in FDM, FlightGear is easily configurable to use an external, custom FDM instead. FlightGear can be configured to be interfaced or communicated with via network packets to achieve the desired display output. In this case, these packets of data are sent from the ECS patterned Rust flight simulator application to FlightGear. The packets contain the position as geodetic coordinates and orientation as Euler angles. FlightGear reads these packets and displays the airplane.

To properly configure FlightGear for the simulation, it must manually be told to accept the new external FDM and listen for the respective packets [32]. This configuration is done from the command prompt upon running the executable. The command-line arguments are listed in Appendix A: Configuring FlightGear as a Visual System.

With FlightGear configured, it can process the received network packets constructed by the Rust application. FlightGear requires that these packets are defined in a particular format to understand them. Before being converted to bytes, the format of the packet must be a structure, which is outlined in FlightGear's source code [4]. The structure contains all of the data needed to define an airplane and how it flies throughout the simulation.

2.5 Networking And Rust

Networking is used to send packets of data to FlightGear so that it can display the desired simulation. To communicate with FlightGear, two sockets are required at each end of communication. In this case, the type of socket being used is the Datagram socket. The protocol involving Datagram sockets is called the User Datagram Protocol (UDP). This protocol is considered "connectionless" because an open connection does not need to be maintained. Packets containing bytes of data may be built and sent to a destination without a connection needing to be established prior [33]. FlightGear, when commanded on startup, can open a socket to receive UDP packets on.

In Rust, a UDP Socket is created and binded to an internet protocol address and port on the local machine. After this, data can be sent and received from any other socket address. The socket created can then be connected to the other socket with its internet protocol address and port. From here, data can be sent and received with a function call [28]. One important thing to be aware of is that before data packets are sent out to

FlightGear, the literal order of the bytes needs to be converted from least significant byte first (i.e., little-endian) to most significant byte first (i.e., big-endian) [33].

2.6 Summary

This background chapter began with a section on the basics of the physics of flight modeling. A Flight Dynamics Model (FDM) is defined as the set of math equations that compute all of the physical forces (i.e., lift, drag, gravity, thrust) acting on a rigid-body airplane [10]. The equations of motion are integrated with respect to time to compute an airplane's behavior with a FDM [3]. The equations of motion determine the airplane's position and the airplane's velocity over time [13]. An airplane's components, such as the wing's airfoil properties, affect how the forces act on the state of the airplane. When an airplane is rotating, the orientation can be described by the three Euler angles representing the *xyz* axis on a coordinate plane pitch, roll, and yaw [3]. A Flight Dynamics Model's process of calculating the equations of motion can be explained in these steps:

1. Calculate the body's mass properties
2. Identify and quantify all forces and moments
3. Take the vector sum of all forces and moments
4. Solve the equations of motion for linear and angular displacements
5. Integrate with respect to time to find linear and angular velocity
6. Integrate again with respect to time to find linear and angular displacement

The ECS architecture is a specific implementation of the DOD programming that is focused on how data is oriented in memory. ECS is a viable alternative to organizing code in software compared to the ubiquitous Object-Oriented Programming (OOP) design. However, ECS also introduces performance and ease of coding benefits. ECS organizes data so that cache misses are minimized, thereby improving code performance. Also, ECS allows easy parallelization due to how it decouples functionality and data in code. Decoupling also eases code development because pieces of code are no longer dependent on other pieces of code to work. Adding, removing, or editing data or functions does not affect other pieces of code, making code development and maintenance an easier process.

Rust is the programming language of choice used in this research. Rust provides some advantageous benefits: memory safety and performance. By checking code during compile time, Rust guarantees that a program will be memory-safe, with no invalid data accesses, while maintaining systems-level programming speed [12]. Rust implements its system of Ownership, Borrowing, and Lifetimes in order to accomplish memory safety. Rust also provides a testing framework that other languages do not employ. This testing framework allows the programmer to easily run unit and integration tests to check that code works as expected. Although Rust’s compiler is not forgiving, the developer knows the code is safe and performant once the code is compiled.

FlightGear is an open-source flight simulator being used as a visualization system in this research. FlightGear can be configured to use an external FDM. UDP packets can be sent and interpreted by FlightGear in order to display an airplane flying. The UDP packets sent only need to contain position and orientation data of the airplane, and that data can be sent to FlightGear to display. The networking to send UDP packets can be accomplished via Rust.

Overall, the background is critical to understanding the research's inner workings in Chapter III. All of this information is tied together to reach the result of a ECS/Rust-based interactive flight simulator using a published FDM that is displayed in FlightGear.

III. Methodology

This chapter discusses the research methods used in this thesis, including what has been accomplished and how it has been accomplished. First, the process of re-implementing Palmer's Flight Dynamics Model [2] and Bourg's Flight Dynamics Model [3] into an Entity-Component-System (ECS) using the Specs Parallel ECS (SPECS) [1] crate is described. Next, how the resulting Flight Dynamics Models (FDM) were verified to work as expected versus the original code is explained. The FDMs were put through Rust integration tests that set up a series of flight scenarios performed on both the original FDMs. The data collected from performing the original was used to compare to the integration test results. Next is the explanation of how the two FDMS were benchmarked for performance and then compared using a Rust crate called criterion. Next, how the FDMs were set up as flight simulators is explained. Additional Systems and Components were developed to handle getting keyboard input and send User Datagram Protocol (UDP) packets for visualization within FlightGear. These additional Systems and Components were added to the ECS-based FDMs to create interactive real-time flight simulators.

3.1 Building the Flight Dynamics Models

This section broadly describes the re-implementing of the two FDMs, which overall share more similarities than not. However, section 3.7 describes the extension of the FDMs into flight simulators, which adds additional Systems and Components.

The approach to building the FDMs using the Rust programming language and designed with the ECS first began with understanding basic flight physics and the Rust programming language. General flight physics knowledge was needed to follow along with what was going on in the FDMs. Both of the textbooks describing the FDMs, *Physics for Game Programmers* [2] by Grant Palmer and *Physics for Game Developers* [3] by David Bourg, were the primary references.

Regarding programming in Rust, the two most utilized resources for writing code in this thesis was the official Rust book, *The Rust Programming Language* [28], and the official code collection called *Rust by Example* [34], which showcases executable code snippets. The first programming task in Rust for this thesis had to do with networking. The task was to send a UDP packet in Rust to FlightGear to interface it. To help accomplish this, FlightGear [4] offered source code that does exactly this in C++. Although this task did not directly progress the development of the FDMs, it taught Rust basics and set up the networking required for flight simulation. The next task was to translate both FDMs from their native languages into Rust.

The first FDM built was the Palmer-based version. The original FDM is described using the Java programming language in Palmer's textbook in Chapter 10: Airplanes. However, Palmer also wrote his FDM using the imperative C programming language. The C version was used to re-implement the code in Rust. This C code is available on the book's GitHub webpage [35]. The second FDM is described in Bourg's textbook and is written in C++ with an OOP design. Multiple chapters of the book come together to describe this FDM – Chapter 7: Real-Time Simulations, Chapter 11: Rotations in 3D Rigid-Body Simulators, Chapter 12: 3D Rigid-Body Simulator, and Chapter 15: Aircraft.

Similar to before, the FDM code authored by Bourg was also meticulously re-implemented into Rust. The C++ source code is available in the book [3].

From a translational standpoint, both models can cause positional translation on all three axes – meaning they can fly in any direction. Also, from a rotational standpoint, the models can rotate freely about all three axes.

The most notable difference between the Palmer model and the Bourg model is how they handle airplane rotations. Palmer's FDM uses 3x3 rotation matrices to calculate the orientation of the airplane. Also, to create a rotation, the Palmer model modifies the Euler angles directly. Bourg's FDM does not use rotation matrices; it uses a quaternion, an alternative to representing rotations with matrices. Where a rotation matrix requires nine values, a quaternion needs four. Bourg's model also activates specific components of the airplane, such as the ailerons, to modify the Euler angles and thus create a rotation.

Concerning implementing the FDMs with the DOD paradigm using the ECS architecture, programming was not begun until after the FDMs worked in Rust after being translated. Ideally, one would begin programming their software from the ground up as an ECS, but it was simpler to convert the code directly from the FDMs native language to Rust before it would be broken up into Systems and Components representing the ECS. From here, the FDMs were restructured into an ECS design using the SPECS framework, which supports the creation of Entities, Components, and Systems in software. There were very limited resources available describing the process of coding the ECS design, especially in Rust. The ECS references used in this thesis were predominately YouTube videos published by Unity [25], [26], a high-profile proponent of the architecture, and *The Specs Book* [27], which is the official documentation on the SPECS framework for Rust.

The two FDMs were identical in their ECS construction: they consisted of one Component, one System, and one Entity. The graphic below describes the ECS organization for both FDMs between the airplane Entity, DataFDM Component, and EquationsOfMotion System. An airplane entity has a DataFDM component. The EquationsOfMotion System's filter specifies the DataFDM Component. So, the job of the EquationsOfMotion System is processed on that airplane Entities Component data.

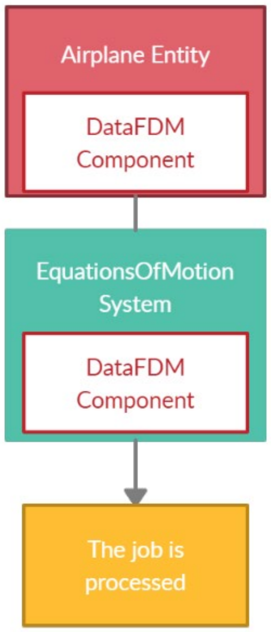

Figure 10: Flight Dynamics Model Entity-Component-System Organization

3.2 Palmer-based Flight Dynamics Model

Palmer offers three versions of his FDM in three different programming languages – C, Java, C Sharp – all of which are fundamentally the same. Although, his Java and C

Sharp (C#) versions implement a Graphical User Interface (GUI) to employ the FDM as an interactive flight simulator. The Java and C# versions also calculate a few extra flight measurements in the equations of motion that the C version does not include: climb angle, climb rate, and heading angle (i.e., yaw angle). With all this considered, the C version was still chosen to translate the FDM into Rust because of experience reasons and because this version omitted a GUI and output the flight data to the console window. However, the calculations of the climb angle, climb rate, and heading angle measurements originally omitted in the C version were added to the Rust re-implementation.

The FDM supports the modification of the airplane's amount of throttle, bank angle, attack angle, and flaps. These modifiable values directly affect the equations of motion, and therefore the airplane's flight behavior to achieve translation and rotation. Although there is no modification solely to yaw the airplane, the airplane's bank motion causes yaw. Section 3.7 explains how the FDM is extended to be an interactive flight simulator – this section only describes this raw FDM.

The `palmer.rs` code file, which contains the main function, is located in the *examples* directory begins by importing the SPECS crate, along with all of the modules containing the Components and Systems. In the main function, the simulation world was created. The Components were registered into the world. A Frames Per Second (FPS) value was chosen, and the time step based on that FPS was determined. That time step (i.e., delta time) was added to the world as a Resource for the Systems to use. A dispatcher was opened to manage the execution of the Systems. The airplane Entity was then built with the mass properties containing the performance data representing a Cessna

Skyhawk. The other data parameters when creating the airplane Entity are the throttle, attack angle, bank, and flaps. These four values can be altered for the desired flight effect. Finally, the main simulation loop was made to process each frame at the desired FPS. The simulation loop can run at that FPS because it sleeps any extra time remaining in the time step after the processing of each frame.

3.2.1 Resource

This FDM uses one Resource: DeltaTime. This Resource represents the time step used in the simulation loop, which is based on the FPS rate chosen. DeltaTime is used by the EquationsOfMotion System when computing the Ordinary Differential Equations (ODE).

3.2.2 Component

The FDM is set up with one Component: DataFDM. The DataFDM Component holds all of the data describing the airplane, which is a rigid body by definition. The EquationsOfMotion System uses this sole Component.

The DataFDM Component specifically holds the values computed by the EquationsOfMotion System, as well as the values that can be altered for a flight: throttle, bank, alpha (i.e., attack angle), and flap. DataFDM also holds all of the mass properties and performance data, such as airfoil performance data. This data is stored in a structure called PerformanceData, which is stored inside of the DataFDM Component.

3.2.3 System

The FDM contains one System that computes the equations of motion: EquationsOfMotion. This System requires WriteStorage access to the DataFDM Component.

This System uses a 4th order Runge-Kutta method to integrate the equations of motion. The Runge-Kutta method involves running a function four times that load the ODEs [2]. The ODEs computed at the end of the Runge-Kutta method contain the flight data outputted to the console: airspeed, distance traveled, and altitude. The equations of motion also take into account the amount of throttle, bank angle, flaps deflection, and attack angle.

The function, called plane_right_hand_side, which is executed four times in the Runge-Kutta method, begins by computing air density. Next, the power drop-off factor is computed, which considers that the thrust generated by the propeller decreases with an increasing altitude [2]. The thrust force is then computed based on the throttle applied and the engine power. The lift coefficient is computed based on the airfoil properties and the airplane's current angle of attack. The presence of the flaps affecting the lift coefficient is then considered. The lift force is then calculated, taking into account the lift coefficient, air density, and wing area. The drag is also calculated with the same considerations. The forces – thrust, drag, and lift – are then considered together and converted into x, y, and z components while examining a rotation matrix based on whether the airplane is banked or climbing. Gravity force based on the mass of the airplane is then added to the z component. However, gravity is only applied if altitude is greater than zero. Lastly, the airplane's acceleration on the x, y, z components is calculated by dividing the forces by the airplane's mass. The accelerations on x, y, and z are then multiplied by the DeltaTime Resource to get the airplane's velocities. These values are loaded into the vector containing the ODE results. This sequence is executed four times. The results from each of the four iterations are taken to determine a final average slope approximation. The airspeed of the frame is calculated by taking the magnitude of the velocity. The calculated distance is then calculated in relation to a fixed origin point (i.e., the initial position).

3.3 Bourg-based Flight Dynamics Model

Bourg's model defines custom classes in C++ that are used to represent a vector, quaternion, and matrix. These custom classes were reinterpreted into Rust modules and were used in this FDM re-implementation.

This FDM can evoke operations that affect the airplane's roll, pitch, yaw, thrust force, and flap deflection. Unlike the Palmer-based model, where these aspects are assigned a numerical value and directly affect the calculation of the equations of motion, this model evokes operations on the airplane's components that affect a specific aspect of flight. For example, if the pilot rolls, the model will stimulate the aileron components on each wing of the airplane to be activated – and this activation of certain components is what is considered in calculating the forces acting on the airplane. This model does not assign a value directly to roll, pitch, yaw, and flaps; instead, it derives these values based on what the components are doing and how they are affecting forces. The exception is thrust force; thrust force is simply assigned a numerical value that affects total force on the airplane. Section 3.7 explains how the FDM is extended into an interactive flight simulator – this section only describes this raw FDM.

The `bourg.rs` code file example begins by importing the SPECS crate, along with all the Component and System modules. In the main function, the simulation world was created. The Components were registered into the world. A FPS is chosen and the time step based on that FPS was determined. That time step time was added to the world as a Resource. A dispatcher was opened. Then, the airplane Entity was built with the mass properties containing the performance data of the no-name airplane described by Bourg. Finally, the main simulation loop was made to process each frame at the desired FPS.

3.3.1 Resources

This FDM uses three Resources: DeltaTime, MaxThrust, and DeltaThrust. The DeltaTime Resource represents the time step used in the simulation loop, which is based on the FPS rate chosen. MaxThrust defines the maximum thrust force capacity that can be produced by the airplane. DeltaThrust is the increment and decrement defined when the thrust force is increased and decreased during the simulation frame. The dt is added to the world as Resource. MaxThrust is set at 3000.0 pounds of force but could be altered if one desires. The DeltaThrust is also able to be modified but is set at 100.0 pounds of thrust. These set values were chosen because that is what Bourg defines in his code.

3.3.2 Component

The FDM is set up with one Component: DataFDM. This Component holds all of the data describing the airplane. The EquationsOfMotion System uses this Component.

The DataFDM Component holds the values computed by the EquationsOfMotion System (e.g., position coordinates and quaternion). DataFDM also holds all of the mass properties and performance data of each of the airplane's components. This data is stored in a structure called PointMass, which is stored as a vector of PointMass elements inside of the DataFDM Component.

3.3.3 System

The FDM contains one System that computes the equations of motion: EquationsOfMotion. This System requires WriteStorage access to the DataFDM Component.

This System begins by resetting the activation of all of the components because it is not known if a roll, for example, is still occurring from the last frame. Now, for interactive flight simulation, this is where the handling of flight control input would happen, but this part is discussed in section 3.7. The next step is calculating the forces and moments acting on the airplane in the function called calc_airplane_loads. This function involves looping through the seven lifting elements of the airplane to find the direction that the forces are acting, finding the angle of attack, computing the lift and drag coefficients, and calculating the moment about this element's center of mass. Once that loop is done, the thrust force is applied, and forces are converted from body space to Earth space to apply the gravity force.

From here, the acceleration of the airplane is calculated in Earth space. Then, the velocity of the airplane is calculated in Earth space. The velocity is then directly used to find the displacement in the position since the last frame. That displacement is then added to the current position coordinates to determine the new position in Earth space. In terms of rotations, the angular velocity in body space is then computed. The angular velocity is then used to determine the quaternion, which can be extrapolated to give us the Euler angles. Finally, the airspeed is calculated by getting the magnitude of the velocity. These are all of the general steps taken in the System to compute the equations of motion in this FDM.

3.4 Palmer-based FDM Equivalency Verification

With the construction of a Rust-based, ECS patterned FDM based on Grant Palmer's works in [2], it was required to find out if it indeed worked as expected. When flying the airplane in FlightGear, the flight mechanics look to be correct from the naked eye. Although to ensure this re-implemented FDM works properly, it was investigated if it replicates the same realistic flight data generated by the C-based FDM original source code [35]. This investigation consists of a series of different flight scenario tests between both FDMs and checked if the flight data from the last frame generated by our FDM matched the flight data generated by the original. The flight data used in the comparison was: position coordinates and airspeed.

To implement the tests, Rust supports integration testing. For each test, the original Palmer FDM was compiled and ran with its default airplane configuration flying under some initial conditions to get some resulting output. This output was then copied over to the Rust test, where the same test conditions for the original scenario were performed on our FDM, and both outputs were compared.

The initial flight parameters include throttle, bank angle, attack angle, and flaps. These parameters are the values able to be modified to affect flight behavior as defined by Palmer. The parameters stay constant throughout a test. The parameters tested were bank, attack angle, and flaps. Only one side of each flight control was tested when applicable (i.e., bank right vs bank left) to reduce redundancy. There is overlap in what is being tested, however. For example, when testing bank, a positive angle of attack and a positive throttle is applied, along with bank.

The execution time of every test was 60 seconds of simulation time. This was enough time for the airplane to take off and fly for about 30 seconds. A time step of 0.1 seconds was chosen for every test, which equates to 10 FPS. This time step was chosen because Palmer uses the same value in his Java code.

A small tolerance, or epsilon, of numerical floating-point error was considered in the results to determine accuracy. A margin of error tolerance is necessary due to the unavoidable error that floating-point computations create because computers cannot exactly represent numbers in binary and must round answers to the nearest representable number. The Rust crate called float-cmp is used to implement the epsilon consideration in the results [36]. The results were tested to the smallest tolerance possible without failing the integration test.

The original C-based FDM was compiled with the GNU C compiler, version 7.5.0. The Rust/ECS FDM was compiled with rustc version 1.47.0. The tests were performed on a Lenovo ThinkPad T440p machine, running Windows 10, with an Intel i74800mq processor.

3.4.1 Test One (Angle of Attack)

This test was designed to find out if a positive degree of angle of attack, while the airplane is accelerating at full-throttle, works as expected.

Flight Control Instructions
Throttle (%): 100
Angle of Attack (deg): 4.0
Bank Angle (deg): 0.0
Flaps Angle (deg): 0.0

Table 1: Test One Palmer-based Flight Control Instructions

3.4.2 Test Two (Bank)

This test was designed to examine if the bank action works properly. The airplane is accelerating at full-throttle. The degrees of bank is set to 10.0, which banks the airplane to the right. The angle of attack is set to 10.0 degrees.

Flight Control Instructions
Throttle (%): 100
Angle of Attack (deg): 10.0
Bank Angle (deg): 5.0
Flaps Angle (deg): 0.0

Table 2: Test Two Palmer-based Flight Control Instructions

3.4.3 Test Three (Flaps)

This test examined the deflection or lowering of the flaps on the airplane. The flaps can either be completely down or at two other angles: 20 degrees and 40 degrees. In this test, the flaps are deflected at 20 degrees. The airplane is set to accelerate at 100%.

Flight Control Instructions
Throttle (%): 100
Angle of Attack (deg): 4.0
Bank Angle (deg): 0.0
Flaps Angle (deg): 20.0

Table 3: Test Three Palmer-based Flight Control Instructions

3.4.4 Test Three (Everything)

The final test incorporates all of the flight control elements: attack angle, bank, and flaps.

Flight Control Instructions
Throttle (%): 100
Angle of Attack (deg): 4.0
Bank Angle (deg): 5.0
Flaps Angle (deg): 20.0

Table 4: Test Four Palmer-based Flight Control Instructions

3.5 Bourg-based FDM Equivalency Verification

Since the focus of this research was building a Rust-based, ECS patterned FDM based on David Bourg's works in [3], it was required to find out if it indeed worked as expected. When flying the airplane in FlightGear, the flight mechanics look to be correct from the naked eye. However, it was investigated if the FDM replicates the same realistic flight data generated by the original C++ FDM described in Bourg’s textbook. This investigation ran a series of different flight tests between both FDMs, intending to check if our FDM’s generated flight data was equivalent to the original in multiple scenarios. The flight data used in the comparison was: position coordinates, Euler angles, and airspeed. This data represents exactly where the airplane is at a given time and how the airplane is oriented.

Similar to the integration testing performed on the Palmer-based FDM, the flight data generated by Bourg's original FDM under some flight parameters was used in a series of Rust integration tests to replicate that same scenario. The initial parameters include coordinate position, airspeed, velocity, and forces. The starting parameter values chosen to be used in these tests are the same values that Bourg sets in his FDM source code. They were chosen because the values are set up so that the plane is in motion as soon as the simulation is started – with velocity, force, thrust, and airspeed defined greater than zero. The starting position coordinates represents a location relative to an arbitrary fixed point. The distance from that fixed point is measured in feet. For example, the starting z-coordinate of 2000.0 represents height, or altitude, in feet from the ground. The starting coordinate point chosen for every test is the same location that Bourg's model used in his code. The starting flight values chosen while testing the C++/OOP FDM and the Rust/ECS FDM are:

Variables	**Initial Values**
Position Coordinates (x, y, z)	-5000.0, 0.0, 2000.0
Velocity Vector (x, y, z)	60.0, 0.0, 0.0
Force Vector (x, y, z)	500.0, 0.0, 0.0
Thrust Force (pounds)	500.0
Airspeed (feet/second)	60.0

Table 5: Initial Flight Data for Each Bourg-based Test

For each test, the two FDMs were given the same flight control maneuvering instructions for the airplanes to follow. Every flight control, roll, pitch, yaw, thrust, and flaps are tested in the six total tests. The first test does nothing once the simulation is started. The following tests focus on a specific flight control, excluding thrust, which is applied in each test. The final test examined every flight control.

The flight controls are implemented into the test for a defined amount of simulation frames. For example, from frames one through five, thrust is increased, and so on. The flight controls have to be implemented this way because the airplane components that affect flight control have to be activated in specific frames of the simulation, and it is not possible to input the flight control perfectly with keyboard input. The flight data at the end of each test execution was recorded for each FDM. Only one side of each of the flight controls (i.e., roll right vs roll left) was tested to reduce redundancy. The execution time of the tests is 30 seconds of simulation time. The FPS is set at 30, which equates to 900 total frames for every test scenario.

Following the Palmer-based equivalency tests, a tolerance was used to account for the numerical floating-point in the results. This tolerance was necessary due to errors created by floating-point number computations because computers cannot exactly represent numbers in binary and must round answers to the nearest representable number. The Rust crate, float-cmp, was used to compare the results within a tolerance [36]. The results were tested to the smallest tolerance possible without failing the integration test.

The original C++/OOP FDM was compiled with Visual Studio 2019 Community Edition, and the Rust/ECS FDM was compiled with rustc version 1.47.0. The tests were performed on a Lenovo ThinkPad T440p machine, running Windows 10, with an Intel i74800mq processor.

3.5.1 Test One (No Flight Control)

This test was designed to determine if the FDM works when absolutely no flight control actions are taking place to maneuver the airplane as it is flying.

Frame by Frame Flight Control Instructions
None

Table 6: Test One Bourg-based Flight Control Instructions

3.5.2 Test Two (Roll)

This test was designed to examine if the roll action works properly. First, the thrust is increased, then the airplane pitches up for 8 seconds (240 frames), and finally, the airplane ailerons are evoked to roll right the airplane for 2 seconds (60 frames). The airplane then holds that rolling position for the remainder of the test. A pitch of 8 seconds was chosen to stabilize the airplane once the simulation is started. 8 seconds is enough time to get the airplane flying straight ahead. When the simulation starts, the airplane dips down dramatically without increasing the thrust and pitching up, so that is why pitching up is necessary before rolling.

Frame by Frame Flight Control Instructions
1 – 5: Increase Thrust
6 – 246: Pitch Up
247 – 307: Roll Right
308 – 900: None

Table 7: Test Two Bourg-based Flight Control Instructions

3.5.3 Test Three (Pitch)

This test was designed to be sure that pitch works. This test applies the pitch control to every frame throughout the entire execution time to ascend the airplane. The pitch flight control evokes the elevators. If the airplane were going faster caused by more than 1000 pounds of thrust force, the airplane could stall – but in this case, the airplane ascends smoothly.

Frame by Frame Flight Control Instructions
1 – 5: Increase Thrust
6 – 900: Pitch Up

Table 8: Test Three Bourg-based Flight Control Instructions

3.5.4 Test Four (Yaw)

This test examined the yaw flight control. First, the thrust is increased for 5 frames to 1000 pounds of thrust total. Next, the airplane is stabilized by pitching up for 8 seconds (240 frames). Once the airplane is flying straight, the airplane rudders are evoked to yaw right for 2 seconds (60 frames).

Frame by Frame Flight Control Instructions
1 – 5: Increase Thrust
6 – 246: Pitch Up
247 – 307: Yaw Right
308 – 900: None

Table 9: Test Four Bourg-based Flight Control Instructions

3.5.5 Test Five (Flaps)

This test examined the lowering of the flaps on the airplane. The flaps can either be up or down during the simulation; the flaps are toggled on and off. The flaps are lowered during the entirety of the test, and that is all.

Frame by Frame Flight Control Instructions
1 – 900: Flaps Down

Table 10: Test Five Bourg-based Flight Control Instructions

3.5.6 Test Six (Everything)

The final test examined the flight behavior from every flight maneuver throughout the test.

Frame by Frame Flight Control Instructions
1 – 5: Increase Thrust
6 – 246: Pitch Up
247 – 307: Yaw Left
308 – 368: Yaw Right
369 – 429: Roll Right
430 – 490: Roll Left
491 – 505: Pitch Down
506 – 511: Thrust Down
512 – 900: Flaps Deflected

Table 11: Test Six Bourg-based Flight Control Instructions

3.5.7 Custom Type Module Verification

Bourg's FDM uses three custom C++ classes that support the creation and operations of vectors, matrices, and quaternions. Like Bourg's FDM, these classes were re-implemented into Rust modules for use in our Bourg-based re-implemented FDM. The classes are described in the textbook's appendixes A, B, and C [3]. These modules were not used in the Palmer-based FDM.

Bourg provides three test code files for each class inside the source code containing the custom classes [3]. In these code files, Bourg demonstrates the various operations. However, Bourg omitted the testing of some operations used in the FDM – those operation tests were added in C++ to get data to compare to.

Although classes, as they exist in C++, do not exist in Rust, they can be simulated in Rust code using a structure and an implementation block. The structure lets us define a custom type, while the implementation block lets us define some implementation of functionality for the structure type, which is a vector, quaternion, and matrix [28].

The classes were re-implemented in Rust to match the exact functionality that Bourg intended. A series of Rust unit tests were devised to ensure that the operations on the vector, matrix, and quaternions were correct. Each unit test checks each operation used by this FDM. Each of these operations was compiled and ran with the original C++ code by Bourg. The resulting output was then used in the Rust unit tests to be compared to the Rust re-implementation output. The operations in the Rust tests were given the same input as what was used to gather the C++ code results.

3.5.7.1 Vector Tests

This list of vector operations checked by the Rust unit tests:

1. Magnitude
2. Normalization

3. Reversal
4. Addition
5. Subtraction
6. Cross product
7. Dot product
8. Multiplication by scalar
9. Division by scalar

3.5.7.2 Matrix Tests

The list of matrix operations checked by the Rust unit tests:

1. Inverse
2. Multiplication by vector

3.5.7.3 Quaternion Tests

The list of quaternion operations checked by the Rust unit tests:

1. Magnitude
2. Addition
3. Multiplication by scalar
4. Division by scalar
5. Conjugate
6. Multiplication
7. Multiplication by vector

8. Rotation by vector
9. Construction by Euler angles
10. Euler angle extraction

3.6 FDM Benchmarking

A benchmark of the Rust-based Bourg and Palmer FDMs was included in this research to provide a relative performance comparison. A Rust crate called criterion [37] was used to implement statistical analysis-driven benchmarking. Criterion can set up group benchmarking comparisons between functions with a command line argument. Criterion also uses Gnuplot to generate detailed graphs of the results. Also, criterion can calculate if a benchmark has regressed or increased in performance compared to the last benchmark.

First, a benches directory was created at the root of the project to set up benchmarking using criterion. In this directory, a code file imports the criterion crate, and imports all of the functions and data necessary to run the two functions that execute the Palmer and Bourg models. Next, the two functions containing the Palmer and Bourg models to be executed and compared are defined. The two functions that are compared are not the ECS-based versions – the airplanes are instead created as an object to be operated on. This decision was made because it is not clear how to implement an ECS structure into a criterion benchmark as it requires functions to compare, and only the relative performance of the raw models is at question. Lastly, a criterion function is defined that describes how the two functions are benchmarked. For this research, it is set up to test the models with a single number of frames to execute for each of the multiple frame rates (i.e., FPS). Multiple frame rates are inputted to determine if the time step between frames affects performance due to the Runge-Kutta or basic Euler methods used either model to integrate the equations of motion. Both the number of frame iterations and multiple frame rates may be modified as desired. However, the benches demonstrated consisted of 100 frames for each 10, 30, 60, 100 FPS bench.

3.7 Building the Flight Simulators

This section explains the extension of the Palmer-based FDM and Bourg-based FDMs into flight simulators. The setup of both simulators is similar enough to share this one section – however, a handful of differences are pointed out. Additional considerations required for simulation not previously mentioned in the sections, 3.2 and 3.3, describing the creation of the raw FDMs are noted.

With the FDMs built as an ECS consisting of a single Component and System, the Systems and Components were added to each FDM that support keyboard input and output of network packets to a visualization system to use in interactive flight simulation. With the ECS architecture, the process of extending the FDMs into flight simulators is straightforward.

Only a few simple lines of codes need to be written to add the new Systems and Components. The code file modules in which the Components and Systems are located need to be imported into code. The new Components need to be registered to the world. Finally, the Systems need to be added to the dispatcher. These are the only steps required to add the Systems and Components to the ECS architecture using SPECS.

Some considerations regarding the body space coordinates representing the airplane's orientations in both simulators had to be made. The coordinate system FlightGear uses is different than the ones used in each FDM. For example, FlightGear depicts a positive yaw degree as yaw to the left. This difference was causing the airplanes in FlightGear not to be depicted properly when maneuvering. So, negations to both FDM's Euler angle values representing rotations had to be done in the MakePacket System, where the flight data is packaged for FlightGear. For the Palmer-based simulator, the yaw angle had to be negated. For the Bourg-based simulator, both the yaw angle and the pitch angle had to be negated. Due to the difference in coordinate systems between the models and FlightGear, these negations fixed the discrepancies during simulation so that the FDMs were adapted to depict the airplane within FlightGear properly. Lastly, when beginning the simulation, FlightGear automatically loads with the airplane facing East with an initial heading of 90 degrees. To compensate for this, the yaw angle in either FDM is always offset by 90 degrees, which faces the airplane in the correct direction to fly. So, if one were to use these FDMs with a different visualization system, these considerations may be necessary.

3.7.1 Simulation Loop

The simulation loop strategy chosen makes frames per second dependent on a constant game speed [38]. And to ensure numerical stability, the FPS chosen for the main simulation loop was 30, which equates to a fixed 33 milliseconds delta time from one frame to the next. With this time step, the simulation is stable. Although, other reasonable FPS values also work fine, such as 10. This fixed time step strategy was chosen for simplicity's sake. This strategy works because it is known that the integration of the equations of motion reliably takes less real processing time than 33 milliseconds, and so the simulation can maintain a smooth frame rate [20]. This strategy might normally be an issue for slow computers, but the FDMs are not very demanding on hardware. The downside is that for fast computers, some frames per second are left on the table due to the fixed time step [38]. There is a frame rate variable in the main function that may be modified to a realistic value as to not crash the simulation.

3.7.2 Resources

The Resources at work are the same as what is used by both FDMs. The Palmer-based simulator uses only the DeltaTime Resource. The Bourg-based simulator uses DeltaTime, MaxThrust, and DeltaThrust.

3.7.3 Components

Components contain the data variables to be accessed by a System. The three Components required for the flight simulator are KeyboardState, FGNetFDM, and DataFDM.

3.7.3.1 KeyboardState

The KeyoardState Component holds the boolean variables that correspond to the action that will take place in the simulation when a specified flight control key is pressed on the keyboard. This Component is used by the EquationsOfMotion System and the FlightControl System.

Because the Palmer-based FDM cannot modify yaw angle directly, the KeyboardState Component specific to that simulator does not contain and variables designating a keypress to yaw left or yaw right, whereas the Bourg-based simulator does.

3.7.3.2 FGNetFDM

The FGNetFDM Component uses the FlightGear defined FGNetFDM structure, which contains all of the data necessary to be sent to FlightGear in the form of a packet. This original FGNetFDM structure that FlightGear accepts is written in C++ and posted in FlightGear's official GitHub repository [4]. It was translated into Rust for this research. This Component is used by the MakePacket System and the SendPacket System.

3.7.3.3 DataFDM

The DataFDM Component holds all of the data describing an airplane. This is the same Component defined in the previous sections, 3.2, and 3.3, for each respective FDM being used by the EquationsOfMotion System. However, this Component is also now accessed by the MakePacket System, which is described in the next section discussing the Systems.

3.7.4 Systems

There are four Systems required to support interactive flight simulation: FlightControl, EquationsOfMotion, MakePacket, and SendPacket. Systems access certain Components that are needed for reading or writing, and the type of access must be specified. ReadStorage is only read access to the Component data, and WriteStorage is reading and also writing over the Component data.

The figure below displays the organizational breakdown of the Components and Systems used in the flight simulators. The storage type, WriteStorage and ReadAccess, is denoted by a W and a R on the Components required by the Systems, respectively.

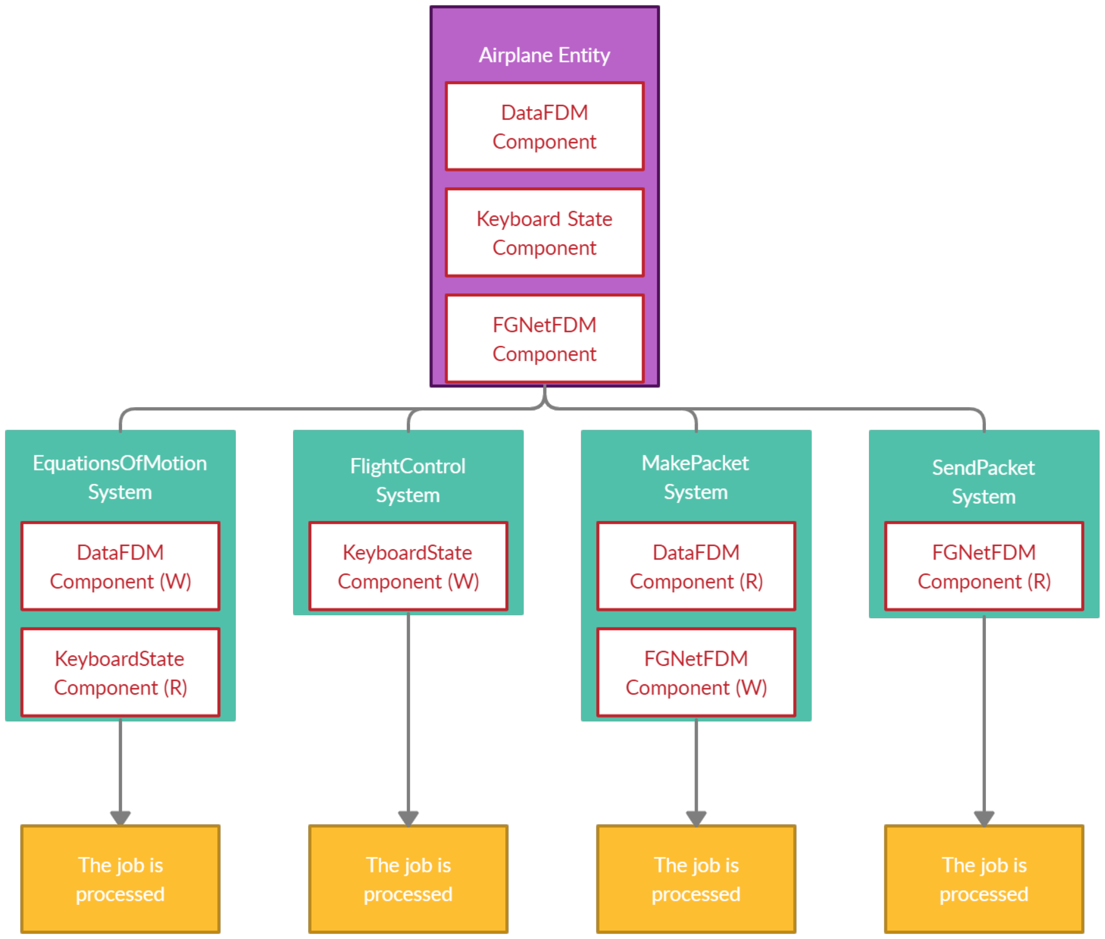

Figure 11: Flight Simulator Entity-Component-System Organization

For a single frame, FlightControl determines if any and what keys are being pressed. Then, EquationsOfMotion takes that keypress into account for the calculation of the resulting flight behavior. Next, with the equations of motion calculated, MakePacket updates the FGNetFDM Component with that successive data. Finally, SendPacket sends the constructed packet to FlightGear. Each System is dependent on the previous System executed to update data.

The flight simulators have three write-read conflicts between all four Systems based on the Component access type required by each System. These conflicts are detected by the dispatcher when automatically attempting to parallelize the Systems. Due to these conflicts, every System in each flight simulator needs to be executed sequentially, instead of concurrently. The conflicts occur because every System, except the first System in the order, needs to wait for another System to update some data before it can start working on that simulation frame. There are write-read conflicts between FlightControl and EquationsOfMotion, EquationsOfMotion and MakePacket, and SendPacket. Although the dispatcher knows these conflicting Systems cannot be run in parallel, the dispatcher does not assign a logical execution flow for the flight simulator. This conflict could lead to non-deterministic behavior where the execution order is inconsistent from simulation frame to frame, therefore leading to an inconsistent result [27]. So, the specification of the deterministic sequential execution order of the dispatcher in code is needed. The figure below represents the execution order per frame of both flight simulators.

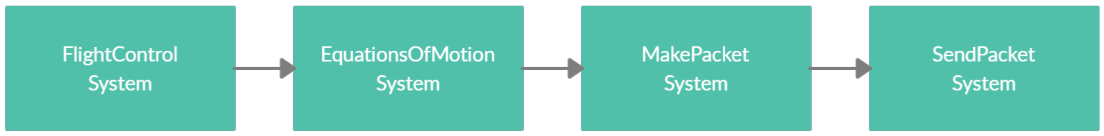

Figure 12: Flight Simulator System Execution Flow

The SPECS framework makes adding a System execution order by specifying System dependencies easy. When a System is added to the dispatcher, the dispatcher can be told to wait for another System, or Systems, to finish before another System may begin. In other scenarios where Systems do not have any conflicts, the execution order specification is not considered, and the dispatcher does its job automatically managing parallelization.

3.7.4.1 FlightControl

This System ultimately handles getting the keypresses from the keyboard and then changes the KeyboardState Component data. So, it has WriteStorage access to KeyboardState.

A Rust crate called device_query [39] is implemented to detect the keyboard keypresses, which can read the current state of the keys on the keyboard (i.e., the key is being pressed or not). In code, device_query gets a list of all the keys being pressed at the current instance, and then a check is made to see if any of the keys in that list match the keys that are tied with a flight control that affects the equations of motion in our FDM. Since more than one key can be pressed at once, all of the designated keys are checked, and when a key is being pressed, the KeyboardState data that corresponds to that keypress is toggled to true. Before any of the KeyboardState data is toggled to true, the data is defaulted to false to start fresh for that frame. Once the FlightControl System has done its job, the EquationsOfMotion System takes care of incorporating the keypresses into calculating the flight behavior.

The Palmer-based FDM does not designate a keypress to modify yaw, so for that respective flight simulator, there are no flight controls to explicitly yaw left or yaw right. The yaw angle is dependent on the yaw that takes place due to the bank action.

Each flight simulator's keyboard input flight controls are laid out in Appendix B: Flight Simulator Keyboard Controls. The two simulators are set up similarly regarding flight control keyboard input, but there are some differences to note.

1. In the Palmer-based simulator, the amount of a keyboard action is limited to a range of values (e.g., limited roll angle and throttle application).
2. In the Bourg-based simulator, two more keys are designated to yaw left and yaw right, whereas the Palmer-based simulator cannot directly yaw with a keypress.

It is important to note again that the flight control keypresses directly affect the values they represent in the Palmer-based simulator to explain difference #1. For example, rolling to the right in this simulator will directly change the bank angle value, then the equations of motion consider that bank angle. Whereas in the Bourg-based simulator, rolling right does not directly alter the bank angle. A roll to the right activates the ailerons to roll the airplane, the equations of motion consider that components activation, then a bank angle is determined from the new airplane orientation, and finally, that orientation is displayed in FlightGear. This difference is worth explaining because a component can be activated with no limit. Thrust is an exception; similar to all the input in the Palmer-based simulator, in the Bourg-based simulator, the thrust force is directly altered by a keypress. The thrust is incremented or decremented by a constant number, defaulted to 100, out of the total maximum thrust force capacity. Considering a component can be activated by endless keypresses, a barrel roll can be performed in the Bourg-based simulator. However, the Palmer-based simulator does not employ this freedom. In the Palmer-based simulator, the bank angle, attack angle, and flaps have a value range. Attack angle ranges from -16 degrees to 20 degrees. Bank angle ranges from -20 degrees to 20 degrees. Flaps are deployed at either 0 degrees, 20 degrees, or 40 degrees. The values on each range are incremented or decremented by 1, but the throttle application is incremented or decremented by 5% each keypress.

3.7.4.2 MakePacket

This System aids in graphics generation of the airplane within FlightGear by loading the FGNetFDM structure, defined by FlightGear [4], with data that the SendPacket System sends. The data loaded contains the state of the airplane, including the geographical position and orientation. This System accesses both the DataFDM Component and the FGNetFDM Component as ReadStorage.

In this System, the FGNetFDM Component is updated with the current calculations from the EquationsOfMotion System, which are stored in the DataFDM Component. When the new data variables are passed into the FGNetFDM structure, the data must be first converted into an array of bytes in native byte order, meaning the byte order native to the machine compiling the software. From here, the byte array is then read as a big-endian representation and turned into an integer value that FlightGear can interpret.

The data updated is the position of the airplane and also the orientation of that airplane. The FGNetFDM position data updated is latitude, longitude, and altitude. The orientation variables updated are phy, theta, and psi, representing the Euler angles roll, pitch, and yaw, respectively. The FlightGear version is also required to be passed on to the FGNetFDM structure.

3.7.4.3 SendPacket

Now that the FGNetFDM Component is updated and converted to big-endian byte order, this System begins by serializing the FGNetFDM structure into a packet of u8 bytes in the form of a vector. The Rust crates called bincode [40] and serde [41] are used to do this.

Next, this System serves to send the packet. First, a socket is opened. Next, the FlightGear socket is connected by a port to the socket opened. Lastly, the packet is sent off to FlightGear's socket. This System only needs ReadStorage access to the FGNetFDM Component.

3.7.4.4 EquationsOfMotion

This System is what computes the equations of motion for the flight simulators. This System explanation is almost the same as the System defined in the previous sections, 3.2 and 3.3, that outline the respective FDMs. However, there are two important differences between the FDMs defined by their respective authors versus this System that is configured to function as part of an interactive flight simulator interfacing FlightGear:

1. The KeyboardState Component data is considered as to whether or not a key has been pressed before the equations of motions are computed to account for that flight behavior.

2. The coordinate position displacement calculated is represented as East, North, Up (ENU) Cartesian coordinates, and are used to calculate a new geodetic coordinate location consisting of degrees of latitude, degrees of longitude, and meters of altitude.

With that, this System now requires ReadStorage access to KeyboardState and requires WriteStorage access to DataFDM.

The System begins resetting all of the airplane component states so that they are not activated to affect the equations of motion in the Bourg-based model. The Palmer-based model does not do this because a state stays constant until a keypress changes it. For both simulators, the KeyboardStates are then checked to see if any states are toggled true. If a state is true, the associated component is activated for this frame in the Bourg-based simulator, or the state is directly modified in the Palmer-based simulator. However, the thrust states are not associated with a component and are modified accordingly, like how the Palmer-based model handles it. With a component activated like the ailerons, caused by the roll right keypress state, for example, the FDM takes that into account for the calculation of that frame. That is the part that makes the FDM interactive.

Following keyboard input handling, the System continues as normal to calculate the equations of motion the same as described in sections 3.2 and 3.3 that describe each respective FDM. However, when the new position is being calculated, a conversion must take place. The displacement in meters calculated by each FDM is represented as ENU Cartesian coordinates, but FlightGear requires geodetic coordinates to represent the airplane's position on the Earth within FlightGear. A crate called coord_transform [42] is used to do the math to calculate a new geodetic coordinate position based on a change in displacement in meters calculated by the FDM.

3.8 Summary

The methodology first explains how the FDMs were re-implemented into ECS and how the FDMs were verified to work as expected. The FDMs were translated to Rust from their native programming language, C and C++, for the Palmer model and Bourg model, respectively. Then the working Rust-based FDMs were re-implemented into the ECS architecture using SPECS.

The re-implemented ECS FDMs were then tested against their original code. Rust integration tests were used to handle testing. A series of flight simulation scenarios were performed on the original FDMs. The flight data generated on that scenario's last frame was used in the Rust integration test that performs that same scenario on the ECS FDMs. At the end of the test, the flight data generated by the ECS FDMs is compared to the original results to a floating-point tolerance. The integration test outputs pass or fail depending if the results match to that tested tolerance.

The benchmark performance comparison of the two FDMs was set up using the Rust criterion crate, which uses statistical analysis to derive performance. The Bourg and Palmer model were both set up as functions in a code file inside the project's *benches* folder. The benches were input with a single number of frames to execute for each of the multiple frame rate inputs. The benches demonstrated consisted of 100 frames for each 10, 30, 60, 100 FPS bench. The two functions were benched as a group and compared.

The build of the flight simulators is finally explained. The Bourg and Palmer FDMs explained in sections 3.2 and 3.3 are extended into flight simulators by adding Systems and Components to handle keyboard input and making/sending UDP packets to FlightGear for visualization.

IV. Analysis and Results

This chapter explains the results obtained by the tests performed in the methodology. First discussed are the integration tests performed to determine equivalency of our Palmer-based Flight Dynamics Model (FDM) designed using the Entity-Component-System (ECS) written in Rust compared to the original imperative C-based FDM. Next, the integration tests comparing the Bourg-based Rust/ECS FDM to the original FDM written using Object-Oriented Programming (OOP) and C++ are discussed. Next, the equivalency of the re-implemented custom vector, matrix, and quaternion modules used in the Bourg-based FDM compared to the original C++ code is verified. Additionally, the benchmarking results between the two FDMs are presented. Lastly, the completed flight simulators using the FDMs are examined.

5.1 Palmer-based FDM Equivalency Verification

The series of Rust integration tests investigating our Rust/ECS FDM versus the original C-based FDM under some specific scenarios laid out in Chapter III demonstrated accuracy in our FDM given that the flight data passed each test with a floating-point epsilon of 0.000001. Each test focused on determining the accuracy of a specific flight control manipulated by the FDM. One test, however, applies all of the flight controls during the simulation period. Overall, these results found that the FDM is working as expected within a floating-point error.

5.2 Bourg-based FDM Equivalency Verification

The series of Rust integration tests investigating our Rust/ECS FDM versus the original C++/OOP FDM under some specific scenarios demonstrated accuracy in our FDM given that the flight data matched to a floating-point tolerance of 0.01 in every test. Although this is a larger epsilon than the Palmer-based tests, this can be attributed to the FDM's use of 32-bit floating-point numbers, whereas the Palmer-based model uses more precise 64-bit floating-point numbers throughout the software. Each test focused on determining the accuracy of a specific flight control manipulated by the FDM. One test, however, applies all of the flight controls during the simulation period. Regardless, with small tolerance of 0.01, all the Rust integration tests pass.

5.2.1 Custom Type Modules

To add an extra layer of equivalency verification to our Bourg-based re-implemented FDM, Rust unit tests were added to check the operations defined by the custom vector, matrix, and quaternion modules versus the C++ classes they are based on. All of the unit tests passed without the need for a tolerance check.

5.3 FDM Benchmarking

The criterion benchmark results determined that the Palmer model is faster on average than the Bourg model. Figure 13 below is the comparison graph generated by criterion that shows the performance difference. The graph shows that the Palmer model is faster, about twice as fast, than the Bourg model for every frame rate input. It is not important to get caught up on the numerical results as the results are dependent on the machine running the benchmarks. For example, a machine plugged into power will result in a lower average time for either model. However, the trend regardless of machine should demonstrate that the Palmer model is faster by some ratio, which also depends on the machine.

Line Chart

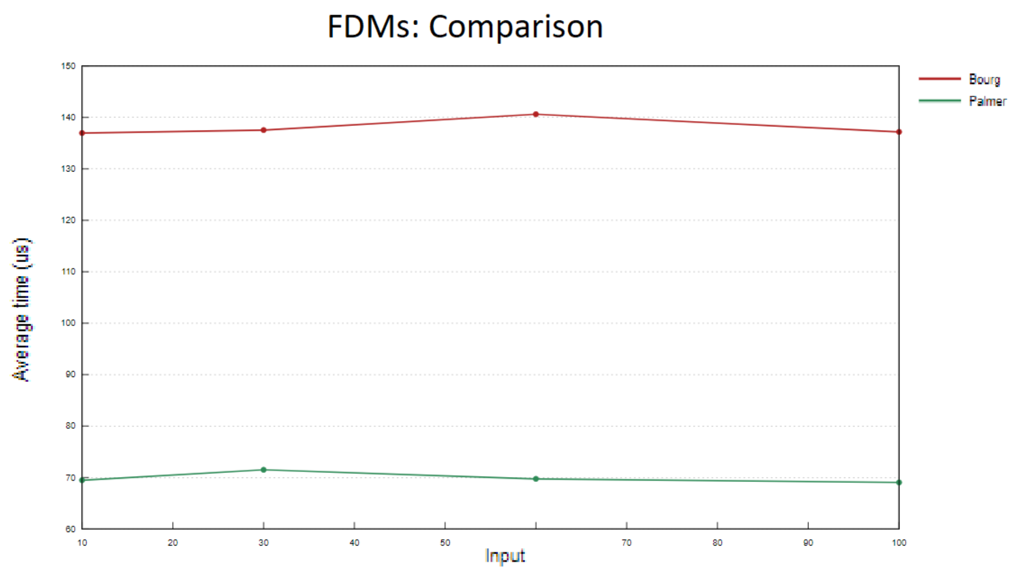

Figure 13: Performance Benchmark of Bourg and Palmer Models

This benchmark outcome is sensible considering that the Palmer model is more simplistic than the Bourg model. The Bourg model contains more detail in terms of how each airplane component contains mass/surface properties that contribute to the calculation of lift and drag forces. Furthermore, the lift and drag coefficients are computed in relation to the airplane's more detailed lifting surfaces while also taking into account a rudder is more complex than Palmer's model. So, it is reasonable that the Palmer model is quicker.

5.4 Flight Simulators

The System and Component built to calculate the equations of motion for both the Palmer-based and Bourg-based FDM re-implementations discussed in Chapter III are ultimately able to come together with three additional Systems and two Components to create a complete flight simulator. The additional Systems and Components support the functions that the raw FDMs do not handle in terms of tracking keyboard input and stimulating the graphics of FlightGear [4] via UDP packet. These two flight simulators employ the realistic physics generated by the equations of motion from their respective FDMs. The decoupling nature of the ECS architecture made the process of tying together all the Systems and Components seamless.

Running the flight simulators is simple. Appendix A: Configuring FlightGear as a Visual System describes how to execute the FlightGear application with specific command-line arguments. These command-line arguments set up FlightGear to accept external software to interface it and receive flight data via UDP packet. Once FlightGear has loaded the environment, either flight simulator example created in this thesis can be executed, and the airplane can be flown using the keypresses listed in Appendix B.

Overall, both simulators' flight experience at 30 Frames Per Second (FPS) is smooth and replicates the realistic flight dynamics outlined by Palmer and Bourg to a floating-point margin of error. Figure 14 below shows what a user sees when executing the software created in this research within FlightGear.

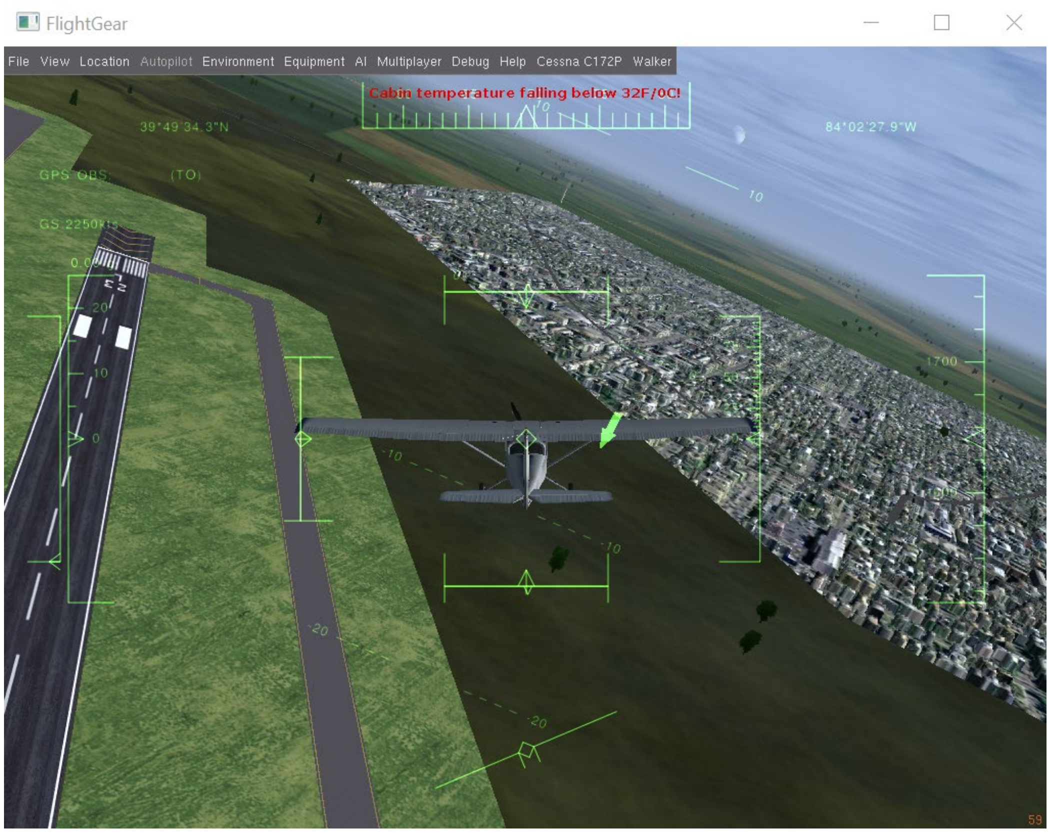
FlightGear
File View Location Autopilot Environment Equipment AI Multiplayer Debug Help Cessna C172P Walker
Cabin temperature falling below 32F/0C!
39°49'34.3"N
84°02'27.9"W
GPS OBS
(TO)

Figure 14: FlightGear Display During Simulation

5.5 Summary

This research's goal of building ECS/Rust-based interactive flight simulators using published FDMs was completed. The FDM created that is based on Palmer's [35] original FDM was determined to be accurate to a floating-point tolerance of 0.000001. The FDM based on Bourg's [3] original FDM was determined to be accurate to a floating-point tolerance of 0.01. The results were gained by testing the FDMs created against their original code in Rust integration tests. A series of flight simulation scenarios were performed on the original FDMs. The flight data generated on that scenario's last frame was used in the Rust integration test that performs that same scenario on the ECS

FDMs. For each test, the flight data generated by the Palmer-based ECS FDM was accurate to the original by a floating-point tolerance of 0.000001, and the Bourg-based ECS FDM was accurate to the original by a floating-point tolerance of 0.01. This difference in tolerance can be attributed to the fact that the Palmer model uses 64-bit floating-point values in code, whereas the Bourg-model uses less accurate 32-bit floating-point values.

The benchmarking completed between the two FDMs resulted in the Palmer model being faster on average than the Bourg model. This is due to the level of detail that the Bourg models the airplane's lifting surfaces, like the wings, and how each airplane component affects the lift and drag forces.

The creation of flight simulators that use the ECS FDMs was a success. Additional ECS Systems and Components were added to the FDMs to support keyboard input and UDP packet updating and sending for graphic visualization. Overall, using FlightGear for visualization, a user can execute the software built in this research, using either the Bourg or Palmer examples, and can fly a simulated airplane with realistic behavior anywhere in the world. Reference Appendix A to configure FlightGear for use upon executing the application, and reference Appendix B for the keyboard controls.

V. Conclusion

This chapter summarizes the research and results obtained throughout the thesis. The results found during testing and analysis of the two Flight Dynamics Models (FDM) and their subsequent flight simulators are reiterated. Next, this research's contribution impact on the field of simulation and modeling is described, and possible future research extending this thesis concerning the Entity-Component-System architecture (ECS) is explained.

6.1 Research Conclusions

This thesis research concludes that the use of the Data-Oriented Design (DOD) paradigm and specifically the DOD-based ECS architecture in flight simulation is feasible and a good decision when considering its performance and improvements in ease of coding. Furthermore, the memory-safety guarantee made by the Rust programming language, while retaining systems-level language performance, was useful in coding flight simulators. Everything considered, through the re-implementation of two FDMs, and thus two flight simulators, into Systems and Components using the Specs Parallel ECS (SPECS) [1] framework, the viability of the ECS architecture was confirmed.

The first re-implemented FDM was based on Grant Palmer's works in his published textbook titled *Physics for Game Programmers* [2]. The FDM described in his book was transformed into a Rust-based, ECS-based version. This re-implementation was deemed successful through a series of Rust integration tests comparing the flight data calculated between our re-implementation versus Palmer's original imperative C-based version. The

Rust integration tests ran the Rust/ECS FDM through a simulation for 60 seconds and took the last frames flight data, and compared it to the flight data generated under the same parameters by the original code. The Rust integration tests all passed with a floating-point tolerance of 0.000001, verifying that the FDM works as expected.

The second re-implemented FDM is based on David Bourg's works in his published textbook *Physics for Game Developers* [3]. This FDM was also re-implemented using the ECS architecture and the Rust programming language. A series of successive Rust integration tests comparing the flight data calculated by our re-implementation versus Bourg’s C++ and Object-Oriented Programming-based version was also conducted. Like the previous tests, the FDMs ran through a flight path performing some maneuvers and compared the flight data from the last frame of the original to the flight data calculated by the integration test. The integration tests were accurate to a 0.01 numerical floating-point tolerance, verifying that the FDM works as expected.

Furthermore, concerning Bourg’s flight model, the custom vector, matrix, and quaternion modules that were re-implemented in Rust based on Bourg’s custom C++ classes were tested. Rust unit tests were set up to check all of the operations that are performed in this FDM. Similar to the integration tests explained above, the operations were performed in C++ with some input to gather the baseline data to be compared in the unit tests. The original data was copied to each respective unit test to be compared to the output of our Rust implementation. All the tests passed with no need for tolerance, verifying that these Rust modules function as expected.

Using the Rust crate called criterion, a benchmark comparison determined that the Palmer model is faster on average than the Bourg model. The benches were input with a single number of frames to execute for each of the multiple frame rate inputs. The benches demonstrated consisted of 100 frames for each 10, 30, 60, 100 FPS bench. The benchmark results completed between the two FDMs are due to the detail that the Bourg models the airplane's lifting surfaces, like the wings, and how multiple airplane component affects the lift and drag forces.

With the two FDMs built using the ECS architecture, the addition of Systems and Components that support keyboard input and graphic visualization extends each FDM to a full-scale flight simulator. Because of the ECS design's decoupling nature, this extension of each FDM into flight simulators is effortless. Although not measured in this thesis but evident in [9], the performance benefits of ECS are indeed at work by increasing the software's cache usage efficiency.

Overall, the flight simulation experience within the FlightGear [4] visualization system for both simulators is stable, smooth, and looks realistic. Additionally, the simulators, based on the Rust integration testing results, replicate the realistic flight dynamics that the original FDMs employ. Next, it is known that by using SPECS, the simulators are managing memory efficiently and parallelizing the Systems when possible. Lastly, by programming with the Rust language, it is guaranteed that memory issues such as data races, dangling references, and buffer overflows are not in the simulation software.

6.2 Significance of Research

The contribution of this thesis lies in the field of flight modeling and simulations. The successful build of a Rust-based, ECS-designed, real-time FDM demonstrates an alternate, modern, and powerful design strategy over current FDMs. Not only is this strategy available for use, but it is useful to increase performance through efficient memory-management and parallelization. This strategy also increases code maintainability due to the ECS architecture's decoupling nature; this is a valuable feature that could save time and effort when maintaining the codebase later in its lifespan.

6.3 Future Research

Future possible research lies in the improvement of both re-implemented FDMs in their ECS design. The Component containing all of the data representing the rigid body airplane, called DataFDM, is quite large. It is larger than ideal for what an ECS-based software would want to use. An important aspect of ECS is that the Components are split into chunks that are as small as possible to improve efficiency by bringing only what is necessary into the cache. Though the way the EquationsOfMotion System is coded, it does need the entire DataFDM Component. However, an improvement in the ECS design would be that of determining the data dependencies within the EquationsOfMotion System. If someone were to determine what parts of the System depended on what data and split the EquationsOfMotion System and DataFDM Component into smaller pieces to run concurrently – that would be a substantial improvement in the ECS design. For example, maybe three smaller Systems total could be created where the first two Systems can run in parallel for the first half of the program, and then the next System finishes the job. Hypothetically, this would increase total performance by 25%.

Another design improvement would be implementing a Rust crate that can handle the vector, matrix, and quaternion types and their operations. As it stands, this thesis' code re-implements the custom vector, matrix, and quaternion classes originally defined by David Bourg in his textbook [3]. Although the code is functional, the redefinition of these custom classes was extraneous given that there are certainly importable crates that handle these types of structures and their operations.

Appendix A: Configuring FlightGear as a Visual System

The command line to execute the FlightGear application for simulation use in this research is:

```
fgfs.exe --aircraft=ufo --disable-panel --disable-sound --
enable-hud --disable-random-objects --fdm=null --
timeofday=noon --native-fdm=socket,in,30,,5500,udp
```

Note: Replacing `--aircraft=ufo` with `--aircraft=c172p` will display a Cessna 172P Skyhawk.

Command line breakdown:

`fgfs.exe` – runs the executable

`--aircraft=ufo` – select an aircraft name

`--disable-panel` – disable instrument panel

`--disable-sound` – disable sounds

`--enable-hud` – enable heads up display

`--disable-random-objects` – disable random scenery objects

`--fdm=null` – turn off the built-in Flight Dynamics Model

`--timeofday=noon` – time of day

`--native-fdm=socket,in,30,,5500,udp` – sets up FlightGear to receive data from an external source by opening a socket, which accepts in packets at a rate of 30 hz on port 5500 [43].

Appendix B: Flight Simulator Keyboard Controls

Palmer-based Flight Simulator	
Key	**Action**
E	Throttle Up
D	Throttle Down
Down Arrow	Angle of Attack Up
Up Arrow	Angle of Attack Down
Left Arrow	Bank Left
Right Arrow	Bank Right
K	Flap Deflection at 20 Degrees (1 press) Flap Deflection at 40 Degrees (2 presses)
L	Reset Flaps

Table 12: Palmer-based Flight Simulator Keyboard Controls

Bourg-based Flight Simulator	
Key	**Action**
E	Thrust Up
D	Thrust Down
Down Arrow	Pitch Up
Up Arrow	Pitch Down
Left Arrow	Roll Left
Right Arrow	Roll Right
N	Yaw Left
M	Yaw Right
K	Flap Deflection
L	Reset Flaps

Table 13: Bourg-based Flight Simulator Keyboard Controls

Bibliography

[1] slide-rs hackers, “Specs.” https://crates.io/crates/specs.

[2] G. Palmer, *Physics for game programmers*. Apress, 2005.

[3] D. M. Bourg and B. Bywalec, *Physics for Game Developers, Second Edition*. O’Reilly Media, 2013.

[4] “FlightGear Flight Simulator.” https://www.flightgear.org/.

[5] R. R. Hill and J. O. Miller, “A History Of United States Military Simulation,” *2017 Winter Simul. Conf.*, 2017.

[6] J. A. Vagedes, D. D. Hodson, S. L. Nykl, and J. R. Millar, “ECS Architecture for Modern Military Simulators,” 2019.

[7] “Create a world with more play,” 2020. https://unity.com/solutions/game.

[8] “Unity helps bring the masses together in Mediatonic’s absurd multiplayer,” 2020. https://unity.com/case-study/mediatonic-fall-guys.

[9] J. A. Vagedes, “A Study of Execution Performance for Rust-Based Object vs Data Oriented Architectures,” Air Force Institute of Technology, 2020.

[10] “Flight Dynamics Model,” *FlightGear Wiki*. http://wiki.flightgear.org/Flight_Dynamics_Model.

[11] “Degrees of freedom (mechanics),” *Wikipedia*, 2020. https://en.wikipedia.org/wiki/Degrees_of_freedom_(mechanics).

[12] M. T.S., *Rust In Action*, Manning Ea. Manning Publications, 2019.

[13] W. Durham, *Aircraft Flight Dynamics and Control*. Wiley, 2013.

[14] “Flight dynamics (fixed-wing aircraft),” *Wikipedia*, 2020.

https://en.wikipedia.org/wiki/Flight_dynamics_(fixed-wing_aircraft).

[15] "Airfoil," *Wikipedia*, 2020. https://en.wikipedia.org/wiki/Airfoil.

[16] "Flight control surfaces," *Wikipedia*, 2020. https://en.wikipedia.org/wiki/Flight_control_surfaces.

[17] "Local tangent plane coordinates," *Wikipedia*. https://en.wikipedia.org/wiki/Local_tangent_plane_coordinates.

[18] "Moment of inertia," *Wikipedia*, 2020. https://en.wikipedia.org/wiki/Moment_of_inertia.

[19] G. Fiedler, "Integration Basics," 2014. https://gafferongames.com/post/integration_basics/.

[20] G. Fiedler, "Fix Your Timestep!," 2014. https://gafferongames.com/post/fix_your_timestep/.

[21] R. Nystrom, "Game Programming Patterns," 2014. https://gameprogrammingpatterns.com/game-loop.html.

[22] A. Rhemann, "A Handbook of Flight Simulation Fidelity Requirements for Human Factors Research," *Natl. Tech. Inf. Serv.*, 1995.

[23] G. Dussart, V. Portapas, A. Pontillo, and M. Lone, "Flight Dynamic Modelling and Simulation of Large Flexible Aircraft," *Flight Phys. - Model. Tech. Technol.*, 2018, doi: 10.5772/intechopen.71050.

[24] M. M. Duquette, "The development and application of SimpleFlight, a variable-fidelity flight dynamics model," *Collect. Tech. Pap. - 2007 AIAA Model. Simul. Technol. Conf.*, vol. 1, no. August, pp. 133–149, 2007, doi: 10.2514/6.2007-6372.

[25] Unity, "Overview - Intro To The Entity Component System And C# Job System

1/5," *YouTube*, 2018. https://www.youtube.com/watch?v=WLfhUKp2gag&t=186s&ab_channel=Unity.

[26] Unity, "Overview - Intro To The Entity Component System And C# Job System 2/5," *YouTube*, 2018. https://www.youtube.com/watch?v=z9WE3fwre-k&t=1s&ab_channel=Unity.

[27] T. Schaller, *The Specs Book*, 1st Editio. Crates.io, 2019.

[28] Steve Klabnik and C. Nichols, *The Rust Programming Language*. 2015.

[29] S. Klabnik and C. Nichols, *The Rustonomicon*. 2019.

[30] R. Apodaca, "Rust Ownership By Example," 2020. https://depth-first.com/articles/2020/01/27/rust-ownership-by-example/.

[31] "The Rust's Community Crate Registry." https://crates.io/.

[32] "Command line options," *FlightGear Wiki*. http://wiki.flightgear.org/Command_line_options.

[33] B. Beej and J. Hall, "Using Internet Sockets," 2019.

[34] "Rust By Example." https://doc.rust-lang.org/rust-by-example/.

[35] G. Palmer, "Source code for 'Physics for Game Programmers,'" *Apress*. https://github.com/Apress/physics-for-game-programmers.

[36] M. Dilger, "float-cmp." https://crates.io/crates/float-cmp.

[37] B. Hesiler, "criterion." https://crates.io/crates/criterion.

[38] K. Witters, "deWiTTERS Game Loop," 2009. https://dewitters.com/dewitters-gameloop/.

[39] Ostrosco, "device_query." https://crates.io/crates/device_query.

[40] D. Tolnay, "bincode." https://crates.io/crates/bincode.

[41] D. Tolnay, “serde.” https://crates.io/crates/serde.

[42] D. Kramer, “coord_transforms.” https://crates.io/crates/coord_transforms.

[43] “Property Tree/Sockets,” *FlightGear Wiki*.

http://wiki.flightgear.org/Property_Tree/Sockets.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 25-03-2021	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From - To)* Sep 2019 - Mar 2021

4. TITLE AND SUBTITLE Two Published Flight Dynamics Models Rewritten in Rust and Structured as an ECS	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Willis, Chad A, 2d Lt	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way Wright-Patterson AFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT-ENG-MS-21-M-095
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/RQQD 2180 8th Bldg 145 R034 Wright-Patterson AFB, OH 45433 (312) 672-6556	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This thesis explores using the Entity-Component System (ECS) architecture to implement a Flight Dynamics Model (FDM) by re-implementing two published versions in the Rust programming language using the Specs Parallel ECS (SPECS) for military simulation advancement. One FDM is based on Grant Palmer's published textbook titled Physics for Game Programmers, and another is based on David Bourg's textbook titled Physics for Game Developers. The ECS architecture is based on the Data-Oriented Design (DOD) paradigm. The specific ECS selected for this research is the Rust-based SPECS framework. One goal of this research is to define the Components and the Systems of an ECS for the two FDMs and use them to build a flight simulator by creating additional Systems and Components to read keyboard inputs and send User Datagram Protocol (UDP) packets to the FlightGear application for visualization. The ECS Systems used to calculate the equations of motion both for FDM's functionality were tested against the original code and were found to work as expected within a floating-point tolerance.

15. SUBJECT TERMS

Real-Time Simulations, Flight Dynamics Models, FlightGear, DOD, ECS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. Douglas D. Hodson, AFIT/ENG
U	U	U	UU	104	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-3636 x4719 Douglas.Hodson@afit.edu

Reset