DESIGNING AND BUILDING A RADAR SIMULATION USING THE ENTITY
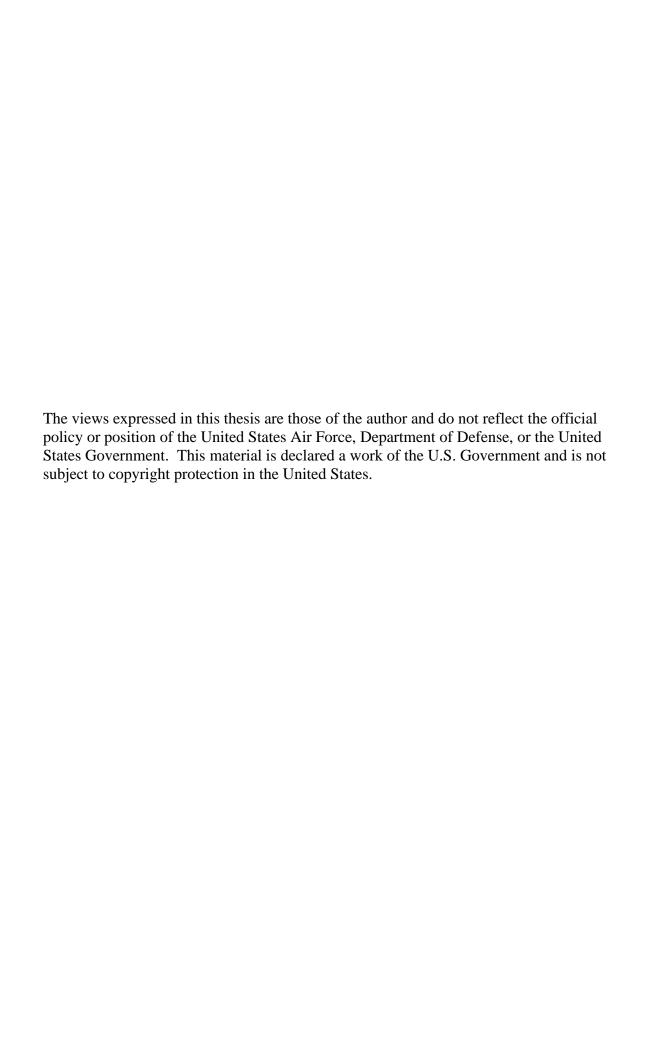COMPONENT SYSTEM

THESIS

Brennen T. Garland, 2d Lt, USAF

AFIT-ENG-MS-21-M-038

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-21-M-038

DESIGNING AND BUILDING A RADAR SIMULATION USING THE ENTITY
COMPONENT SYSTEM

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

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

Brennen T. Garland,

2d Lt, USAF

March 2021

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

AFIT-ENG-MS-21-M-038

DESIGNING AND BUILDING A RADAR SIMULATION USING THE ENTITY
COMPONENT SYSTEM

Brennen T. Garland,

2d Lt, USAF

Committee Membership:

Dr. Douglas D. Hodson, Ph. D.
Chair

Dr. Scott L. Nykl, Ph. D.
Member

Major Richard Dill, Ph. D.
Member

AFIT-ENG-MS-21-M-038

Abstract

This research explores the implementation of a "medium fidelity" radar simulation using the Entity-Component-System (ECS) architecture. The radar implemented mimics the fundamental characteristics of entities in the open-source Mixed Reality Simulation Platform (MIXR) project, supporting real-time interaction. Previous research has shown the potential benefits of using an ECS-based architecture to support improved execution performance relative to Object-Oriented Programming (OOP) approaches, thus improved real-time interaction requirements [1], [2]. This research implements a well-documented radar model that supports the development of soft real-time human-based interaction simulations [3]. The radar system modeled in this research mimics the "out-of-the-box" fidelity defined in the OOP-based MIXR architecture [4]. This research creates components (i.e., data) to represent antenna patterns, target cross-sections, and emissions. The systems (i.e., computer logic or behavior) create and compute so-called "emission" data processed in phases that represent initial transmission and reception. Validation of the model was accomplished by creating test scenarios and comparing outputs with calculated values.

Table of Contents

List of Figures

List of Tables

DESIGNING AND BUILDING A RADAR SIMULATION USING THE ENTITY COMPONENT SYSTEM

I. Introduction

1.1 Background

Object-Oriented Programming (OOP) has been the central programming paradigm used and taught in software development. Data-Oriented Programming (DOP), a less popular, well-established programming paradigm, reduces computer code complexity and improves execution performance. DOP has gained popularity through the use of Entity-Component-System (ECS), an architecture based on DOP principles, in-game development. The ECS architecture is famous in the gaming world, as it has proven itself to be influential in the development of high quality, efficient game engines [5]. Many believe that this architecture could be used in other fields besides games to improve software performance and software aging [6].

High fidelity radar simulations are computationally expensive and must become more efficient to improve usability [7]. The expense is not a problem unique to radar simulations, but it shows a need for efficient simulation design across many domains. Studies have shown that developing simulations with the ECS architecture could improve computational efficiency [2], [8]. While these studies are helpful and necessary, a full simulation is different from an experiment because problems can occur in a real simulation unseen in the experiments. The problems would most likely be large design patterns or behaviors unique to simulations seen in the composition of concepts rather than these concepts isolated in an experiment.

1.2 Problem Statement

ECS is an organizational design pattern or architecture used in games to improve games' execution and maintainability. Through the lens of software design, games and simulation have much in common, but unlike gaming, the simulation world has not seen a practical ECS architecture implementation [2]. Studies show the proposed benefit of using the ECS architecture in the simulation world instead of OOP, but no practical military simulation exists to legitimize the architecture's use [2], [8]. Experimental research reveals the use in a controlled environment, but an implementation can show how the architecture works under real conditions, giving it more legitamcy to be implemented in the real word [9]. This research investigates the use of this pattern to model a radar system that supports military simulations.

1.3 Research Questions

This research hopes to answer the following questions regarding the use of the ECS architecture in a simulation:

- Can a radar simulation be implemented using ECS at the same fidelity as the MIXR package?
- Can this model be implemented with a pure ECS architecture?
- How can cross-system communication be avoided in the ECS architecture to promote strong decoupling?
- Does tension exist between a pure ECS design and the granularity of systems and components?

This research answers these questions to understand ECS's use within a simulation system. The questions should show the strengths and weaknesses of ECS with simulations, provide a reputable source for more knowledge on the subject, and provide a way to increase software effectiveness in the military and other domains.

1.4 Research Goals

This research designs and builds a radar simulation organized using the ECS pattern. ECS is a fundamentally different data-oriented computer science pattern from Object-Oriented Programming (OOP) approaches to software design. It was defined and is used in the gaming world to construct games that consist of hundreds, if not thousands, of interactive entities.

1.6 Contributions

This thesis contributes to the fields of:

- **Modeling & Simulation**: ECS is a commonly used architecture in games that has scarcely been implemented in simulations. There is active research to move ECS into simulations, but the literature is limited [10], [11]. This research directly impacts the modeling and simulation fields for military simulations.
- **Software Design**: This research evaluates the advantages and disadvantages of using an ECS in a real, interactive simulation. The application of this architecture clarifies the areas that it performs well.

1.5 Hypothesis

The expected result is a working simulation that executes faster (than a comparable OOP design) and easily integrates with other represented aspects typically modeled in a military domain (e.g., a flight dynamics model to control movement). The

expected experimental results are better execution performance (due to data organization), especially as the number of represented entities grows.

1.7 Thesis Overview

This thesis divides into five chapters. Chapter II provides background information about the programming concepts and tools to develop the simulation and relevant background on the radar system. Chapter III covers the characteristics and design of the simulated model. It considers the MIXR program and its characteristics and the applications of many concepts introduced in Chapter II. Chapter IV discusses the simulation structure, implementation of MIXRs characteristics, consideration of design decisions, and subjective analysis of the ECS architecture in simulations. Chapter V is a summary of the research and details future applications.

II. Background

2.1 Overview

This chapter describes the different architectures applied, the programming language used, and a radar simulation background.

2.2 Radar

This section covers a high-level overview of radar. The purpose is to cover the aspects of radar that are necessary to understand this research. Radar is an acronym for the term "radio detection and ranging" and is described by in the following quote:

> "A radar is an electrical system that transmits radio-frequency (RF) electromagnetic (EM) waves toward a region of interest and receives and detects these EM waves when reflected from objects in that region" [12].

This quote identifies two areas of interest that the background research covers: the radar itself and the environment in which it interacts. This research is not concerned with modeling every aspect of a radar system in great detail; only a few components of a radar system are modeled, covered later in this section.

The environment is the system outside of the radar where electromagnetic (EM) waves and other entities exist. Most of the simulation is responsible for simulating the behaviors and interactions of these entities and waves.

Two subsections follow: an overview of how EM waves carry information and how the radar interacts and uses that information.
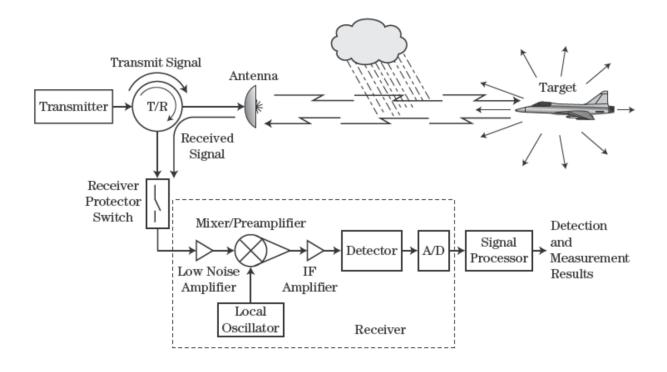
Figure 1. The radar system interacting with the environment [12].

2.2.1 Environment

The radar system interacting with the environment in Figure 1 provides a visualization of a basic radar. The antenna emits an EM wave that reflects off the target back to the antenna. The rain cloud is an essential feature because it introduces the idea that its environment affects the signals propagating or passing through it. The environment contains many random EM waves (e.g., *noise*). *Principle of Modern Radar* notes that any object "with a temperature above absolute zero will be radiating EM waves at, collectively, almost all frequencies" [12]. In addition to noise, the EM wave reflects off many objects, not just the target, producing other signals that the antenna picks up. These various reflections are known as *clutter* and might be an essential aspect of the

6

environment. A model must also consider the EM wave's interaction with the environment. This paper divides this interaction into two categories: EM wave propagation in terms of transmission and reflection. Both are covered, but understanding EM waves are necessary to model the interactions.

2.1.1.1 Electromagnetic Radiation

An electromagnetic wave is a self-propagating wave made up of an electric field and magnetic field perpendicular to one another [12]. Each field creates the other with a little less energy until the entire wave dissipates. The self-propagation allows it to move through space at the speed of light, c. Many different properties can describe an EM wave, but this research focuses on wavelength, λ; frequency, f; and intensity, Q. Intensity can be modeled by Equation 2.

$$Q = \frac{P_t}{4\pi R^2} \qquad\qquad \textit{Eq. 1}$$

P_t is the transmitted power, and R is the distance from the antenna. An EM wave can be modeled in a simulation by the above variables.

An EM wave is unique to the model because of its speed. An EM wave moves at a fixed rate of $3.0 \times 10^8 \frac{m}{s}$, the universal "speed limit," and is the only entity that travels at these speeds. Modeling each wave could be possible but modeling them individually at a meter resolution would require a fast simulation due to the EM waves' speed. Every frame would need to execute within three nanoseconds at this speed, which is fast even for a high-end, business computer. Add in that many EM waves exist within an environment, and the simulation can soon slow down by trying to model them individually. Instead, the EM waves can be abstracted and modeled as groups to allow for

7

a robust simulation that can model more than just EM waves. Abstracting EM waves into groups also requires an abstraction of how they interact with the environment as a group instead of individually.

2.2.1.2 Propagation

This section specifically discusses the propagation of an EM wave through a medium, not including clutter. EM wave propagation can become detailed, but this section serves as a basic overview of topics necessary to understand the simulation's functionality. The medium that an EM wave travels through has different mechanisms, some of which can exist simultaneously, affecting the EM waves travel. Examples of these mechanisms are atmospheric absorption, atmospheric refraction, or surface multipath. [12]. Each of these mechanisms contributes to the EM wave model. Atmospheric absorption is an example that shows these factors. An EM wave is attenuated, or loses amplitude, through an atmosphere. The attenuation is dependent on two factors: absorption and scatter [12]. Absorption occurs when objects within the atmosphere absorb energy, in the form of heat, from the EM wave [12]. Scatter occurs when a particle reflects the EM wave away from the receiver [12]. The behavior of an EM wave changes depending on either the presence or the precision of these factors. It could also add in different types of conditions to the same scenario. These would change the EM wave's behavior and, ultimately, the signal the radar would receive.

2.2.1.3 Reflection

Reflection is a complex phenomenon with many factors. This section only focuses on the result of the EM wave's interaction with a target. This research assumes that a target is an object of interest. Reflections from features in the environment are considered

clutter. At first glance, it may seem sensible to add in the physics for the target's response, but adding these can be complex and unnecessary [13]. The goal is not a simulation to find the radar-cross-section (RCS) of a target but to simulate one already found.

Reflection can be represented through many different levels of detail, which makes the concept difficult to understand. From a general perspective, a wave is scattered in all directions, mimicking an asymmetrical gain pattern. Representing this scatter in all directions is complicated and depends on the fidelity required of the model. Looking at a reflection as a function of the incident angle is much easier and can still represent an object's real RCS [13].

2.2.2 Radar Components

This section covers the physical components of a radar and how the radar interacts with EM waves.

2.2.2.1 Radar Architecture

A radar is made up of a transmitter, receiver, antenna, and signal processor. Figure 1 shows the components of the radar in a simple format [12]. While Figure 1 contains more components than described, this research abstracts the mixer component because its unessecary to represent the desired behaviour. Radars can be configured in many ways, but this paper focuses on a monostatic radar (e.g., a radar with the transmitter and receiver connected to the same antenna). These types of radars traditionally use pulsed radar, a radar that transmits EM waves in pulses. The length of a pulse is defined by the *pulse width*, τ, and is typically 0.1 to 10 microseconds [12]. Understanding the speed at which the pulses are transmitted, in conjunction with the previous discussion on

EM waves, one can see how the model can be slowed down by many EM waves in a short amount of time and the need for abstraction.

2.2.2.2   Signal Reception

A typical radar transmits a pulse for the pulse width duration, and then the receiver listens for any reflections. The previous sections cover what occurs in the environment during this time, but now the focus must turn to the radar itself. As covered in section 2.2.1, the world is full of EM noise. The radar cannot detect which received EM waves are related to those it transmitted. Instead, the receiver listens for EM waves covering the frequency at which it transmits and tries to differentiate the reflected signal's environmental noise. To differentiate a signal from a target and noise from the environment, radar designers must create and modify the signal power relative to the noise. This ratio is called the signal-to-noise ratio and is vital in detecting targets using radar [12]. The radar range equation is a volatile equation that can change to fit the radar's need and fidelity. This equation finds the signal-to-noise and signal-to-interference ratio for the radar.

Many factors can also define this how the radar receives EM pulses. Radar can measure the target position, polarization, and resolution. These different measurements can all be precise if desired but come at the expense of performance. For instance, the smaller the radar's resolution, the faster the simulation must process each frame. Modeling the smaller details required to represent a finer resolution takes more processing power.

2.2.2.3 Antenna Gain Pattern

One of the critical aspects of radar is the gain pattern. The gain pattern determines the area in which an antenna transmits and receives EM waves; in other words, it shows the area where an antenna can detect a target. It shows details such as the main lobe and side lobes. The main lobe is where most of the antenna's power is directed. The side lobes are areas where unintentional radiation is emitted. The invisible nature of EM waves is difficult to imagine because they are invisible. In this same way, an antenna covers a particular area, and the gain pattern determines the strength or weakness in which an antenna transmits or receives a signal at a specific location.

Since the antenna transmits in a three-dimensional space, the gain pattern is also three-dimensional. The gain pattern used to be represented by two-dimensional polar graphs conveys the gain pattern, but today, three-dimensional representations are more helpful with technology. Figure 2 shows two types of gain patterns: symmetrical and asymmetrical. The asymmetrical patterns are more complex and are not widely used in radar antennas; instead, the symmetrical ones are preferred to detect and track a target more accurately. This accuracy happens because a more definite, precise beam gives a better indication of the target's location.
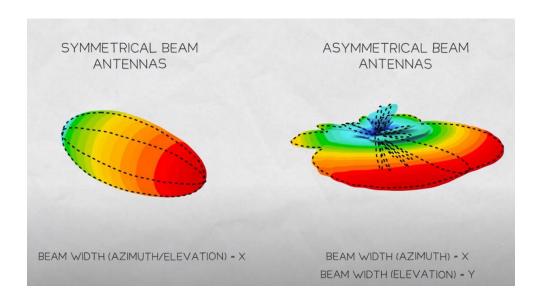
Figure 2. Three-dimensional representation of gain patterns [14]**.**

2.3 Data-Oriented Programming

Data-Oriented Programming (DOP) requires a different way of thinking compared to a traditional Object Oriented Programmin (OOP). DOP is relatively new and does not have as much popularity as other programming paradigms such as OOP, nor does it have as much research [15]. Almost every computer science student learns OOP, but few learn DOP in college courses. It has gained more popularity due to the architectures, such as ECS, organized and based on separating data from programming logic or code. Its novelty is not well defined and can have slightly different ideas depending on the developer. The best way to understand the general principles of DOP is to compare it to OOP.

2.3.1 Object-Oriented Programming

OOP is a paradigm where the programmer focuses on objects. Objects are groupings of data and code, or members and methods. Objects, in simple terms, are

"grouped code associated with some internal state." [16] The members define this state, and the methods change the state. To correctly design objects, OOP has four essential pillars: encapsulation, inheritance, polymorphism, and abstraction [2].

Encapsulation hides the details of a system. In OOP, the class structure hides the complexity of the members and methods. When presented with an object, a user only needs to know how to use it, not the details of how it works. Encapsulation is used to achieve abstraction, which is the simplification of a system to its essential parts. Continuing with the same example, the interface that the object presents to the user would be an abstraction. Abstraction is about presenting the necessary components so that an object can be used with encapsulation.

The principle of inheritance is similar to how inheritance works in the real world. An object, called the child, inherits specific methods and members from another object, called the parent. Inheritance is a means to implement the last principle, polymorphism.

Polymorphism occurs when an object can take many forms given the same interface (e.g., a shape object changing into a specific triangle, square, or circle). The details may be different, but each maintains the essential shape functionality. For instance, a circle would have a radius while a square would have a length, but they can both be drawn. Each shape could have an individual interface, but this would be inconvenient as each shape would have separate implementations, such as `drawSquare()` or `drawCircle()`. It would be convenient for each object to implement the same draw function, then a `draw()` function can be used without specifying which shape is being accessedd. Polymorphism, in this way, can help implement the abstraction principle defined earlier. Inheritance is a useful tool to implement polymorphism. Continuing with

13

the shape example, the parent could be a shape, whereas the child would be a specific

shape. The parent class would create the draw() method, and the children would inherit

this method and implement it themselves. This allows for the user to create a shape,

specify the type, and use the same interface for each type, with the possibility of a few

exceptions. While this is a summary and does not dive into the intricate details of OOP, it

gives an adequate understanding to compare against DOP.

2.3.2   Differences of DOP

Llopis argues that programming is simply a way of processing data, so it should

focus on the data itself, not an abstraction of the data as in OOP [16]. The focus on data is

the core tenant of DOP. Instead of organizing data into objects, DOP organizes data into

homogeneous groups. The primary, tangible advantage of DOP comes from efficiently

utilizing cache.

Because OOP is organized around objects, anytime data from an object needs to

be accessed, the entire object must be brought into cache even if the whole dataset is

unneeded. Table 1 shows the storage location of two objects that contain members for

height and width. Each object is listed as eight bytes long and starts at the lowest listed

address.

Table 1

0x04	0x00	0x14	0x10
Height	Width	Height	Width
Object 1		Object 2	

14

Table 1 shows how objects are stored together. If the program just modified the width

data, it would also bring the height into the cache. Instead, DOP emphasizes grouping

data based on similarities over grouping objects. Table 2 shows the implementation of the

two objects in DOP format.

Table 2

0x40	0x60	0x44	0x64
Height	Width	Height	Width
Object 1		Object 2	

The two 'objects' are merely an index for two different arrays, a height and width array.

The member data is stored continuously in memory. An example that shows the

advantages of DOP would be a change in position. If two objects need to have a position

change, OOP would separately load each object into cache. The DOP approach would

simultaneously load both positions and then operate on them, saving both memory and

time. This DOP principle assumes similar data is usually processed at the same time. For

instance, DOP assumes that if one object's position changes, then all object's positions

also change. While this difference may seem subtle, it can have impacts when dealing

with large amounts of data. This way of processing data is especially helpful in a

simulation where code execution tends to happen in frames. Typically, updates to all

entities with similar attributes happen at discrete times within these frames. In the

position example, the position data for all objects are updated by a certain function.

This organization of code and data is especially beneficial in parallel programming. Instead of worrying about modifying objects and race conditions, the programmer can see what kind of data is being modified and avoid the race conditions. While this does not eliminate the difficulty of parallel programming, it does make it easier.

DOP is a way of thinking that sets up principles for a programmer to use. While a programmer can choose to implement it in many ways, architecture can create different structures and definitions to create a program.

2.4 Entity Component System

This section will cover the details of ECS. It will cover the structure, history, and benefits of using ECS.

2.4.1 History

ECS first became popular in gaming. The Entity-Component architecture, as used by Unity, is often confused with ECS [10]. This confusion is due to the lack of knowledge and standardization surrounding ECS, which creates difficulty in understanding the principles and design of ECS. A discussion of the history of ECS will help with understanding its principles and why it was chosen as the focus of this research.

The four principles of OOP led to several problems in gaming. In her talk Rust for Game Development, Catherine West points out a few of these issues [17]. The first is for the responsibility of interactions. Catherine asks two questions: if, in a game, a player touches another player, which of the two objects initiates the touch? To whom does the touch belong? Secondly, inheritance and encapsulation tend to create massive classes that

contain too much information [18]. Encapsulation causes this problem by requiring extra code to be created to handle any extra members. inheritance has a rigidity problem. For instance, what happened if a project needed a combination of two different parents? Inheritance does not deal with this arrangement easily. In this instance, it led to a type of *blob* class. These classes were bloated with members and methods that were unnecessary, and it became a huge resource strain because unnecessary data was being brought into the cache.

The design principle of composition over inheritance became popular to mitigate these issues. This design idea stressed creating classes from different components instead of inheriting from a parent, whenever possible. While this design method helped, issues with decoupling still existed that section 2.4.6 further discusses. Different developers started using the ECS architecture around this same time without knowing about the other developers' work. While it did not have an official release date, ECS was created to implement DOP principles in a clear architecture. Understanding each part of ECS is vital to understand how the architecture works.

2.4.2 Entity

An entity is anything that is not part of the environment. For example, in a game, an entity could be any object composed of simpler parts, such as a player or monster. The entity is a simple concept to link different components together. The entity is similar to an object instance in OOP, except that it does not contain any data or methods. In some cases, this could be accomplished with an identification number or a name.

17

2.4.3 Component

The component is the structure that holds data and is modeled as a `struct`. The data inside the `struct` is the same as member variables in OOP. They keep the state for the different entities; however, the components are stored differently in OOP. Components are all stored together in an array instead of storing objects in an array in OOP. This storage method allows the data to be manipulated simultaneously and saves on execution time, as discussed in the DOP section.

2.4.4 System

Looking at the ECS from the OOP perspective, if the entity is the object instantiation and the component is the equivalent of the object's state, then the system is the object's methods. In ECS, the system holds code that manipulates and transforms the components. Each system has a different focus and allows the designer to determine how the systems function. Few, if any, design patterns or standards exist for how to create systems. The only guiding principle is that systems are to be kept simple and singular in their tasks. The goal should be to operate on as few components as possible to avoid bringing too much data into the cache at one time. For instance, a movement system would contain the code for transforming the position data based on velocity or acceleration. The systems, in most cases, operate on all the components at one time.

While decoupling does have many benefits, when used in practice, it creates some difficulty. It does not allow an entity to know about another. Nor do the systems know anything about entities; they only know about raw data, the components. This separation makes communication between entities, also known as message passing, difficult. While message passing is not impossible, the functionality is limited and is difficult to setup.

18

One of the benefits of the ECS architecture is the ease of parallel programming. DOP already improves the parallelization of the code, but the ECS architecture adds to it. The clear separation of code and data allows for a more organized way to parallelize code. When looking at an EM wave's properties, one system could focus on changing position while another would focus on changing power. The two systems can quickly run in parallel as they would not change the same data. In OOP, the two systems would change the same object and create a race condition that must be accounted for and handled.

2.4.5 Composition

Section 2.14.1 introduced that composition is the idea that an entity would be composed of multiple components instead of having objects inherit from a parent. Composition allows for more flexibility when creating entities. If an entity needs a different feature, adding the component is easy and requires no change to a hierarchy. It also stops large class structure as entities only contain what they need. The ECS is designed around the idea of composition over inheritance and effectively implements it. The entities are merely a composition of components. Systems are the way to transform and process these components.

2.4.6 Software Aging

Software aging refers to how a codebase changes over time. Software that is difficult to maintain and change would age poorly. Software functionality inevitably changes, and these changes can create more costs and problems if not designed to age well. As software ages and different functionality is required, programmers and designers eventually need to change the source code. One aspect of how software is designed that

has a large impact on aging is coupling. As defined by the International Standard, coupling is the "manner and degree of interdependence between software modules." [19] Tight coupling is when modules are highly dependent on one another, meaning that a change in one requires a change in the other. When large software programs are tightly coupled together, a change in the software has a rippling effect throughout the rest of the program. The rippling effect requires a large amount of work for one change. It also provides room for errors to occur if one of the dependent components was not considered, which is a common problem.

The DOP and the ECS architecture is designed around the loose coupling. A change in one system should not affect the other systems. Neither should a change in one entity affect another since the entities know nothing about each other. While both kinds of coupling have their strengths and weaknesses, loose coupling is beneficial for software aging. ECS is often looked at through the lens of computer performance, but this paper also looks at the improvement of software aging. Software aging could significantly impact military software in the future, affecting the cost, maintenance, and software applications testing.

2.5 Rust

Rust is a relatively new systems-level programming language that focuses on speed and safety [20]. One of the main advantages of using Rust is the strict code quality that it requires. These strict regulations require the programmer to write safe code that mitigates data races. The language seeks to find and stop memory errors during compilation instead of at runtime. Many popular programming languages, such as Java

and C#, use a garbage collector to manage memory automatically, but Rust uses a system of ownership [21], [22]. This system does not occur at runtime and is not as expensive as a garbage collector; subsequently, making the language just as fast as languages without a garbage collector, but arguably, just as safe as those with garbage collection [23].

2.5.1 Ownership

In Rust, ownership is the system that allows Rust to create a memory-safe program without using garbage collection. While the system works for primitive types, such as chars and ints, it is mainly built for dynamically allocated classes on the heap. This memory is referenced via a pointer. The best way to explain ownership is through an example. In this example, a pointer, called str1, is created for a dynamically allocated string. Another variable, str2, needs to point to the same string. If a statement such as str2 = str1 is used, a shallow copy occurs; that is, the pointer is copied to str2. Both variables point to the same data, a change in one would mean a change in another. Also, if memory is deallocated for one, the other would become a null pointer. Rust avoids the null pointer problem by requiring that once the pointer of str1 is copied to str2, str2 takes ownership, and str1 is no longer valid. The pointer is valid when str2 gives str1 ownership. Rust allows for borrowing within this ownership system. Borrowing allows a variable to use the value that another variable owns. In the previous example, if the programmer would still like str2 to have access to str1 without taking ownership, then str2 could merely borrow the pointer. In this way, both variables can access the data. Variables can also do a mutable borrow, although it can make things more complicated. Within the same scope, a reference cannot be both mutably and immutably borrowed.

21

While multiple immutable borrows can occur in one scope, the compiler limits the scope to one mutable borrow. This borrowing system precludes race conditions from occurring, especially in parallel code.

2.5.2 Lifetimes

Another vital system in Rust is the lifetime system. A lifetime is the "scope for which that reference is valid." [24] The main objective is to prevent dangling references, a reference that refers to a null value. While the ownership system and borrow checker can also prevent dangling references, the lifetime is another defense line that is more specific to the problem. Lifetimes exist in other programming languages, but they are implicit instead of explicit. In some cases, Rust requires the programmer to define the lifetime when it is unclear. The explicit declarations make the idea difficult to understand since this explicit definition is unique to Rust. An essential trait of lifetimes is that they are relative. A lifetime on its own does not tell the compiler anything. It must have another variable or term to compare the lifetimes. For instance, if a lifetime of g is defined for variables a and b, then the compiler can compare the scopes of the two variables and determine the produced behavior. Lifetimes are mostly useful in functions when returning references, and the return is dependent on a conditional. In this case, the compiler does not know which reference is returned and how the references' scopes relate to the returned value.

2.5.3 Specs Parallel ECS

SPECS is a Parallel ECS framework for Rust. It includes a few features that make it useful: component storage, system framework, and world management. The different types of component storage take advantage of RAM and cache efficiency. The two-

component storage types used in this research are DenseVecStorage and HashMapStorage [25]. The former creates two vectors, one for the data and one for the associated entity id. Specs documentation explains that DenseVecStorage is best used for bigger components than the architecture size and components that are frequently used [25]. Using a redirection table, the DenseVecStorage can make bigger components than the architecture size efficient for RAM. The HashMapStorage is best used for entity-specific components and not used often because the hash insertion is expensive.

The system framework available in specs is easy to use. It also automatically assumes everything runs in parallel, unless multiple systems request write access to the same component. This feature is appealing because writing parallel code can be challenging. Specs allow for specifying which components each system needs and whether it reads or writes to these components.

World management takes care of the internal data structure's setup and maintenance. It also is responsible for holding the dispatcher, which is necessary for parallelizing the systems. The dispatcher keeps the systems in check that have dependencies so that they can run in parallel. If one system depends on the result of another, the dispatcher takes care of the details of running the two systems.

2.6 Summary

This chapter covered the basics of a radar system and the environment. It discussed the properties of an EM wave and how it interacts with its environment, and the radar system's structure. It covered how these different aspects of radar can be modeled in different levels of fidelity. Each component of these systems can add or subtract from the

fidelity. Many factors can be included and varied in the model to affect this fidelity. However, this section was not exhaustive but covered a breadth of different areas to represent the problem at hand.

This chapter was necessary to understand the information and literature that supports this research. Without understanding each topic, the research methods and results would not make sense. Each topic contributes to the radar model this research simulates, the design of which is covered in Chapter 3.

III. Methodology

3.1 Overview

The purpose of this section is to describe the radar model for the simulation. The intent of the model is to represent radar by implementing certain charterisitcs in ECS. First, it introduces the model's fidelity and the Object-Oriented Programming (OOP) package characteristics, MIXR, from which it is based. Then it discusses the data necessary to represent elements of the model. Finally, an outline of how the model should be tested is proposed.

3.2 Fidelity

Chapter II demonstrated that radar could be modeled with different levels of detail depending on the simulation or study's purpose. The level of detail modeled is called its fidelity. As covered in Chapter II, a model can be made to simulate the speed at which an EM wave travels and track every wave created by the transmitter. This level of detail would slow down execution performance due to the processing and memory requirements that it would not achieve to simulate the entire radar process in real time [13]. Bowen states, "simple models that provide useful insights are prefered over complex models that do not achieve the goals of the model." [26] This model's fidelity aims to simplify the radar process to provide useful insights into the different features and effects of targets, environment, and radar setup.

The model in this research is primarily based on the fidelity described in the MIXR package. Outside of the basic radar model outlined in Chapter 2, "out of the box" models included with MIXR have six characteristics that this model accounts for: [4]

- Antenna gain pattern as a function of beam angle

- The range between a source and target(s)

- The relative velocity between a source and target(s) (i.e., Doppler shift)

- Electromagnetic characteristics (frequency, pulse width)

- Target radar cross-section as a function of the relative geometry

- Electronic countermeasures (i.e., jamming)

These characteristics are defining factors for the fidelity of the model. Everything else in the simulation is made to represent and simulate these characteristics.

3.3 Radar Process

Before understanding this model's different characteristics, understanding the stages of radar for this fidelity is necessary. The different states in which the data can exist is referred to as a phase, and a transformation marks the beginning and end of a phase. A breakdown of these phases from the perspective of radar is necessary to understand the model. Using the RRE, all the variables associated with detection are shown as follows:

$$P_r = \frac{P_t G_t G_r \lambda^2 \sigma}{(4\pi)^3 R^4} \qquad\qquad \text{Equation 2}$$

Where,

P_r is the received power in watts

P_t is the peak transmitted power in watts.

G_t is the gain of the transmit antenna.

G_r is the gain of the receive antenna.

λ is the carrier wavelength in meters.

26

σ is the mean RCS of the target in square meters.

R is the range from the radar to the target in meters [12]

These terms are covered in section 2.2. In an OOP based simulation, these variables are often represented by objects at different levels of detail. In an ECS design, the systems compute the values needed using data defined by the components. The equation 2 is only one form of the RRE, and it assumes a collocated transmitting and receiving antennas. However, other situations exist with separated antennas, and so a model based on the above equation would have a high degree of coupling to one situation. It would not easily allow for another situation to be modeled correctly. Rather than using the complete form of the RRE to calculate the desired values, it would be better to calculate different aspects in phases.

The first phase would be the power density, Q_i, at the range of the target from the transmitting antenna:

$$Q_i = \frac{P_t \, G_t}{4\pi \, R^2} \qquad\qquad \text{Equation 3}$$

This calculation represents the transmitted wave from the perspective of the target. The second phase would be the reflection of that perceived wave from the target into the environment. This reflected wave is simply the product of the power density at that distance and the RCS value of the target:

$$P_{refl} = Q_i \sigma = \frac{P_t \, G_t \sigma}{4\pi \, R^2} \qquad\qquad \text{Equation 4}$$

From the receiving antenna's perspective, the power received considers the range between the target and antenna and the receiving antenna's gain, bringing the model to equation 1.

The goal of this simulation is to model the process through these different perspectives. This approach should allow for a high degree of decoupling and division of labor between the different systems necessary within the ECS architecture. Each system is also responsible for transforming the data in each phase without interfering with other systems' transformations. As a complete simulation, these systems accurately model the six characteristics presented in MIXR.

3.4 Model Characteristics

MIXR is an OOP-based radar simulation framework that allows for radar implementation tailored to an intended purpose [4]. A discussion of the six significant radar characteristics modeled with MIXR defines a clear methodology for how this research's simulation is modeled.

The first characteristic addresses an antenna's gain patterns. The antenna gain pattern could be modeled by simple shapes like a prism to a highly complex 3D model with varying directivity at certain angles. This model's fidelity hopes to achieve a simple pattern with the main beam and average sidelobes.

The second and third characteristics identify the range and the relative velocity between a source and a target, respectively. These characteristics seem straightforward as a simple equation could calculate the values; however, this method would not accurately model a radar because it does not have perfect information, as these equations require.

Instead, these characteristics largely depend on how the EM waves are abstracted. Each

of these is dependent on data extracted from EM wave detection. For instance, a radar

calculates range by measuring the time between when a pulse was sent and when the

radar detects the pulse's reflection, multiplied by c. The travel time needs to be abstracted

with the rest of the EM wave and affects these calculations' accuracies. While the

calculations may be simple, they are tightly coupled with complex abstractions that affect

how the data is represented.

The fourth is an abstraction of EM wave characteristics. The model should

abstract the number of pulses, time, and reflections of EM waves through this

characteristic. A transmitter can create many pulses in the span of a few milliseconds, and

the goal is to simulate them in lumps, which give similar results to real-world effects.

The fifth, target radar cross-section (RCS), is a function of relative geometry.

Similar to the antenna gain pattern, this characteristic can be represented in many

different ways. A highly detailed approach explores the physics behind reflections and

scattering to provide a highly accurate RCS. This model represents these detailed

characteristics through a data file given by the user. It only extracts values calculated

using these complex methods and applies them. In this way, the model is not responsible

for the calculations but can still use and represent a high fidelity if necessary. A target has

a set of values for RCS at certain angles and returns the values illuminated by a radar.

The last characteristic to be modeled is electronic countermeasures, otherwise

known as jamming. This model should be able to emit EM waves that interfere with

received EM waves. In this way, the transmitting signal does not accurately interpret the

signals from the environment.

3.5 Data

Along with the characteristics, the model must contain two entities: radar and targets at the most basic level. Without these two entities, a radar model cannot exist. An analysis of these elements in each stage is important to understand the data necessary to represent them.

The first stage is from the antenna to the target. The antenna has a set of basic parameters that must be implemented to characterize the EM waves it would send. This data includes power, wavelength, and gain. In this first stage, the data must be combined with the environment, range, and power density at the target's location. This stage also includes external factors that could impact the transmission of the wave through the atmosphere.

The second stage includes transforming the data that the target sees from the antenna to reflection data. This stage is much different from the last because the data sent back needs to be processed individually. Beforehand, the data could be transformed as a whole EM wave. Now, the target affects each piece of data differently. The frequency is affected by the doppler effect, but this does not affect the reflection wavelength. The RCS is affected by the angle representing the EM wave hitting the target, but it does not affect any other data. Each piece of data should be processed individually, achieving high modularity, thus having little to no effect on other parts of the program if these individual transformations do not occur.

The third and last stage is the signal reception stage and is similar to the first. In this stage, the antennas receive the EM waves from the environment and produce some output. As this is a simulation, the systems may contain perfect information, but the

received data should not reflect this. Instead, this model should accurately reflect a real radar situation. The data that is written to the antennas reflects this information.

3.6 Testing

This simulation needs to be tested to validate that it calculates the correct values. Testing is implemented using the Rust testing methods. These methods allow for unit, documentation, and integration testing. The unit tests are responsible for testing the functions basic functions that calculate values, such as a target's RCS or whether two objects have collided. The integration tests check that the interaction between larger systems produces the correct values. For instance, an integration test would check that a correct reflection was created after a target is illuminated. This research does not include documentation testing.

3.7 Summary

This section covered the design of the radar model and characteristics to be implemented. It showed how radar will be modeled by examing the process of radar and how it can be divided into data structure and processes. While many implementations could have been chose, the characteristics were chosen from a proven radar model that served as a good basis to create a ECS based simulation.

IV. Results and Analysis

4.1 Overview

This chapter presents and analyzes the the program structure. A discussion regarding the structure of the program is covered first. It is also necessary to cover the reasoning behind many design decisions, exposing underlying patterns within ECS and the benefits and pitfalls of the architecture. Finally, the results of the testing are analyzed to review the efficacy of the model.

Table 3. Components and their attributes

Position		EMWave	
Variable	**Type**	**Variable**	**Type**
x	32-bit float	azimuth_width	32-bit float
y	32-bit float	elevation_width	32-bit float
z	32-bit float	frequency	32-bit float
direction	32-bit float	power	32-bit float
Illumination		wavelength	32-bit float
Variable	**Type**	**Velocity**	
angle	32-bit float	**Variable**	**Type**
rcs	32-bit float	x	32-bit float
frequency	32-bit float	y	32-bit float
power	32-bit float	z	32-bit float
wavelength	32-bit float	**Target Illumination**	
Antenna		**Variable**	**Type**
Variable	**Type**	illumination	Vec<Illumination>
azimuth_width	32-bit float	**RCS**	
elevation_width	32-bit float	**Variable**	**Type**
frequency	32-bit float	angles	Vector<f32>
power	32-bit float	values	Vector<f32>
wavelength	32-bit float	avg_rcs	32-bit float
gain	32-bit float		

4.2 Program Structure

For this program, the structure can be explained by examining the entities, components, and systems. Following the principles of DOP, beginning with the data is the best way to approach the structure. Table 3 depicts component data and how they are organized. For instance, the position component is made of x, y, and z variables that are a 32-bit float type.

Each of these components makes the different entities. Table 4 depicts the entities created for this simulation and the components attributed to each.

Table 4. Entities and their components

Radar	Target
Antenna	Position
Position	RCS
EM Pulse	Target Illumination
EM Wave	Velocity
Position	Antenna

While the components merely make up the model's data, the design's central portion comes from the transformations. Table 5 depicts the systems and the components involved in each. These systems run in a specific order (at 60 Hz), a framerate standard for simulations. A short discussion of these systems provides a clear understanding of the simulation's functionality.

Table 5. Systems and the components they use

Transmit Signal		InteractionDetection	
Read	**Write**	**Read**	**Write**
Antenna	EMWave	Position	TargetIllumination
Position	Position	EMWave	
RCSSystem		RCS	
Read	**Write**	**DopplerShiftSystem**	
RCS	TargetIllumination	**Read**	**Write**
JammingSystem		Velocity	TargetIllumination
Read	**Write**	**ReflectionSystem**	
TargetIllumination	EMWave	**Read**	**Write**
Antenna	Position	TargetIllumination	EMWave
Movement			TargetIllumination
Read	**Write**		Position
Velocity	Antenna		
	Position		

The transmit signal is a system that creates an EM wave entity. The interaction detection system uses the created EM wave and checks for a collision with any entity that has an RCS component. If a collision occurs, it creates a target illumination component.

Once a target has an illumination, the RCS and doppler shift systems apply the physical effects to the illumination's RCS and frequency attributes. If a jammer exists, the jamming system creates an EM wave designed to interfere with the receiver that created the initial wave. At the same time, the reflection system creates a reflected EM wave.

Next, the antenna receiver system runs. This system is like the interaction detection system but runs from the perspective of the receiving antenna. Any reflections that it picks up flow through the radar system. Finally, the movement system transforms the position of each component based on velocity or rotation values.

4.3 Implementing Model Characteristics

This section discusses and analyzes the implementation of each of the characteristics through the lens of the program's structure. The abstraction of electromagnetic characteristics was implemented first. The EM wave was created as an entity with a single component. The component captures the attributes of a single EM wave such as frequency, wavelength, and power. It represents multiple pulses as an emission. Each of these attributes could have been abstracted into different components to create one entity, making parallelization easier, however, it would make detection interactions and illuminations more difficult. The entity, not one component, is responsible for the interaction. Splitting into components and giving one attribute responsibility would not reflect the behavior of a real EM wave. As discussed later in this chapter, the illuminations must be stored differently and not as components. Recording the illumination would not be possible with this storage solution.

An antenna's gain pattern as a function of beam angle was implemented through the antenna and a data file. The data file describes the gain at a given angle for the antenna. The systems apply the gain to the EM wave depending on the angle at which it was transmitted or received.

The range between the targets is derived from the power level that the antenna receives and the abstracted time that it takes to move between the two entities. The simulation has perfect information about the range, but this method is used to provide a more accurate model as this is how a real radar calculates range.

The Doppler shift system calculates the Doppler shift of the reflections. This system reads an illumination and calculates the relative velocity of the two entities. The velocity

35

can then be used to find the frequency of the reflected EM wave. This new frequency is written in the illumination.

The RCS of a target is variable depending on the angle at which a target was illuminated. Variable RCS was achieved through a method of mixing the doppler shift and gain pattern methods. The RCS is gathered from a data file that describes the RCS for each given angle like the gain pattern. The value is then applied to the illumination like the doppler shift.

Finally, the last characteristic, electronic countermeasures, was easy to implement by adding a system before the reflections were created. If the target had an antenna and had an illumination, another EM wave would be transmitted to counter the reflected EM wave's signal.

4.4 Design Analysis

This section analyzes the ability to base a real simulation on the ECS architecture. Many avenues existed to implement the model, with some fitting into the ECS architecture better than others. As in the design of *Polyphony*, a graphical user interface based on ECS, many issues arose with the implementation that could be mitigated by a less refined ECS design [27]. Collisions, data representation, processing methods, and modularity are three major topics that emerged, showing the advantages and disadvantages of ECS.

4.4.1 Collisions

One of the most challenging tasks in this design was creating the collision system. As described in section 2.4.4, systems are independent and passing information between

them is difficult. This independence makes collision systems difficult because they must pass information. Implementing this system highlights a major issue with ECS: cross-system communication [28].

The collisions of interest include the EM wave and target and the EM wave and antenna. In any given collision, two events must occur: detection of the collision and a response to the collision. In a more straightforward scenario, if a ball collides with a bat, the program must detect the ball hitting the bat, then it must bounce the ball off the bat. ECS has two principles that make this difficult: division of labor and decoupling.

The first is more of an implicit principle derived from the idea of efficiently using cache. *Division of labor* is a term used to capture the idea of breaking up the processing into simple tasks through different systems. Ideally, systems should be working with as few components as possible to accomplish simple tasks [2]. If a system is created to do complex tasks, it likely works on many components, which is contradictory to one of the main advantages of ECS and DOP, cache efficiency. If the cache tries to bring in too many components, it does not have enough memory to store them and result in a cache miss, which ECS is designed to avoid. While a complex system with multiple tasks and few components may be possible, this principle exists because it is not likely. Systems should not be designed to accomplish the most work possible; instead, they should aim to have as few components as possible. If a system has too many components and overloads the cache, the benefit of ECS is lost. For this implementation, creating an entire system for collisions can be complex. This division is difficult because of decoupling. Instead, dividing up the detection and response into different systems would be the correct approach.

Decoupling is an explicit principle that calls for independence between systems and entities. Entities do not need to know about other entities, and one system should not need to know about another system. Each system is concerned with its component data and nothing else, which makes communication between different systems and entities difficult.

Specifically, for this program, when a radar illuminates an entity, the illumination must be detected by one system, then a reflection, which is the response, must be created by another. This division of the detection and response into two systems is problematic because the two cannot communicate about the details of each. The simulation could break either the division of labor or decoupling principles to create collisions, but SPECS makes this difficult. Breaking decoupling is difficult in SPECS because it has system classes that do not allow for any communication type. Breaking the division of labor principle is problematic because the system could become overloaded in the future. The system would work on five different components and combine four different systems: detection, reflections, RCS, and doppler shift. While this implementation is possible, caution must be exercised because future development must be considered. The system may be reasonable for now, but this design would make it vulnerable to bloating because it would contain any features desired that occur between detection and reflection.

Another avenue that could be taken is to create a component to record the illumination. This component would be different from others as it only exists to record the collision. One issue exists with this avenue: an entity having multiple components of the same type. If a component is created for a collision, then the target must accommodate multiple collisions if more than one antenna's gain pattern covers that

target. Accommodating multiple components of the same type is not bad programming in ECS, the idea is not popular and not recommended, but besides this recommendation by the community, it does not have support from SPECS. When two components of the same type are added, the framework writes over the component added first with the second component.

With all these considerations in mind, it seemed the best route would be creating a workaround for the multiple components. The reason is that a larger system seemed like a slippery slope. A blob-like system developed in OOP is highly undesirable and seemed much more likely if a strictness was not upheld in this labor division. Instead, the multiple components deviate from a pure ECS architecture, but it seems to have a small impact on performance than the implications of a complex system or working around the decoupling of systems. A workaround was created by assigning a TargetIllumination component to a target and giving it an attribute of a Vector. This way the vector holds multiple illuminations while the target still has only one component.

The collisions were a significant design issue to overcome with ECS. The architecture does not accommodate this kind of behavior well and shows one of its weaknesses. In this situation, it does seem that an OOP representation may have been better suited to the problem.

4.4.2 Data Representation

A pure ECS architecture is strict on the structures that exist in the program. The only structures that exist are components and systems. Entities are merely a way of organizing the components, the simulation itself does not use entities. This minimalistic

39

structure makes it difficult to model complex realities, such as an EM wave. Overall, this makes representing different kinds of data challenging.

EM waves become difficult to model because of their behavior. They are modeled as both instantaneous and persistent. Like a wave, they carry information through different objects instantaneously, but they are also modeled as gain patterns that can cover a set area for a whole frame. In an OOP program, these are modeled as messages, but ECS does not support these specifically, instead, a component must be made into a message. So, an EM wave can only be represented as a component or an entity.

As a component, it would be attached to a radar entity and represent a gain pattern. This structure creates a problem because the gain pattern is not an aspect of a radar, but instead is an aspect of an antenna, which cannot have a component because it is a component. The antenna must then become an entity and create communication problems with a radar antenna. Overall, the EM wave represented as a component creates a ripple of issues and is best represented as an entity where it interacts with other objects, just as in reality.

The illumination data, discussed in the collisions section, was represented as a component. The issue with this representation was that entities could not contain multiple components of the same type. This data could have been represented as an entity and allow for entities to have multiple illuminations, but this again creates issues with communication between its associated entity.

These issues reveal a problem in ECS with representing complex data structures. The multiple component issue may only exist in SPECS and not be indicative of ECS's ability to represent structures, but a hierarchy of components could be helpful in each of

these scenarios. The decoupling aspect of ECS is helpful in many situations, as covered later in this chapter, but can create problems when trying to communicate between different entities.

4.4.3 Processing Methods

Another difficult process to implement was the reflection process. Realistically, an EM wave could produce multiple reflections. For instance, if a pulse is transmitted and reflects off a target, then that reflection hits another target and produces another reflection in a seemingly infinite loop. Traditionally, simulations run in a game loop at a set rate of 60 Hz. At each iteration of the game loop, the systems are run in a specific order to promote a certain data flow. The loop makes the reflection system difficult because a system would need to run at each creation and propagation of the EM wave. This loop could run indefinitely depending on the simulation scenario, which would require a lot of boilerplate code in the game loop, given the decoupled nature of ECS. One avenue that many games take to solve this issue is an event system. Common in OOP, an event system invokes either a method or any other kind of function when an event occurs. In this case, if an emission hits a target that produces a reflection, an event would trigger the reflection system, which would trigger a propagation system. While this idea is possible to implement with ECS, it does not seem to be the best use of the architecture.

The problems with an event system revolve around two principles: code organization and cache efficiency. The first problem is minor, but still a problem. An event system requires some type of glue between the systems. While this was already somewhat violated in the collision system, this kind of design would create more tightly coupled systems because it would require the implementation of a callback function. The

system requires one system to call another, but if it does not, then it would not be an event system. Secondly, the ECS architecture's most attractive quality is the efficient use of cache and an event system does not prioritize this ideal. Cache is prioritized through batch processing. The systems bring in their components all at one time and process them together. An event system is triggered when only one target needs to produce a reflection. It does not address the situation where many targets produce a reflection at one time. In that case, the reflection system would be called for each target and would process them individually. In this way, an event system does not prioritize the cache in the way intended for an ECS architecture. While the event system could potentially produce a faster simulation, this research explores the implementation of an ECS architecture in a simulation.

The best implementation of this reflection process often follows the traditional game loop approach, though it creates a basic restraint on the simulation. It can only produce the number of reflections specified in the game loop. The positive of this design is the modularity of the ECS architecture. It allows for multiple calls of this process in the game loop.

4.4.3 Modularity

One of the benefits of using an ECS architecture is how code is decoupled. One avenue of research for this topic was modularity. A simulation that could have components and systems as plugins would be highly beneficial. This feature could allow for both a spectrum of fidelity and different features, depending on what the implementation desired, with little to no change to the rest of the code. While ECS has

many principles, which seem to allow this modularity, it was clear that a basic pattern must exist as a foundation to build.

The pattern can be seen in the main game loop and was discussed in the fidelity section. The reasoning behind this pattern has a few factors involved. First, both component and entity management occur in the main game loop through the world object. So, these different stages were influenced by where this management needed to occur. Secondly, these stages are fundamental to the radar process as described in Chapter 2. Without each of these stages, a radar simulation does not reach the minimum level of fidelity needed for a production level simulation.

Overall, this high level of decoupling did prove to be a major advantage of ECS. As discussed in the Software Aging section of Chapter 2, the nature of decoupled code makes it far easier and cheaper to maintain. This simulation showed the effectiveness of the decoupling by adding in systems and components with ease. The jamming system was added at the end of this research and did not impact other parts of the program.

V. Conclusions and Recommendations

5.1 Overview

This chapter summarizes the research and results from this thesis. It addresses the original research questions considering the research results. A discussion of the impact this research has on simulations and recommended future work concludes this paper.

5.2 Research Conclusions

This research concludes that the ECS architecture can effectively be used in real-time simulations. It successfully built a functional radar simulation based on the characteristics of an OOP-based counterpart, MIXR. An analysis of the implementation revealed patterns where the architecture is optimal and deficient, and it answered the research questions proposed in the introduction.

This research shows that a simulation can be implemented using a pure ECS architecture. A pure ECS architecture is not ideal for radar simulations because they need cross-system communication, which is forbidden in a pure architecture. Pure implementations also struggle with representing all the data necessary for complex simulations. While the simplicity of ECS is a strong advantage, it is a disadvantage when modeling complex structures. Many characteristics of ECS can be utilized along with a different architecture.

Applying ECS to a radar simulation seemed to show fundamental issues with a pure implementation. When using a pure ECS, communication between systems is not possible. If ECS is to be used in simulations that require cross-system communication, a blend of ECS with other architectures would be necessary. It also shows that the strict

44

data layout of entities and components may not be the most efficient layout. A tension

exists between the strict layout and the ability to efficiently and intuitively relate complex

data. While further studies would need to be conducted, it seems that the ability to

separate data while keeping a relationship with each other will help with parallelization

and not interfere with decoupling between systems.

It seems that the ECS architecture could be combined with OOP patterns to

create a simulation. Using both in conjunction could help to highlight the advantages

while mitigating their disadvantages. Using an event system or callbacks, as common in

OOP, could help fix the strict processing loop that must be followed in ECS. Using ECS

principles of decoupling and batch processing can improve cache efficiency and code

maintenance that has become an issue in OOP.

5.3 Research Significance and Future Work

As the hardware improvements continue to decline, the software must become

more efficient to improve its quality and performance. The ECS architecture is a practical

implementation of DOP that allows for this kind of software performance increase. This

work has focused on researching, building, and analyzing the use of ECS in simulations.

This research has shown that the ECS architecture can be used to build real-time

simulations. A strict, pure implementation may not be ideal, but it has advantages that

can be utilized with other patterns.

The ECS architecture has many areas of future work that could benefit simulation

and the Air Force. The following suggest areas of future work:

Many different ECS architecture exists, both for Rust and other popular languages such as C++. While some research compares different ECS frameworks, few have done an in-depth study that shows each [28] strengths and weaknesses. A study into how each of these frameworks could be best used in a simulation would be helpful. Looking at how these frameworks tackle ECS's disadvantages would show which use case they best support.

An ECS framework specific to military needs would provide a great resource for the Air Force. A framework suited to military needs could directly deal with common problems seen in ECS for military applications.

Research into using ECS with other design patterns. This research focused on creating a simulation based on a pure ECS implementation. Creating a simulation with less rigidity to ECS principles would create a more effective program.

A study of the execution performance of this ECS simulation could be helpful. Rust has packages that can aid in benchmarking. Comparing the speed to an OOP counterpart could provide objective evidence to it's performance.

Appendix

Bibliography

[1]   T. Härkönen, "Advantages and Implemenation of Entity-Component-Systems," Tampere University, 2019.

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

[3]   M. Danielsson and G. P. Bohlin, "A High Performance Data-Driven, Entity-Component Framework For Game Engines With Focus on Data-Oriented Design." [Online]. Available: https://gitlab.bthstudent.se/autious/mg.

[4]   B. T. Garland, D. D. Hodson, S. L. Nykl, R. Dill, and M. R. Grimaila, "A Framework for Modeling a Real-Time Radar System," pp. 1–4.

[5]   A. Martin, "Entity Systems are the future of MMOG development – Part 1 – T-machine.org," Sep. 03, 2007. http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/ (accessed Nov. 03, 2020).

[6]   P. Buttfield-Addison, M. Geldard, and T. Nugent, "Entity Component Systems and You: They're Not Just For Game Developers (SAConf NY 2019) - YouTube," Feb. 2019, Accessed: Jul. 20, 2020. [Online]. Available: https://www.youtube.com/watch?v=SFKR5rZBu-8&feature=youtu.be.

[7]   A. Sudol, S. Gordon, H. Ford, E. Inclan, D. N. Mavris, and M. Z. Miller, "Simulation of radar signal propagation via multipath," 2018, doi: 10.2514/6.2018-4060.

[8]   D. D. Hodson and J. Millar, "Application of ECS Game Patterns in Military Simulators," 2018.

[9]   D. H. Peters, T. Adam, O. Alonge, I. A. Agyepong, and N. Tran, "Republished research: Implementation research: What it is and how to do it," *Br. J. Sports Med.*, vol. 48, no. 8, pp. 731–736, 2014, doi: 10.1136/bmj.f6753.

[10]   L. I. Hatledal, Y. Chu, A. Styve, and H. Zhang, "Vico: An entity-component-system based co-simulation framework," *Simul. Model. Pract. Theory*, vol. 108, p. 102243, Apr. 2021, doi: 10.1016/j.simpat.2020.102243.

[11]   A. V. Davis and S. Wang, "A Concurrent Entity Component System for Geographical Wildlife Epidemiological Modeling," *Geogr. Anal.*, 2020, doi: 10.1111/gean.12258.

[12]   M. A. Richards, J. A. Scheer, and W. A. Holm, *Principles of Modern Radar*, I., vol. 1. Edison, NJ: SciTech Publishing, 2010.

[13]   H. Chen, R. Gentile, and R. Graessle, "Algorithms to Antenna: Radar Target Modeling," Jul. 2019. https://www.mwrf.com/technologies/systems/article/21849840/algorithms-to-antenna-radar-target-modeling (accessed Aug. 11, 2020).

[14]   RF Elements, *Inside Wireless: Antenna Beam Width*. RF Elements, 2019.

[15]   N. Askarbekuly, E. Uzbekova, M. Mazzara, and K. Fedoseev, "A Case Study on Object-Oriented and Data-Oriented Design Paradigms in Game Development," 2020. doi: 10.13140/RG.2.2.16657.66405.

[16]   N. Llopis, "Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP) – Games from Within," 2009. http://gamesfromwithin.com/data-oriented-design (accessed Jul. 06, 2020).

[17]   C. West, *Using Rust For Game Development*. 2018.

[18]   M. West, "Evolve Your Hierarchy," Mar. 07, 2007. http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/ (accessed Jan. 18, 2021).

[19]   International Organization for Standardization, "International Standard ISO/IEC/ IEEE 24765," *Syst. Softw. Eng. — Vocab. 2nd Ed.*, vol. 2017, 2017.

[20]   "Rust Programming Language." https://www.rust-lang.org/ (accessed Jan. 22, 2021).

[21]   "The Java® Language Specification." https://docs.oracle.com/javase/specs/jls/se8/html/index.html (accessed Jan. 22, 2021).

[22]   "C# docs - get started, tutorials, reference. | Microsoft Docs." https://docs.microsoft.com/en-us/dotnet/csharp/ (accessed Jan. 22, 2021).

[23]   J. Blandy, *Why Rust?* Sebastopol: O'Reilly Media, 2015.

[24]   S. Klabnik and C. Nichols, "The Rust Programming Language," *Rust*. https://doc.rust-lang.org/stable/book/ (accessed Sep. 10, 2020).

[25]   "The Specs Book." https://specs.amethyst.rs/docs/tutorials/ (accessed Jan. 09, 2021).

[26]   K. Bowen and K. McNaught, "Mathematics in warfare: Lanchester Theory," *Lanchester Leg. - A Celebr. Genius*, 1996.

[27]   T. Raffaillac and S. Huot, "Polyphony," *Proc. ACM Human-Computer Interact.*, vol. 3, no. EICS, pp. 1–22, Jun. 2019, doi: 10.1145/3331150.

[28]   D. Hall, C. P. Slo, and Z. Wood, "ECS Game Engine Design," 2014.

Acronyms

DOP Data-Oriented Programming. 1, 21, 14, 15, 16, 17, 18, 19, 20, 33, 37, 45

ECS Entity Component System. iv, 1, 2, 3, 4, 12, 16, 17, 18, 19, 20, 22, 25, 27, 28, 31, 32, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46

EM Electromagnetic. 5, 6, 7, 8, 9, 10, 11, 19, 23, 25, 29, 30, 33, 34, 35, 36, 37, 40, 41

MIXR Mixed Reality Simulation. iv, 2, 4, 25, 28, 44

OOP Object-Oriented Programming. iv, 1, 2, 3, 12, 13, 14, 15, 16, 17, 18, 19, 25, 27, 28, 39, 40, 41, 44, 45, 46

RCS Radar Cross Section. 9,27, 29, 30, 31, 32, 33, 34, 36, 38

RF Radio Frequency. 5

1. REPORT DATE *(DD-MM-YYYY)* 26-03-21	2. REPORT TYPE Master's Thesis	3. DATES COVERED *(From – To)* October 2019 – March 2021

TITLE AND SUBTITLE Desiging and Implementing a Radar Simulation Using the Entity Component System	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Garland, Brenen T., 2d Lt, USAF	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/ENY) 2950 Hobson Way, Building 640 WPAFB OH 45433-8865	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT-ENG-MS-21-M-038
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Lab 2180 8th Bldg 145 R034 Wright Patterson, OH 45433 (312) 672-6556 ATTN: Jim Zeh	10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RHIQ
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT DISTRUBTION STATEMENT A. APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

This research explores the implementation of a "medium fidelity" radar simulation using the Entity-Component-System (ECS) architecture. The radar implemented mimics the fundamental characteristics of entities in the open-source Mixed Reality Simulation Platform (MIXR) project, supporting real-time interaction. Previous research has shown the potential benefits of using an ECS-based architecture to support improved execution performance relative to Object-Oriented Programming (OOP) approaches, thus improved real-time interaction requirements. This research implements a well-documented radar model that supports the development of soft real-time human-based interaction simulations. The radar system modeled in this research mimics the "out-of-the-box" fidelity defined in the OOP-based MIXR architecture. This research creates components (i.e., data) to represent antenna patterns, target cross-sections, and emissions. The systems (i.e., computer logic or behavior) create and compute so-called "emission" data processed in phases that represent initial transmission and reception. Validation of the model was accomplished by creating test scenarios and comparing outputs with calculated values.

15. SUBJECT TERMS Radar, ECS, DOD, Simulations

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON Dr. Douglas Hodson, AFIT/ENG
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(Include area code)* (937) 255-3636 x4719 Douglas.Hodson@afit.edu
U	U	U	UU	62	