Simulating a Target's Radar Signature
using Object Oriented Programming and MATLAB

By

Brandeis Marquette

A MASTER OF ENGINEERING REPORT

Submitted to the College of Engineering at
Texas Tech University in
Partial Fulfillment of
The Requirements for the
Degree of

MASTER OF ENGINEERING

Approved

_____
Dr. J. Smith

_____
Dr. A. Ertas

_____
Dr. T. Maxwell

_____
Dr. M. Tanik

October 12, 2003

ACKNOWLEDGEMENTS

This report would not have been possible without the support of many people. I would like to thank my supervisor Jim Kviakosky for encouraging me to enroll in the masters program and for Raytheon for offering such a program to its employees. I would also like to thank Dr. Raymond Samaniego for providing insight and guidance in creating the simulation and helping me understand many of the mathematical and signal processing concepts.

I would also like to express my thanks to the entire Texas Tech staff and guest speakers. Their courses on modeling and analysis, problem solving, OOP, architecting, design, and decision making helped provide inspiration and novel new ideas that made completion of this project possible.

I would also like to thank my wife for having patience and being understanding in the completion of this paper. I would like to thank her for her encouragement and support.

I would also like to thank my fellow Texas Tech students in making this course all the more enjoyable. I would like to thank Chris and TJ for being troopers in completing group projects and for all the laughs. I would like to thank Tim Smith for helping me consume apples and Jim and John for keeping Tim in line. I would also like to thank the Garland folk for the insight and perception they contributed to the class discussions.

TABLE OF CONTENTS

DISCLAIMER

The opinions expressed in this report are strictly those of the author and are not necessarily those of Raytheon, Texas Tech University, nor any U.S. Government agency.

ABSTRACT

In designing today's SAR, MTI, and radar tracking systems, it is often necessary to perform extensive simulations to test the viability of various radar designs. For these simulations, one must be able to effectively reproduce how radar sees a target. Radar perceives objects by transmitting electromagnetic pulses and observing the returning echo. By timing the lapse between pulse and echo, and by measuring the apparent Doppler frequency shift, radar distinguishes targets from the surrounding noise. This range/Doppler information the radar sees, or "video phase history", is the raw data that must exist before any further radar signal processing may be performed.

In order to simulate video phase history, one must model both the radar and its target. The radar model requires simulating physical properties of the transmitted electromagnetic pulses. The target model requires simulating how the radar pulses reflect or "scatter" off the target back to the receiver. A sufficient sampling of scatter points on the target's surface must be achieved in order to distinguish various target characteristics. One must model how the radar pulses reflect off each scatter point and determine whether it is "shadowed" by some other part of the target. Finally, one must model the yaw, pitch, roll, and velocity of each scatter point to simulate the target's motion relative to the radar.

For my project, I propose to develop a simulation to model radar returns. This simulation will model both the radar and its target. Of primary interest, however, is designing the simulation such that a radar engineer may rapidly model complex geometric targets such as an airplane, missile, or ship. To achieve this, a library of basic geometric objects, such as cones, spheres, frustums, facets, and cylinders, will be created using object oriented programming techniques. Each object will "know" how to "uniformly sample" its scatter points, determine its reflectivity, and identify if a scatter point is shadowed by other parts of the target. Each object will have a yaw, pitch, roll, and XYZ velocity associated with it. Using techniques from "rendering", each object's dimensions and orientation will be easily definable by the user. Once the radar and targets are defined, their characteristics will be passed to a video phase history engine, which simulates the target's range & Doppler, as, seen by the radar. The video phase history will then be analyzed using simple radar signal processing techniques.

LIST OF FIGURES

LIST OF TABLES

CHAPTER I
INTRODUCTION

Today's modern radar systems are as complicated and as expensive to build as ever. Radar systems are expected to handle a wide variety of tasks, from tracking ships on the open ocean to detecting moving targets on land to providing advanced surveillance for ground combat troops. They are designed with the latest, most technologically advanced microprocessors, digital signal processors, and FPGAs. The algorithms that process and enhance radar imagery are mathematically intensive and difficult to understand and analyze. Building a radar system from the ground up can prove to be a very challenging and expensive proposition, especially if the algorithms, characteristics, and capabilities of the radar are not fully understood. Prior to actually constructing the radar system, it is often desirable to simulate how various radar configurations or signal processing algorithms perform. While various simulations may be designed to model different aspects of the radar, the simulation addressed in this paper explores how electromagnetic energy from the radar is reflected off targets, collected, and processed to create range-Doppler images of the target.

Simulating radar signatures from arbitrarily shaped targets is a desirable capability to have for many different reasons. Often, when building a new radar, there is a limited amount of real measured target data available. Even if it is possible to use existing radar systems to capture and record target signatures, actually acquiring that data through flight tests is expensive. Data describing the target signature is often needed to perform trade studies, to predict system performance, or to evaluate how to best utilize funds. Once the radar is built, it is not a straightforward task to modify the radar's design. With a software model, however, it is relatively painless to modify radar characteristics or test the performance of various algorithms against different targets before the algorithms are made permanent. In lieu of expensive flight tests, software models may even be used to help perform system validation and verification. Instead of trying to arrange for different ships, planes or ground vehicles to perform maneuvers during different weather conditions, the systems engineer can simply input the appropriate parameters into the simulation and observe how the system performs.

When dealing with mathematical models the question always arises as to how close the model matches the real world. Some models, like those that attempt to predict the weather, are only so accurate due to the chaos inherent to nature. Other mathematical models, such as Newton's laws of motion, yield highly accurate predictions, assuming of course the system is kept within certain bounds and conditions. The mathematical models used to simulate radar signatures of targets, some of which are discussed in this paper, have been shown to be highly accurate. Synthetic target models can closely match features found in real measured data. While such models are numerically intensive, with the advent of faster, more powerful computers, they are capable of operating at a higher degree of fidelity and still execute within an acceptable period of time. This enables freedom to experiment with a wide array of targets, configurations, and scenarios with a high degree of confidence in the accuracy of the results.

Historically, in designing software simulations of radar systems, much thought is invested in the algorithmic and mathematical aspects of the model. Rigid proofs are used to validate the mathematics and much effort is spent in verifying the algorithms. When it comes to engineering the software framework that houses the algorithms, however, not much thought is given. Often simulations consist of disjoint code that seems to have been translated directly from the backs of napkins and scratch paper. The moniker "spaghetti code" aptly describes the difficulty in trying to follow the program thread. Even if a structured or object oriented programming language is used, the engineer often fails to follow sound software engineering principles, negating the very benefits and advantages afforded by structured or object oriented design.

There are many advantages to practicing software engineering principles when architecting simulations or analysis tools. First and most important is the savings in time and money. While slapping together a simulation without investing in preliminary design may save time and money in the short run, those cost savings are quickly eaten by the cost of debugging, rework, and maintenance. A solidly constructed program may require more effort up front; however, the advantages of code reuse and readability are well worth the initial investment. If the original author of the simulation is no longer available, a well-designed program can be more easily understood and adopted by someone else. All too

often, when an engineer leaves a project, the simulations and programs he creates are discarded or have to be rewritten from scratch because they are too difficult to understand and maintain as is. For larger, more complicated simulations, adopting established software methodologies allows for multiple developers to work in parallel. If interfaces are well defined integration is seamless. Another benefit of a well-constructed simulation is that it may serve as the foundation for the software used in the real-world system. Given a solidly constructed model or simulation, deriving the system software may be as simple as translating the simulation code into the native programming language used in the system. If no software methodologies are used, however, the system software inherits the maintainability, usability, and readability problems of the poorly designed simulation.

CHAPTER II
BACKGROUND

2.1 Object Oriented Programming

The following sections provide a brief introduction into the history and basic concepts of object oriented programming. The benefits afforded by OO in the design of the simulation are explored and the underlying philosophy used in developing the simulation is presented.

2.1.1 Evolution of Object Oriented Programming

In the 1960's, engineers were designing small, relatively simple programs. The limitations of the hardware and available programming languages limited the complexity and the potential achievable by software. The discipline of software engineering was still in its infancy and engineers relied on their creativity as opposed to any methodology to design software. Even though the programs were neither large nor complex by today's standards, developers still had difficulty remembering all the data needed to develop, debug, and maintain their code. Unfortunately, this led to "spaghetti code" replete with "GOTO" statements, that was both difficult to build upon and a nightmare to maintain. Clearly new software methodologies and greater levels of abstraction were required to manage the increasing levels of software complexity (Wilkie, 1993).

To address this increasing complexity, in the 1960s and 1970s, higher-level languages like COBOL and FORTRAN were developed. A new style of software engineering was developed known as modular or structured programming. In the 1970's, Al Constantine and Ed Yourdon devised a method of developing software that used the function as its primary building block. Known as structured analysis and design, this methodology allowed engineers to organize software routines by their functionality. Structured analysis was very good at capturing the functionality of organizations and modeling scientific and mathematical algorithms, however it was sorely deficient at modeling and managing the data itself (Wilkie, 1993).

With the development of structured design, the bar was raised for the development of even more complex and advanced software. New challenges such as coupling and cohesion became the limiting factor in software development. Coupling refers to the dependence of one section of code on another section and/or data storage technique. Cohesion refers to how well a set of code and its associated data fit together. With structured programming, data is not managed and an understanding of the algorithms internal to a function is required. When modifying a function, a developer must take special care as not adversely affect other routines that utilize the function. All of these factors contribute to poor cohesion and coupling, and hence the need for an even higher level of abstraction and a more advanced software engineering methodology (Wilkie, 1993).

The problems posed by coupling and the need for strong cohesion within a program was addressed with the advent of object-oriented programming. OOP attempted to solve these problems through the following techniques:

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Composition

The following sections provide a brief introduction to each of these concepts. They document how each OOP concept was leveraged to provide for a cohesive and uncoupled simulation design. Whether the techniques were successful or not is evaluated and a critique of their usefulness in creating the simulation program is made.

2.1.2 Encapsulation

An object is a collection of data consisting of properties (attributes) and methods (functional logic). The properties define the state of the object and the methods describe the object's behavior. For example, Figure 1 illustrates how an object may be used to represent a cylinder. The cylinder's radius and height are properties specific to the cylinder. Properties can be primitive data types like integers, characters, or floating point numbers; or they can be other objects like matrices, cylinders, spheres, or even complex shapes like missiles, ships, or airplanes. Objects composed of other objects is known as

aggregation. Aggregation is especially useful in OOP where groups of objects are contained within larger objects. For example, in the simulation, the target object contains a collection of shape objects like cylinders, cones, spheres, and facets, which taken together define the target's geometry.

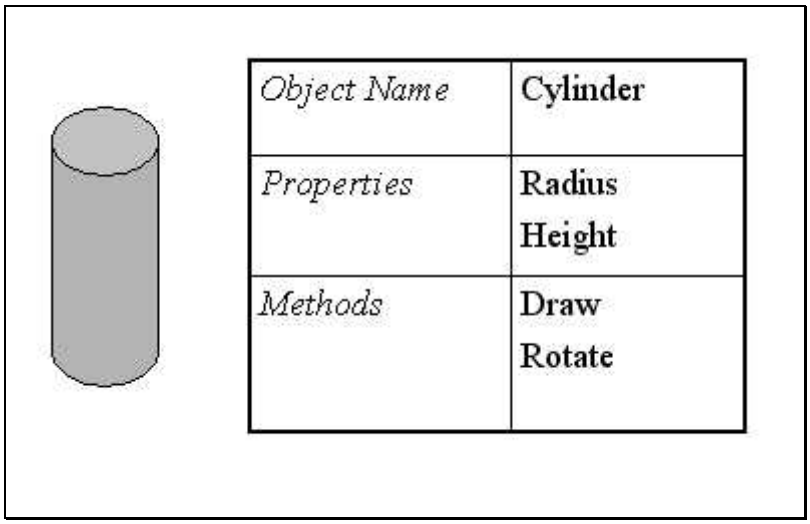

Figure 1. Encapsulation for a Cylinder Object.

The object's methods describe the behavior of the object. Methods are typically used to instruct an object to perform some action, calculation, or data manipulation. In Figure 1, for example, the cylinder's draw method is used to plot the cylinder on the screen and the rotate method is used to rotate the shape about the X, Y, or Z-axis. Two different types of methods can be associated with objects. The first type, called interface methods, are the means of communication between the object and the rest of the program. It is through interface methods that a user gets, sets, or modifies an object's properties. The second kind of method, known as internal methods, are hidden within the object itself and need only be understood by the object's developer. In fact, external users of an object should not even know of the existence of the internal methods within an object. By limiting users of the object to interface methods, the overall complexity of the software is greatly reduced. Hiding internal methods and restricting object access to the interface methods has the added benefit of "shielding" other objects from any internal design changes to the object. This limits the number of knock-on effects to other parts of the program and limits the impact of future modifications to an object's algorithms or data structures. Finally, by limiting the

communication and interaction of objects to interface methods, the amount of coupling within a program is greatly reduced (Wilkie, 1993).

With encapsulation, external accesses to internal properties within an object are restricted to the object's interface methods. In this way, developers strictly control what information flows in and out of an object. Internal properties are protected from being set to invalid values and data passed to the object is validated before it is used in calculations or logic decisions. With encapsulation, the developer may even choose to hide certain object attributes from the user. This concept of data hiding, or data abstraction, further reduces the amount of coupling and complexity within a program. The user need not be concerned with how data is stored within the object, but only with how to access the data through the object's interface methods.

2.1.3 Inheritance

Sometimes objects have similar properties and methods that have just a few minor differences. Instead of creating two separate classes from scratch to represent similar objects, inheritance may be used to share the commonality between similar but distinct objects. For example, a cylinder, facet, and cube all represent different geometric shapes. They are similar in that each shape has a orientation, shape coordinates, and can be drawn or rotated. They are different, however, in that the cylinder's geometry is defined by its radius and height, the facet by its three vertices, and the cube by its length, width, and height. With inheritance, the similarities of the distinct shapes can be grouped together to form a "base-class" from which more specific "subclasses" are derived to handle the intrinsic differences between the different shapes. In Figure 2, for example, "Geometric Shape" is defined as the base-class. This class contains properties that define the shape's location and coordinates along with the methods used to draw or rotate shapes. In creating the cylinder, facet, and cube, it would be useful to not have to rewrite the location and coordinate properties and the draw and rotate methods for each of the different types of shapes. With inheritance, these common properties and methods can be derived or "inherited" from a common base-class. The properties and methods defined in the base class are made available for use in

any derived class. In our example, the cylinder class would "borrow" the draw methods of the shape class and the facet class would borrow the coordinates of the shape class to store the location of its sample points. To handle the differences between the various shapes, properties and methods particular to a shape, such as the radius and height of the cylinder; the three vertices of the facet; and the length, width, and height of the cube, are defined within the sub-classes themselves.

Sometimes with inheritance the same methods defined in the base-class are redefined in the sub-classes. In this case, the methods declared in the lower level base-classes override their inherited counterparts. From the example in Figure 2, if a draw method were added to the cylinder class, it would take precedence over the draw method inherited from the shape base-class. In implementing the code, the two draw methods would be different, defining different actions to undertake when the draw method is invoked for a generic shape versus a cylinder, which must be drawn in a unique manner. In this way, generic methods from a base-class may be overridden to handle special cases when derived classes need to be handled differently. When determining which method takes precedence, the general rule is to search the class hierarchy from the bottom up for like named methods. Once like named methods are found, the search terminates and the first method found is used.

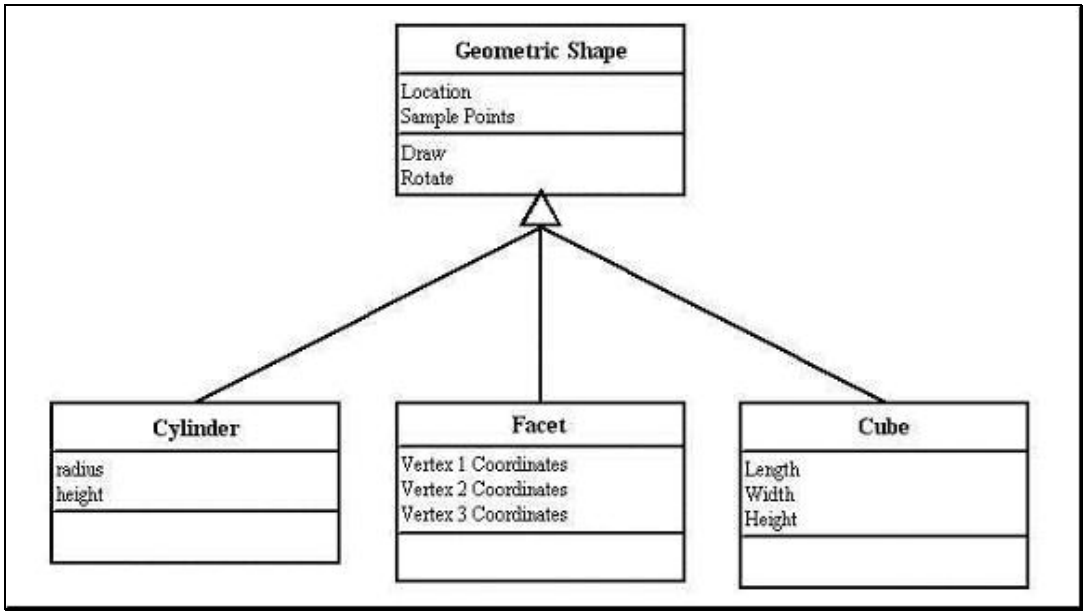

Figure 2. Simple Example of Inheritance.

The inheritance described in the previous section, where sub-classes inherit methods and properties from base-classes, is known as single inheritance. Multiple inheritance allows a class to inherit from more than one base-class. For example, in multiple inheritance the cylinder and cube might inherit properties from not only a shape base-class, but also from a "materials" base-class which defines properties and methods for different types of material like wood, metal, or glass. The OOP community is split as to whether multiple inheritance is a good thing. Some OOP environments, such as C++ and Smalltalk, allow multiple inheritance. Other languages such as Java do not.

The primary concern with multiple inheritance is that it adds unnecessary complexity. For example, multiple inheritance must resolve the situation where similar named methods or properties are inherited from two different base-classes. Another potentially problematic situation is when a sub-class inherits from two different base-classes that in turn inherit from the same parent class. In this case the sub-class inherits twice from the same grandparent class (once from each parent). The programming language or case tool used must be able to resolve this type of problem and provide some type of solution (Dogru, 2003).

In the design of the simulation, multiple inheritance was not used due to the added complexity it introduces. Even though it is supported in MATLAB, whatever perceived benefit it offers is more than offset by complexity it adds.

2.1.4 Polymorphism

Another key feature of object-oriented programming is a behavior known as polymorphism. Polymorphism is the ability of an object to automatically select the correct method to invoke at run time. If the same method is described for a series of different sub-classes, which inherit that same method from a common base-class, then without checking the type of an object the correct method can be automatically invoked. The concept of polymorphism is best understood through example. With polymorphism, the same method must be defined in the base-class and any derived sub-classes. Within the sub-classes, the method is redefined to follow some unique logic or algorithm specific to that sub-

class. For example, in Figure 3, a method to compute the intersection point between a vector and a shape is created. Because vector-shape intersection computations are unique for different shape types, specific intersection methods must be written for specific shapes like cubes, facets, or cylinders. Without polymorphism, to determine whether to invoke the cylinder, facet, or cube intersection method, a series of if-then or case statement logic must be used. With polymorphism, the intersection method may be invoked on any shape that inherits from the generic shape base-class. Based on the specific sub-class the shape belongs to, the proper intersection method is automatically invoked (Dogru, 2003).

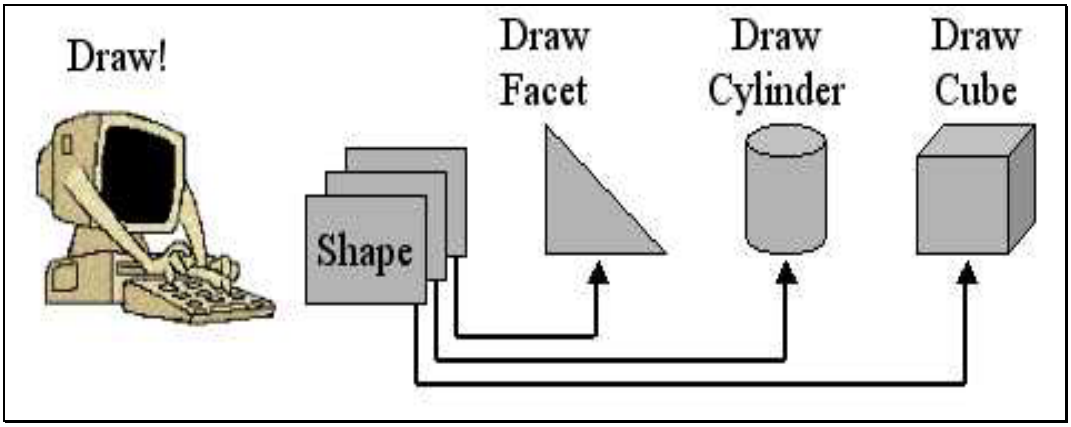

Figure 3. Polymorphism (Dogru, 2003).

Another form of polymorphism that is especially useful is function overloading. In function overloading, the parameters included in a method's argument dictate which code a method executes. For example, two different "location" methods might be defined for a cylinder object. The first location method might accept the cylinder's sample points as input, from which the location of the cylinder is set. The second location method might accept the origin of the cylinder as input, from which the location of the cylinder is set. It is clear that with overloading, both location methods position the cylinder about some fixed point, however they accomplish the task using different inputs and different calculations.

2.1.5 Composition

Encapsulation, inheritance, and polymorphism are all powerful object oriented programming techniques used to represent complex relationships between classes. Another powerful tool available for

describing class relationships is composition. With composition, classes are contained within other classes. While inheritance provides for a mechanism of assuming ownership of a duplicate set of methods, composition provides a means of representing a part-whole relationship between classes. Inheritance represents the "is a" relationship between classes. Composition represents the "has a relationship between classes. For example, in the simulation, the target class "has a" combination of different shapes such as cylinders, facets, and cones that define the target's geometry. As discussed in previous sections, the cylinder "is a" shape and the facet "is a" shape, hence they exhibit the relationship described by inheritance (Dogru, 2003).

2.2 Object Oriented Programming Languages

Several different programming languages offer OOP capabilities. Some of the more popular languages include C++, Smalltalk, JAVA, and MATLAB. Each language offers key object oriented ingredients including encapsulation, inheritance, and polymorphism. In selecting the language for designing the simulation program, the following factors were considered. First, the programming language must perform complex mathematical operations quickly and easily. Second, the language must be widely accepted within the systems engineering environment at Raytheon. Third, the language needed to have built in functions and data types to handle signal processing, matrix and vector operations, and complex numbers. Finally the language must provide the ability to easily generate plots, images, and movies of the target. In Table 1, the ability of some common OOP languages to meet these criteria is provided.

Table 1. Comparison of Object Oriented Programming Languages.

Programming Language Features	C++	Java	MATLAB
Encapsulation	Yes	Yes	Yes
Single Inheritance	Yes	Yes	Yes
Multiple Inheritance	Yes	No	Yes
Pure Virtual Functions	Yes	No	No
Operator Overloading	Yes	No	Yes
Function Overloading	Yes	No	Yes
Templates	Yes	No	No

I/O Capability	Yes	Yes	Yes
Relative Execution Speed	Fast	Slow	Slow/Fast
Built in Mathematical Library	Third Party	Third Party	Yes
Built in Signal Processing Library	Third Party	Third Party	Yes
Built in Plot and Graphing Capability	Third Party	Third Party	Yes
Matrix and Vector Support	Third Party	Third Party	Yes

Of all the OOP languages listed above, MATLAB seemed to be the best fit for creating the simulation program. It is specifically tailored to handle complicated algorithms, with a large library of mathematical functions and tools. Its signal processing toolbox provides the algorithms needed to simulate and analyze the radar returns from the target model. MATLAB also provides built in data types for handling complex numbers, vectors, and matrixes, very useful features to have when dealing with the ray tracing, geometrical transformations, and vector manipulations that will be required. MATLAB is also widely used by systems engineers and is historically the language of choice for creating mathematical simulations. Finally, MATLAB contains a broad array of tools for plotting graphs, displaying images, and even creating AVI movies, all useful features to have for debugging, verifying, validating, and presenting the simulation program.

While MATLAB meets the stated requirements, there are some risks and concerns. First, of all, MATLAB is an interpreted language, meaning the source code is fed into the MATLAB interpreter which carries out the specified actions. This is slow compared to compiled code. With compiled programs, the source code is first translated into the machine code equivalent before it is executed. If speed is a critical requirement to the simulation program, a compiled language like C++ is probably a better choice.

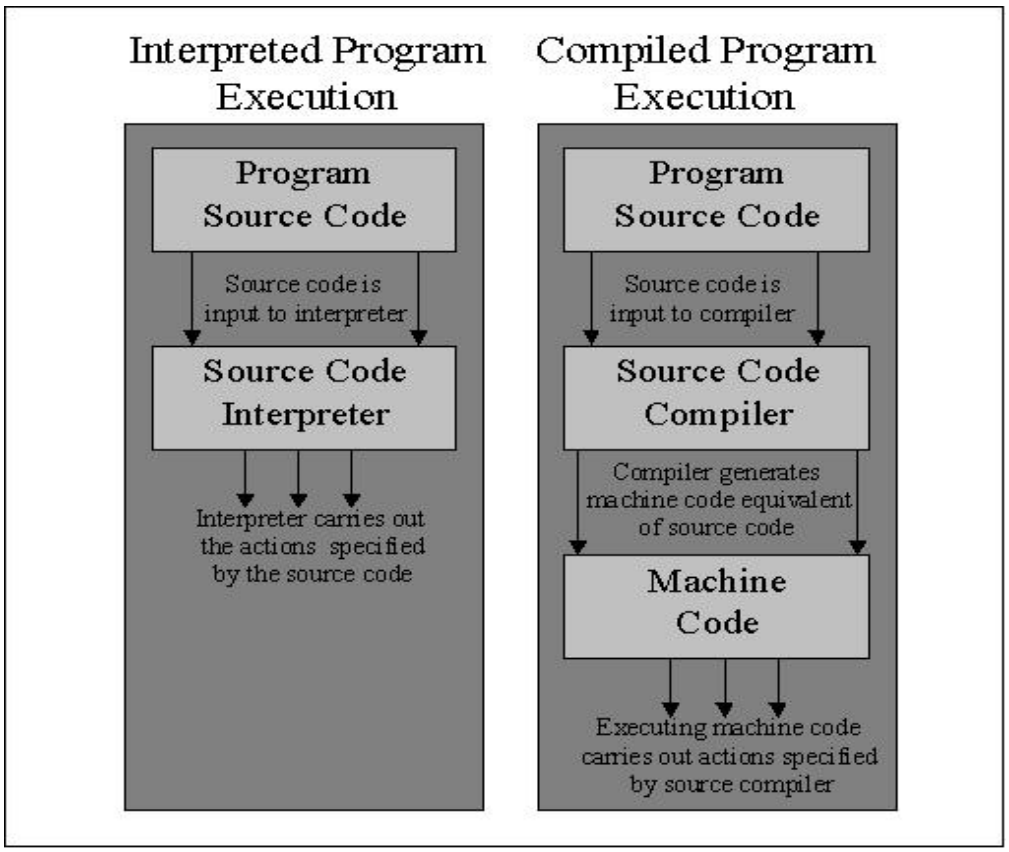

Figure 4. Interpreted vs. Compiled Programs (Horton, 1998).

Another concern with MATLAB is its ability to conform to OOP methodologies. Typically, MATLAB is used as a structured programming language. It was not originally capable of doing encapsulation, inheritance, and polymorphism. These features were only recently added to MATLAB, and whether or not Mathworks was successful at making MATLAB OO capable is a legitimate concern. One of the primary goals of this thesis is to explore how well MATLAB is capable of supporting OOP. In the conclusion of this report, the successes and/or failures of MATLAB as an OOP language is evaluated.

2.3 Object Oriented Software Engineering

While the concepts of encapsulation, inheritance, polymorphism, and composition make OOP a powerful, versatile, and useful tool for developing simulations, without the proper infrastructure, techniques, and philosophies in place, OOP does not deliver upon the promises of improved

maintainability, usability, and affordability. Indeed, poorly designed object oriented code is less desirable to have than structured (or even structured) code due to the added complexity. A parallel may be drawn to that of a racing car vs. an economy car. While a poorly designed economy car is not safe to drive, a poorly engineered racing car is even less desirable due to the higher speeds and more demanding conditions it is designed to operate under.

When designing software, regardless of whether OO or structured programming is used, it is critical that the software is engineered and designed properly *before* coding is begun. Contrary to popular belief, the majority of time spent in developing software should not be spent in the coding phase, but rather in the design phase. By employing proper software engineering techniques before any coding is performed, the impact on the cost of maintenance, debug, and upgrades is greatly reduced. It is critical that the requirements, risks, and tradeoffs are fully understood to mitigate future problems, bottlenecks, or misunderstandings. Since at the design phase, the foundation for the software is established and most of the parameters are set, it is important that sufficient time and analysis be spent in order for the subsequent coding and maintenance to flow as smoothly as possible (Dogru, 2003).

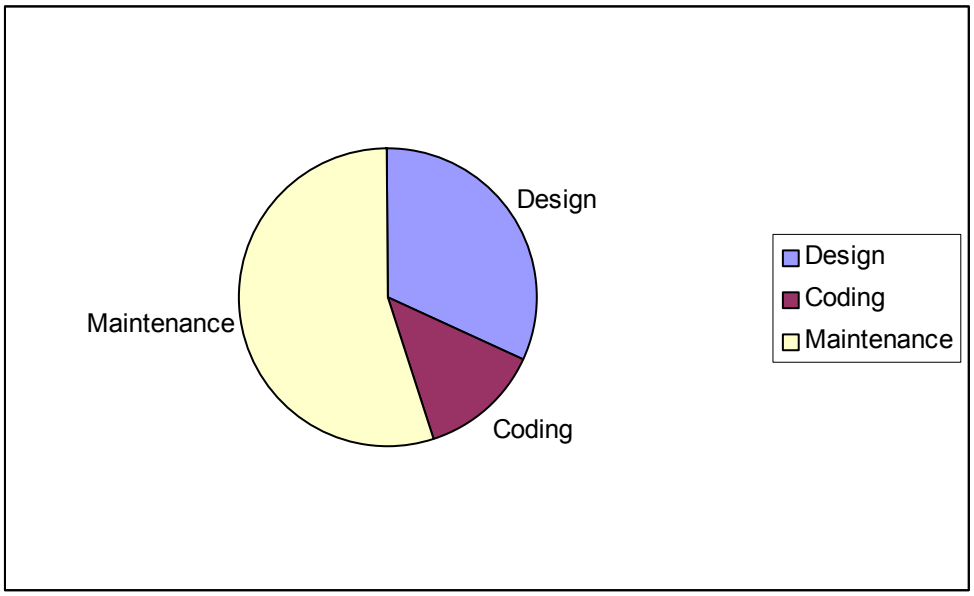

Figure 5. Effort Required for Different Development Activities (Dogru, 2003).

In the following sections, various methods of OO software engineering are introduced and applied to the creation of the simulation. Since the simulation is scientific and mathematical in nature, the techniques and methods are tailored to address the algorithmic concepts required.

2.3.1 Use Case Analysis

Use case diagrams provide a graphical description of the system definition for the customer. While use case diagrams have the capability of providing procedural level details, they are kept at a higher level so as to only convey the highest level functionalities or capabilities of the system. In the diagram, entities external to the system are graphically represented by external actors. System capabilities are graphically represented by ovals, with each oval corresponding to a different system functionality under the capability. The interactions between the actors and the system functionalities are represented within the use-case diagrams by connecting the various actors with the various system functionalities. Specific details related to the use case need not be addressed or explained in the use-case diagram as they are addressed in more specific "interaction diagrams" such as collaboration and sequence diagrams (Dogru, 2003).

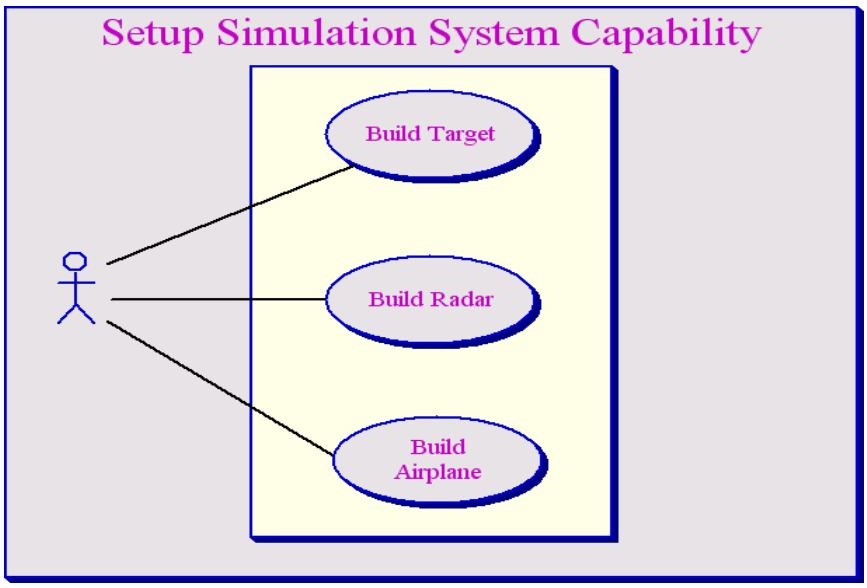

Figure 6. Setup Simulation Use Case Diagram.

The use case diagrams created for the simulation program are shown in Figure 6 and Figure 7. Figure 6 shows the capability of the simulation to build the radar, airplane, and target. Figure 7 shows the capability of the simulation program to create IQ data (how the radar sees the target), create A & B scans, and create a movie of the target as it is in motion. For each of these capabilities, there is an external interface between the simulation and an external actor, the "systems analyst", who serves not only to provide inputs to the simulation, but also to evaluate its output results.

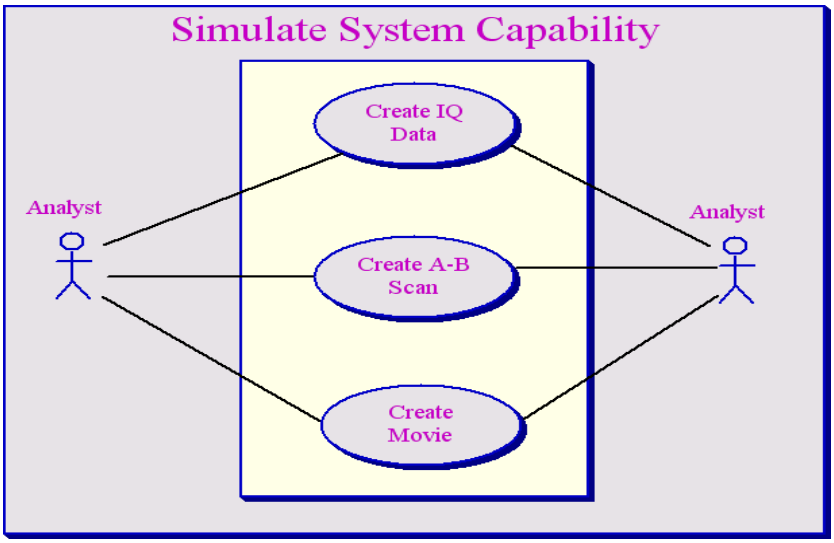

Figure 7. Simulate Use Case Diagram.

2.3.2 Interaction Diagrams

Interaction diagrams map out the dynamic interactions between objects. They are meant to convey the run-time behavior of objects, establishing the order in which messages are passed back and forth. In real-time software, interaction diagrams are useful for establishing timing constraints. For the simulation program, interaction diagrams were useful in establishing the order of events required to correctly simulate the real world behavior. Interaction diagrams may be conveyed using either sequence or collaboration diagrams.

For collaboration diagrams the various objects are represented as nodes. Arrows are drawn from one node to another, representing the flow of information. Attached to each arrow is the name of the

event or method that is triggered by the calling object (on the object being called). It should be noted that drawing an arrow from object A to object B represents object A accessing one of object B's "methods". For example, in Figure 8, an arrow titled "Pick-Up" is drawn from the caller to the phone object, representing the caller using the phone's "pick-up" method (which is inherent to the phone class). Arrows within a collaboration diagram are numbered to convey the order in which events occur. In this way, the sequence of events may be traced throughout time (Dogru, 2003).

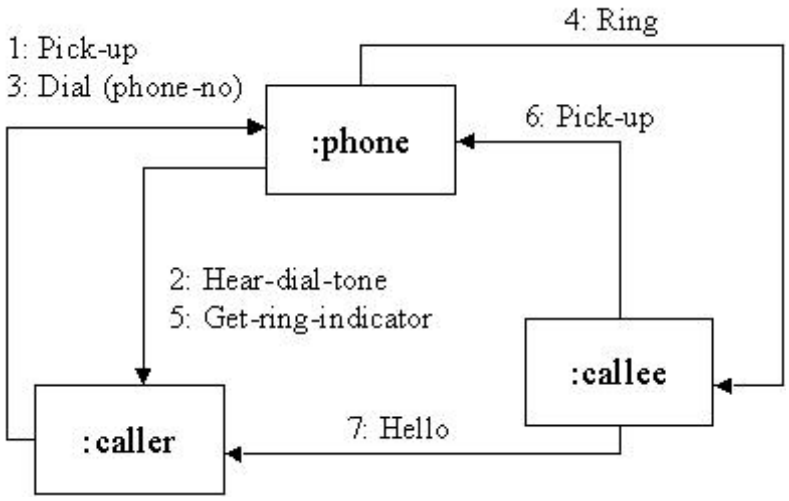

Figure 8. Example Collaboration Diagram. (Dogru, 2003).

Sequence diagrams convey the same information as collaboration diagrams in a slightly different manner. For sequence diagrams, all relevant objects are listed horizontally across the top of the diagrams. Vertical lines are dropped from each object. Messages sent between objects are drawn as horizontal arrows pointing from the sending object's line to the receiving object's line. The order of events progresses vertically, meaning event timing sequences can be determined by "reading" the diagram from top to bottom (Dogru, 2003).

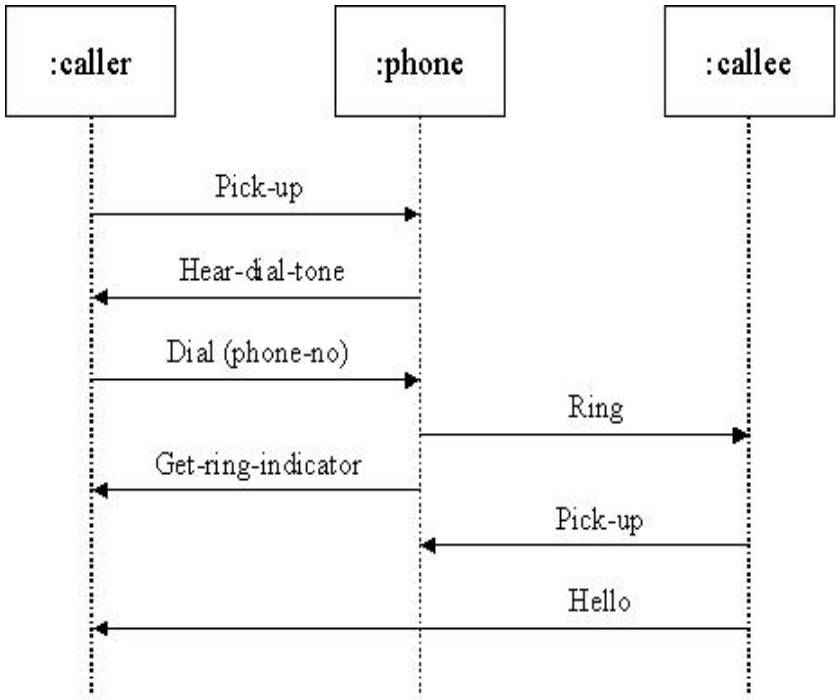

Figure 9. Example Sequence Diagram. (Dogru, 2003).

One of the sequence diagrams created for the simulation program is shown in Figure 10. It illustrates the methods that are invoked and the various interactions required to build targets from a library of shapes. From the diagram, to build a target the user first creates a shape, which has coordinates, normal vectors, surface areas, and a position associated with it. The shape is rotated and shifted to the desired location and added to the target. The target, and the shapes that compose it, can then be rotated as a whole to position the target for the start of the simulation run.

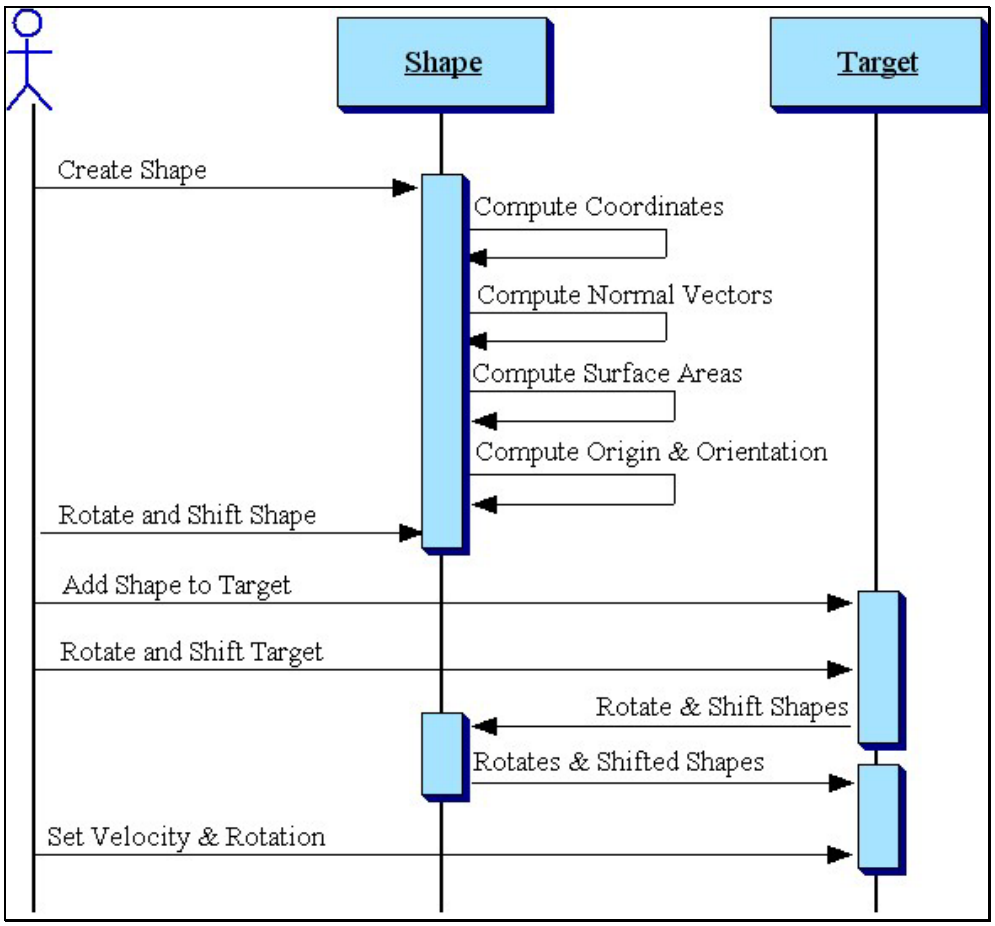

Figure 10. Sequence Diagram for Creating a Target.

2.3.3 Class Diagrams

The class is the fundamental structural unit used in the development of object-oriented software. The class diagram is the primary tool used for establishing class relationships, structures, and interfaces. In the diagram, each class is represented graphically by a rectangular box. Each rectangular box is divided into an upper and a lower section. The upper section lists the properties inherent to the class. The lower section documents the methods used to establish an interface to the class and to instruct the class to perform some service or action (Dogru, 2003).

Different class relationships such as inheritance, association, and composition are represented through graphical interconnections between the various rectangles in the diagram. Inheritance is typically shown by drawing a line from the parent class to the child class, with the parent class having a small

triangle present at its connection point. For composition relationships, the connecting line drawn between rectangles is terminated by a diamond at the composing class's connection point. For association relationships, a line is drawn between the two associated classes. The associative relationship may also have a direction, name, pluralities, and/or roles assigned to it (Dogru, 2003).

The key classes created for the simulation include those shown in Figure 11. The target class defines high level properties of the target, including its motion and an array of geometric shapes that comprise it. The target interface contains methods that set the target's motion, add and remove shapes, and get or set various data about the target's position. The shapes that compose the target are themselves objects. Each shape contains properties such as an array of coordinates, normal vectors, and surface areas. Methods associated with shapes include draw, rotate, and shift. Various methods exist to compute normal vectors, surface areas, coordinates, intersection points, and specular reflection for various scatter points on a shape. Many of these methods and properties are inherited by children including spheres, frustums, facet's, cylinders, and circular discs. These children overload methods defined by their parent shape class based on the type of shape they represent.

To create video phase history, the "IQ generator" class interacts with the target, airplane, and radar. The airplane object defines the relative motion of the aircraft (which carries the radar) relative to the target. The radar object defines the unique characteristics of the radar such as its wavelength, PRF, and pixel spacing. Once the IQ generator creates video phase history, the video processor uses the data to generate A-Scans or range-Doppler maps of the target. These images can be fed to the movie object to create movies of the target as it is seen by the radar over time.

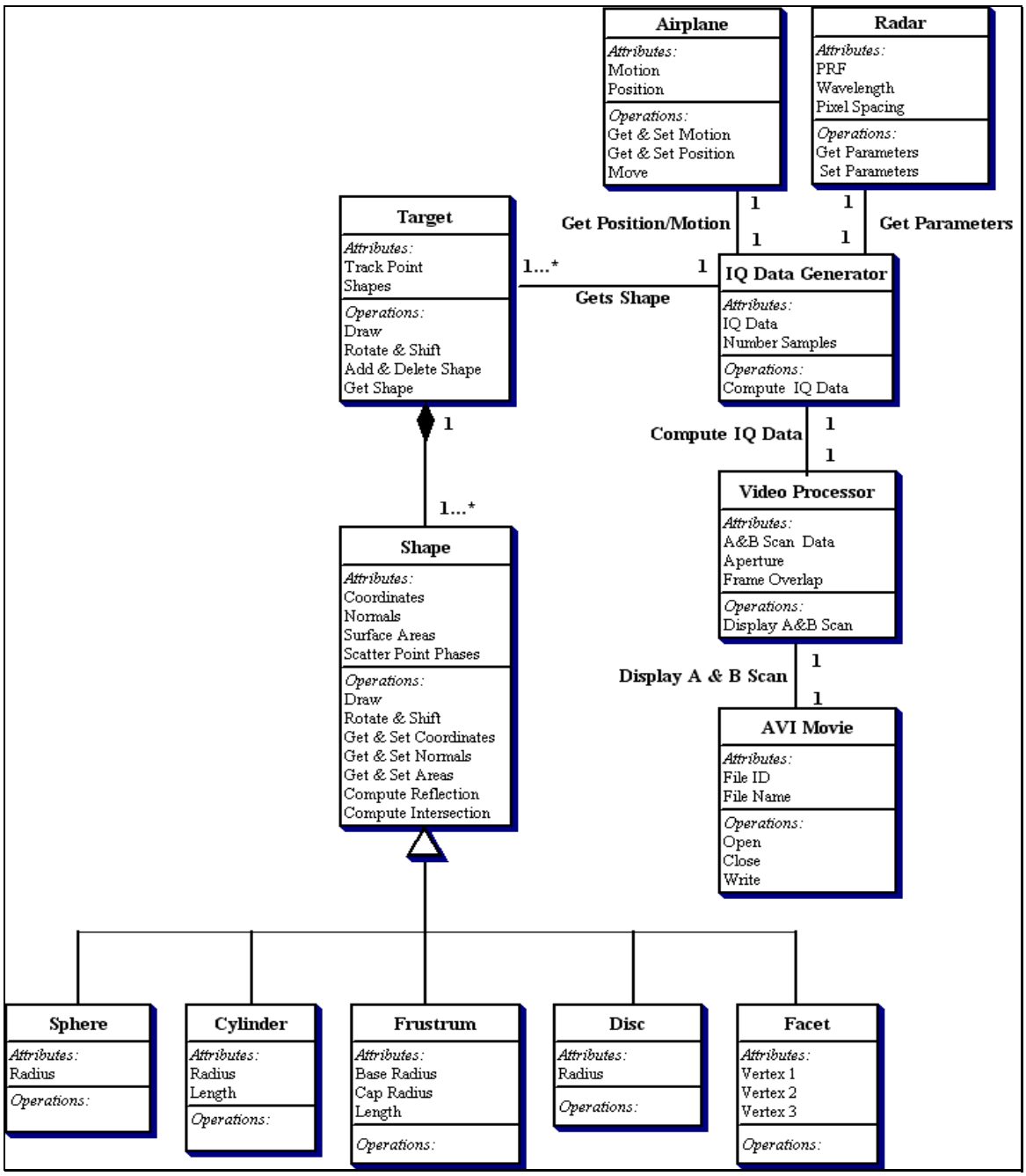

Figure 11. Class Diagram for Simulation.

CHAPTER III
THE BACKSCATTERING BEHAVIOR OF TARGETS

Before development of simulation algorithms can be started, a general understanding of the backscattering behavior of targets must be achieved. The following sections provide a brief overview of how a target reflects radar energy.

3.1 General Scatterer Classes

Man-made targets such as missiles, ships, airplanes, and ground vehicles consist of a wide array of different scatters with a large variety of backscattering behaviors. Extensive work has been done over the past few decades trying to model this behavior; usually by assuming the target consists of a set of idealized features or "geometric primitives" such as facets, spheres, cylinders, and other distinguishing shapes. A sampling of the different type of wave scattering that occurs is shown in Figure 12 and Figure 13. A brief discussion of the various phenomena is provided in the following subsections.

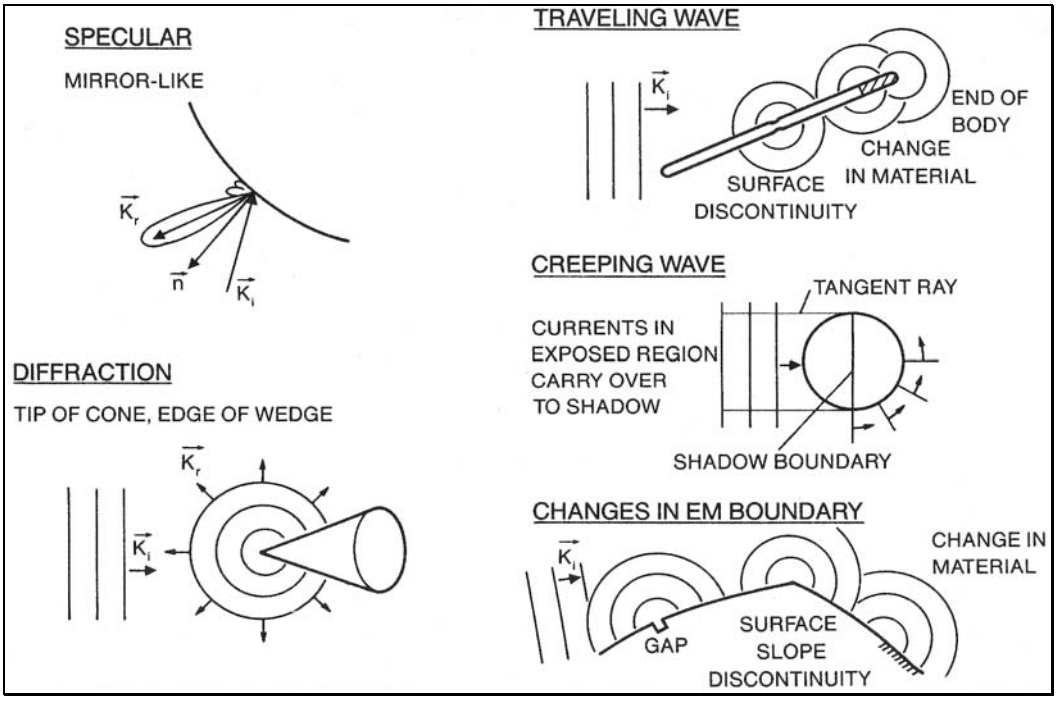

Figure 12. Different Type of Electromagnetic Wave Scattering (Kim, 2003).

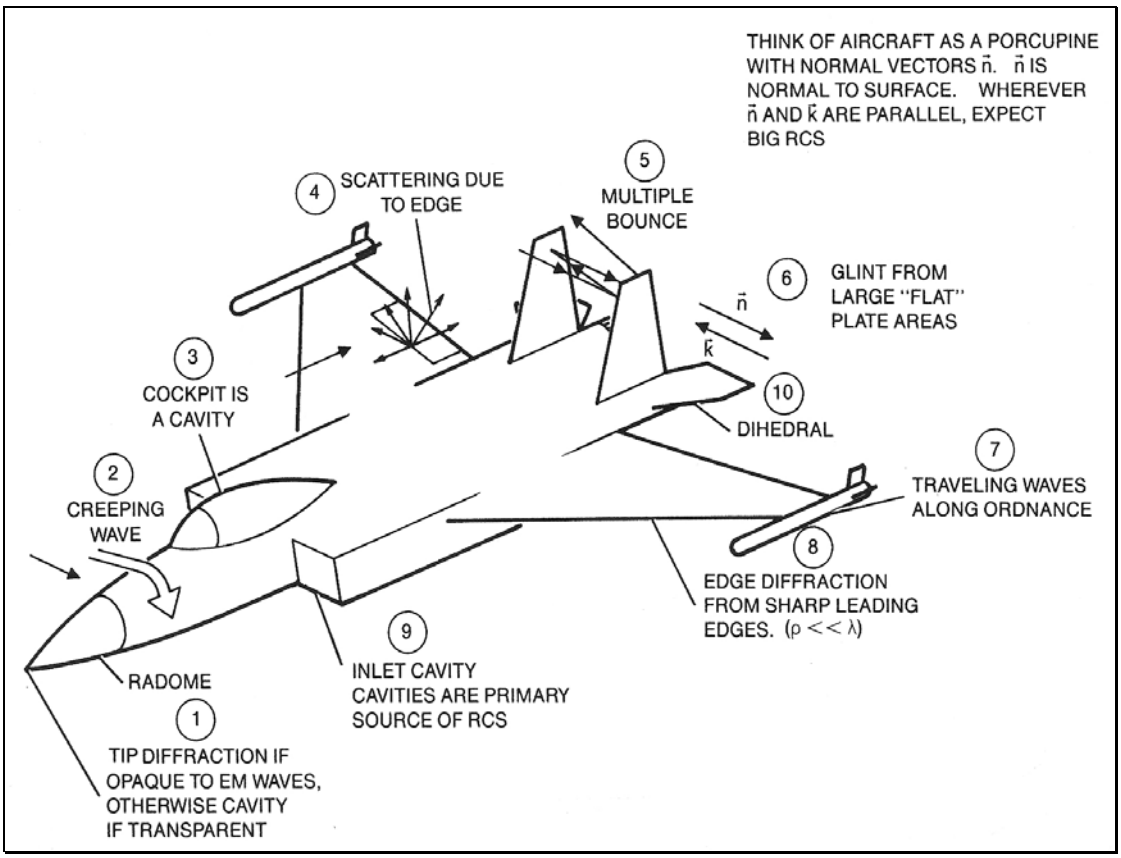

Figure 13. Contributors to Radar Cross Section (Kim, 2003).

For the purposes of this report, and in the design of the simulation, the only scattering phenomena that will be mathematically explored is that of specular scattering. More extensive treatment of the other phenomena is available in numerous other articles and books and is outside the scope of this report. Having limited the simulation to specular scattering, however, does not mean the simulation cannot be adapted and modified to account for other phenomena. On the contrary, by architecting a strong framework for the simulation program using object-oriented techniques, it is well suited for future growth to accommodate other types of scattering phenomena.

3.1.1 Scattering due to Discontinuities

Discontinuities refers to those scatters whose effective extents are relatively small in terms of the radar's wavelength so that they essentially act as fixed point scatters. Some examples of discontinuities on targets include a bolt on the bulkhead of a ship or the edge of a flat plate on a missile fin. In general,

discontinuities may be treated as a point scatterer, meaning its return will be well focused within the range and Doppler bins corresponding to its actual physical location. Another common property of discontinuities is that their returns are relatively weak in comparison to other features from a target. Because their returns are typically 20dB or 30dB below those from stronger target features, they are difficult to detect in target images. It is safe to assume that if one were to inspect a SAR image of a ground vehicle, that most likely none of the responses would come from discontinuities (Rihaczek, 1996).

3.1.2 Scattering due to Smooth Surfaces

Smooth surfaces refer to structures such as flat plates on the deck of an aircraft carrier or the rounded fuselage on an aircraft. In the case of the flat plate, a huge return, commonly referred to as a specular flash, occurs when the plate is oriented at its broadside aspect. The primary concern with specular flashes is that they can be so strong as to drown out other features of the target. In reality, however, specular flashes seldom occur since they are only generated within a small angular sector about perpendicular incidence (Rihaczek, 1996).

Specular flashes are typically a concern when dealing with rounded surface. Unlike the flat plate, with curved surfaces, as the aspect angle changes the effective point of return shifts along the surface in order to maintain perpendicular incidence. Specular flashes from curved surfaces, however, are generally less severe than flat plates since the curvature of the surface restricts the surface area that effectively generates the flash. In summary, smooth surfaces do exhibit the problem of specular flash, however they are observed rarely enough as to not pose a serious threat (Rihaczek, 1996).

3.1.3 Scattering due to Corners

Corner reflectors refer to shapes that are similar to the trihedral, whose three sides are perpendicular to one another. A trihedral acts as a triple bounce reflector, with the unique property that the effective phase center is at the corner point. This means that the corner reflector acts as a strong, fixed point reflector, whose strength approaches that from a flat plate at perpendicular incidence. In reality, shapes in man-made targets often due not conform to that of the ideal trihedral. In ground vehicles,

dihedral corners often act as double bounce reflectors; however, their returns are generally weak save for an incidence perpendicular to the axis of the dihedral. In reality, corner reflectors on targets are a poor approximation and can only be interpreted as irregular corners that trap incoming waves in a complicated manner that cannot be described by multiple-bounce reflections (Rihaczek, 1996).

3.1.4 Scattering due to Regular Cavities

Cavity type reflectors are prevalent in man-made targets. Examples of cavities include the ducts within the engine exhaust of an airplane and the cup shaped wheel of an armored vehicle. Ray-tracing techniques, which are introduced in this paper, can provide a feel for the multiple bounces that occur within the cavity. The combined return from the cavity should contain a band of Doppler frequencies as defined by the width of the cavity. Since the cavity has depth, range delays will also be introduced to the various components of the radar return. In general, the return from a cavity will be spread in range and Doppler depending on the size of the cavity and the aspect angle under which it is viewed (Rihaczek, 1996). While multiple bounce responses due to cavities are not addressed in this paper. The simulation foundation, with its ability to perform elementary ray tracing, can be easily expanded to extend the ray tracing to occur over multiple bounces off the target.

Another class of cavities that should be mentioned are those that are irregular. These types of cavities trap the wave before backscattering it, yet are of such complicated design that the backscattering cannot be described by multiple bounces. Unfortunately, this type of scattering can occur quite often in targets and cannot be adequately analyzed using ray tracing techniques. An example of such an irregular cavity includes the overhang of the turret on the deck of a tank. These irregular cavities pose problems for target identification, due to the fact that the phase mismatches introduced by the dispersive properties of the cavity are often so large as to produce spurious responses that can fall anywhere within or even outside of the vehicle boundaries (Rihaczek, 1996).

3.1.5 Traveling Wave Returns

For a large portion of a target, the radar wave runs along the surface of some feature and is reflected at the features end-point. This class of radar returns is often referred to as traveling wave response. As an example, consider a rocket engine exhaust. The radar wave enters the duct, and neglecting the other forms of scattering mentioned so far, travels along the inside of the exhaust and is reflected at its end. Part of the wave may again be reflected back along the exhaust and be backscattered at its point of entry. The wave may be reflected and backscattered numerous times before its energy is exhausted and too weak to be detected (Rihaczek, 1996).

One interesting phenomena of traveling waves is that they can illuminate features that appear to be shadowed optically. For example, features on the front of a ship's deck that are shadowed optically by the mast, may become illuminated due to traveling wave returns. (Rihaczek, 1996)

3.1.6 Specular Scattering

The scattering phenomena addressed in this paper and implemented in the simulation is specular scattering. Specular scattering depends on the position of the electromagnetic waves in relation to the target's surface normal vectors. In specular scattering, the angle of reflection of a ray off of a surface is equal to the angle of incidence, similar to how light is reflected off a mirror. This suggests that the amount of specular highlight observed by the radar is dependent upon the angle between the line of sight vector and the reflection of the electromagnetic energy off the target's surface. In determining the specular scattering it is crucial to understand the target's geometry to determine the surface normal vectors and angles of reflection. As discussed later, the requirement for understanding the geometry of "scatter points" within the target played a major role in determining to leverage OOP techniques to design the simulation.

3.2 Shadowing of Target Features

In modeling the radar signature of a target, it is important to account for any "shadowing" that might occur. A scatter point may be shadowed when the radar pulse is prevented from reaching it

because the scatter point is obstructed by some target feature. For example, the ship's superstructure may shadow a portion of the ship's deck from view. Similarly, depending upon the plane's aspect angle, portions of the fuselage might be shielded from view by its wing.

It is interesting to compare and contrast the approaches that might be used to determine target shadowing. With structured design, the target most likely consists of a collection of independent scatter points as shown in Figure 14. To determine if a ray drawn from the radar to a scatter point is shadowed by any other portion of the target, *all* scatter points on the target must be checked to determine if the ray intersects them first. In addition, since each scatter point represents some amount of surface area, it must be determined if the ray passes through the immediate *area* represented by the scatter point. These computations are mathematically difficult and computationally very expensive to perform.

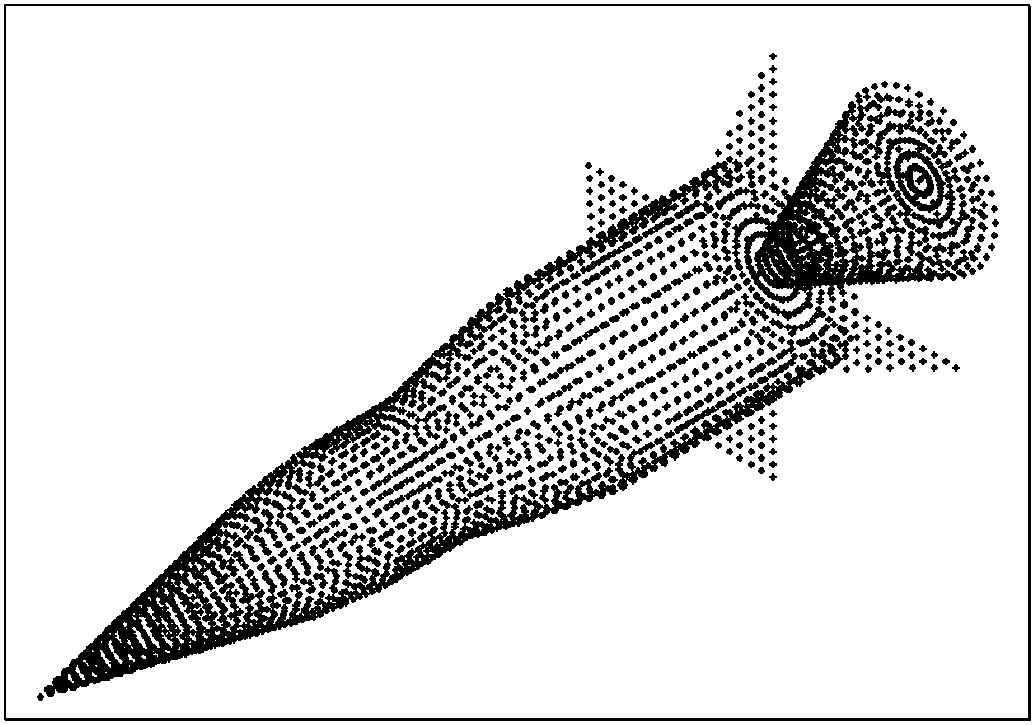

Figure 14. Treating the Target as a Collection of Independent Points.

A better solution is offered using object oriented analysis and design. With OOP, the target may be treated as a collection of geometric objects such as cones, spheres, facets, and frustums. Since the unique geometry of each part of the target is known, ray tracing techniques may be used to determine

whether a particular scatter point is shadowed. To determine if a scatter point is shadowed by shapes within the target, the equation for the “ray” between the scatter point and the radar is plugged into the equations defining each of the various shapes in the target. By solving for and examining the roots of the resulting simultaneous equations, it can be determined whether or not any of those shapes shadow the scatter point from the radar's view. Clearly, by treating the target as a collection of geometric shape as opposed to individual scatter points, object oriented techniques reduce the complexity and computations required to determine target shadowing.

CHAPTER IV
BUILDING THE SHAPE LIBRARY

The library of geometric shapes forms the foundation for the target model used in the simulation. As discussed earlier, OOP provides an excellent framework from which to build and grow the shape library. The library consists of several different unique shapes derived from a general shape class. Each shape inherits the ability to draw itself, compute its scatter point coordinates, compute its reflectivity, determine whether or not it intersects a radar ray, and rotate or orient itself at a specific position. While developing this framework is in and of itself a big step, actually coming up with the algorithms and formulas to perform these tasks is another story. For different types of shapes, different algorithms are required. With the inheritance and polymorphism capabilities of OO, however, adding new shapes, with their unique algorithms and properties, to the shape library is a relatively painless procedure.

In the following sections, an introduction is provided to some basic techniques and algorithms available for use with different types of shapes. With a firm understanding of these basic algorithms, the initial library of five shapes discussed in this paper can easily be expanded to contain additional unique and interesting shapes.

4.1 Computing Surface Areas

Computing the surface area for scatter points on objects such as cones, frustums, spheres, or other more complicated shapes poses an interesting problem. Using the cone as an example, assume the scatter points are placed around the cone's body using a technique where the cone is divided into X uniformly spaced "rings", with each ring containing Y scatter points. Since the circumference of the rings decreases from the base to the tip of the cone, and the number of scatter points per ring is constant, the scatter points near the cone's tip are more densely spaced than those near the cone's base. This difference in surface area must be accounted for to correctly compute the radar energy reflected from each scatter point.

One possible solution to the problem of differently sized scatter points is to change the sampling scheme such that all scatter points are uniformly sized. This solution, however, has its shortcomings. Assume that each scatter point is set to a size that optimally describes the curvature of the cone near its

base. Near the cone's tip, however, there is less surface area available to pack the scatter points into. This means that there are fewer scatter points available near the tip of the cone to adequately describe its curvature. To solve this problem, one might be tempted to size each scatter point small enough so it adequately describes the cone's curvature near its tip. This, however, results in excessively small scatter points, that when uniformly placed around the cone's base, result in an excessive number of scatter points. The "extra" scatter points result in wasted execution time, where execution time is a precious resource given the high number of calculations required.

Given the problems of uniformly sized scatter points, the sampling method where each scatter point is sized according to its location within the shape is preferred. The surface area associated with the various scatter points may be computed several different ways. For well known shapes like cones, frustums, or spheres, there are canned surface area formulas that may be "adapted" to find the area that each scatter point represents. It would be preferable, however, to have a more generic technique available for computing the surface areas for any given *solid of revolution*. A solid of revolution is formed by rotating some "area" (in the X-Y plane) about the X-axis. For example, by rotating the area described in Figure 16 about the X-axis, the frustum shown in Figure 15 is created. It should be noted that the equation describing the "slant" of a solid of revolution need not be a simple linear equation of $Y = mX + b$. The equation may be a quadratic, sinusoidal, or any other interesting formula, so long as it is expressed in terms of X and Y.

The generic formula for computing the surface area for any given solid of revolution is given by Equation 1. From this equation, the only information required to compute the surface area is the equation and the derivative of the equation describing the solid of revolution's "slant". Once this information is known, the integral may be solved to find the formula for computing the solid of revolution's surface area (between some constraints $X = a$ and $X = b$). If the constraints a and b are taken to be the minimum and maximum X boundary for the various "rings" which constitute the solid of revolution, then the surface area for the various scatter points within a ring may be found by dividing that ring's surface area by the number of scatter points within that ring.

Equation 1:

$$surface\ area = \int_a^b 2\pi y \sqrt{1 + \left(\frac{dy}{dx}\right)^2}\, dx$$

where:

$a = Minimum\ X\ boundary\ for\ solid\ of\ revolution$

$b = Maximum\ X\ boundary\ for\ solid\ of\ revolution$

$y = Equation\ describing\ solid\ of\ revolution's\ slant$

$\frac{dy}{dx} = Derivative\ of\ equation\ describing\ solid\ of\ revolution's\ slant$

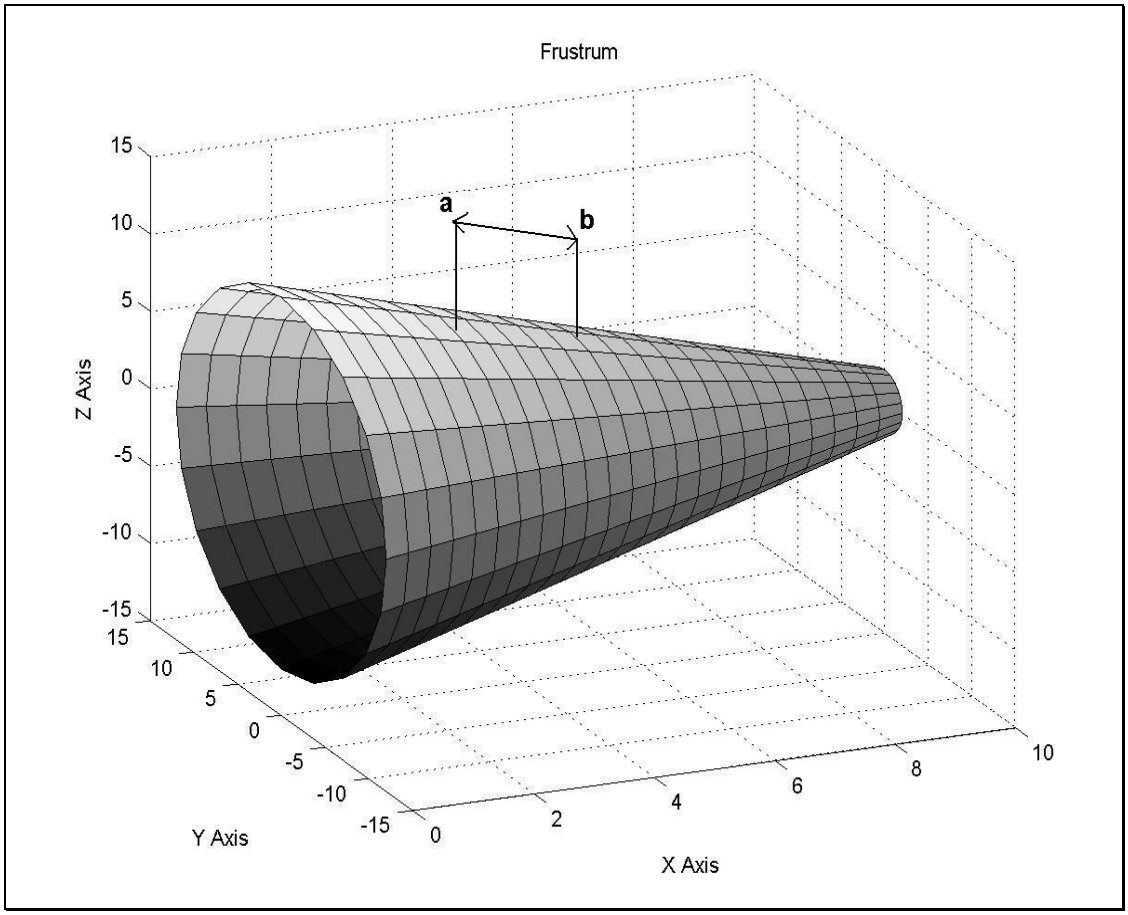

Figure 15. Determining Surface Area of a Frustum Section.

As an example, consider finding the surface area for the various scatter points that constitute the frustum shown in Figure 15. From the two dimensional slice of the frustum (taken in the X, Y plane at Z = 0), it is evident that the equation describing the frustum's slant is given by Equation 2 (with a slope of –1 and Y intercept of +10). Taking the derivative of this equation yields Equation 3 (where m = -1).

Plugging Equation 2 and Equation 3 back into Equation 1 and solving for the integral yields the formula for computing the area of a frustum (between the limits X=a and X=b), as shown by the derivation provided in Equation 4.

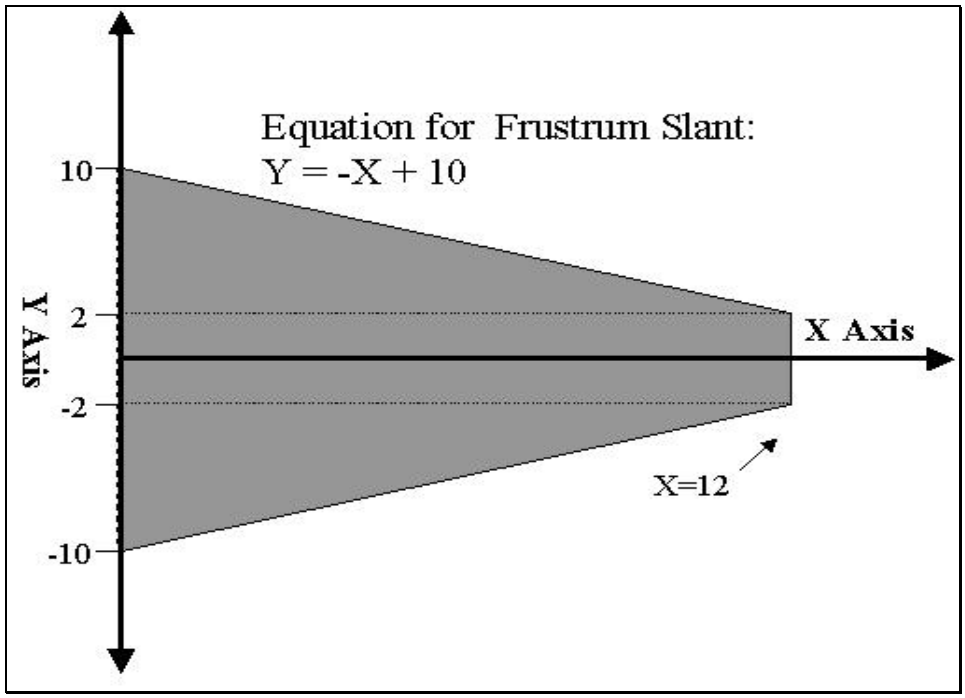

Figure 16. 2D Slice of a Surface of Revolution (Frustum Example).

Equation 2:

$$y = mx + b$$
$$where: \; m = slope$$
$$b = Y\ Intercept$$

Equation 3:

$$\frac{dy}{dx} = m$$

Equation 4:

$$surface\ area = \int_a^b 2\pi \cdot (mx + b) \cdot \sqrt{1 + m^2}\, dx$$
$$surface\ area = 2\pi \cdot \sqrt{1 + m^2} \cdot \int_a^b mx + b \;\; dx$$
$$surface\ area = 2\pi \cdot \sqrt{1 + m^2} \cdot \left[\frac{mx^2}{2} + bx \right]_a^b$$

4.2 Computing Scatter Point Normal Vectors

In computing the radar signature for the various objects that compose the target, the specular scattering off the target's surface must be determined. As mentioned previously, specular scattering depends on the position of the electromagnetic waves in relation to the surface normal vectors of the target's scatter points. In specular scattering, the angle of reflection of a ray off of a surface is equal to the angle of incidence, similar to how light is reflected off of a mirror. This suggests that the amount of specular highlight observed by the radar is dependent upon the angle between the line of sight vector and the reflection of the radar signal off the scatter point. In determining specular scattering, therefore, it is crucial to understand the target geometry to determine the surface normal vectors and angles of reflection for the various scatter points.

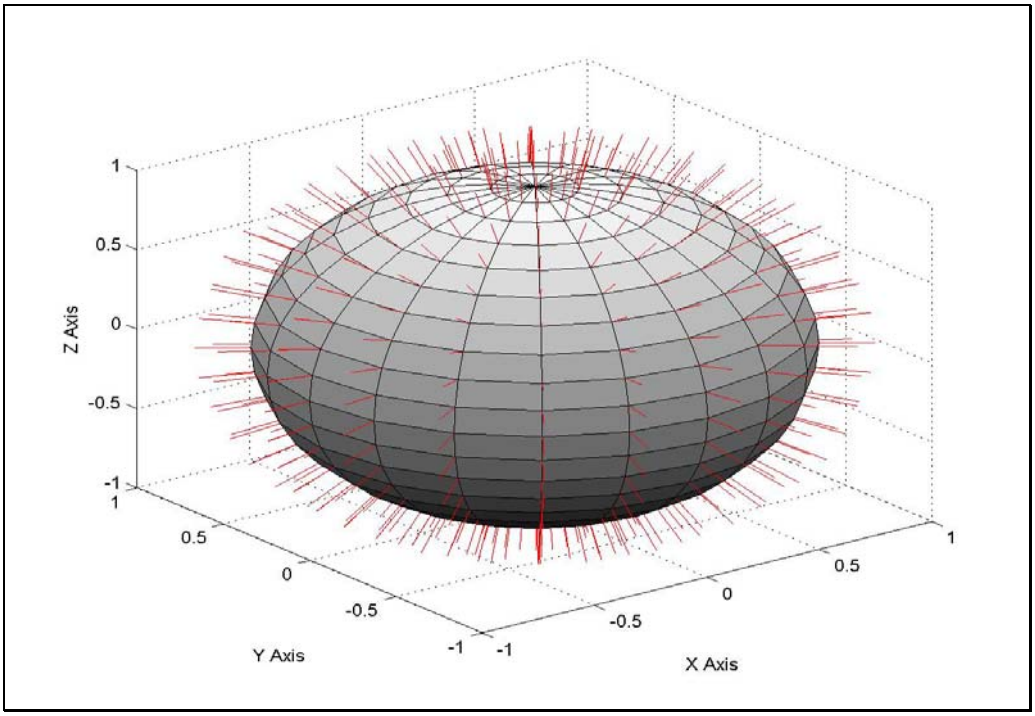

Figure 17. Surface Normal Vectors to a Sphere.

In architecting the simulation program, object-oriented techniques greatly simplifies the computations and logic required to compute surface normal vectors. Without OOP, one might be tempted to design the simulation such that the target consists of individual scatter points as opposed to a collection

of simple shapes. If this were the case, computing the surface normal for a scatter point would require that surrounding scatter points be located, and from their relative positions, the surface normal to the scatter point in question estimated using complicated geometric techniques. While estimating the surface normal is not a difficult computation to perform, finding the surrounding scatter points is mathematically difficult and computationally expensive. Moreover, to accurately estimate the surface normal vectors for a scatter point, the scatter point sampling must be sufficiently dense, since the target geometry most likely varies between scatter points, resulting in the computed normal to a scatter point being slightly off its true value.

A much better solution is to use object-oriented techniques to encapsulate individual scatter points within known shapes. From knowledge of the shape's geometry, one can determine the vector perpendicular to any scatter point contained on the shape's surface. The surface normal vector is determined by finding two vectors tangent to the shape at the scatter point and taking their cross product per Equation 5. The trick is finding two vectors tangent to the shape at the scatter point. For simple shapes like cylinders or cones, tangent vectors can be determined by inspecting the shape's symmetry and formulating the vectors from the knowledge of the shape's geometry. For both simple and complicated shapes, however, the more generic technique of determining the gradient can be used to find the plane tangent to the scatter point and the surface normal vector.

Equation 5:

$$p = u \times v$$

where :

u and v are contained in the plane tangent to the scatter point

The tangent plane to a surface f(x, y, z) = k at a point $P_o = (x_o, y_o, z_o)$ is the plane perpendicular to the gradient vector at P_o. To obtain the formula for the tangent plane, let P = (x, y, z) run over all points in the plane. Then any vector of the form shown in Equation 6 is parallel to the tangent plane, because P = (x, y, z) and $P_o = (x_o, y_o, z_o)$ both lie in the tangent plane, but any vector parallel to the tangent plane must be perpendicular to the normal vector to the plane. This means the vector $(\vec{P} - \vec{P}_o)$ and

$\vec{\nabla} f(x_o, y_o, z_o)$ are perpendicular and their dot product is zero. This relationship can be used to form the relationship described by Equation 7, which can be used to find the equation for the plane tangent to the scatter point.

Equation 6: $\boxed{\vec{P} - \vec{P}_o = (x - x_o, y - y_o, z - z_o)}$

Equation 7: $\boxed{(\vec{P} - \vec{P}_o) \bullet \vec{\nabla} f(x_o, y_o, z_o) = 0}$

As an aide to implementing the gradient technique described above, the following example is provided where the equation of the tangent plane to the point $\vec{p}_o = (1, 2, 3)$ on the surface $3x^2 + y^2 - z^2 = -20$ is found:

$$\vec{\nabla} f = \frac{\partial f}{\partial x}\vec{i} + \frac{\partial f}{\partial y}\vec{j} + \frac{\partial f}{\partial z}\vec{k} = 6x\vec{i} + 2y\vec{j} - 2z\vec{k}$$

$$\vec{\nabla} f(\vec{p}_o) = 6(1)\vec{i} + 2(2)\vec{j} - 2(3)\vec{k} = 6\vec{i} + 4\vec{j} - 6\vec{k}$$

$$\vec{p} - \vec{p}_o = (x-1)\vec{i} + (y-2)\vec{j} + (z-3)\vec{k}$$

The Equation of the tangent plane is therefore:

$$0 = (\vec{p} - \vec{p}_o) \bullet \vec{\nabla} f(\vec{p}_o)$$

$$0 = \left[(x-1)\vec{i} + (y-2)\vec{j} + (z-3)\vec{k}\right] \bullet \left[6\vec{i} + 4\vec{j} - 6\vec{k}\right]$$

$$0 = 6(x-1) + 4(y-2) - 6(z-3)$$

$$0 = 6x - 6 + 4y - 8 - 6z + 18$$

$$0 = 6x + 4y - 6z + 4$$

One should note that in taking the cross product of two vectors, the order in which the cross product is done effects the direction of the resulting vector according to the "right hand rule". The right hand rule states: Let θ be the angle between vector U and V, and suppose U is rotated through the angle θ until in coincides with V. If the fingers of the right hand are cupped so they point in the direction of rotation, then the thumb roughly indicated the direction of U cross V. This means that in computing the surface normal vectors to a shape like a sphere or cylinder, care must be taken in the ordering of the cross product so the resulting normal vector points in the correct direction.

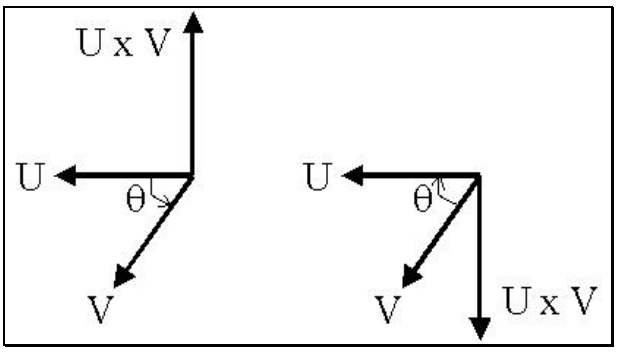

Figure 18. Right Hand Rule.

It should be noted that whenever a shape is moved or rotated the surface normal vectors must be recomputed. Since the surface normal vectors are already known, however, they do not need to be recomputed using the numerically expensive cross product technique described above. Instead, they can be recomputed by simply performing the same geometric transformations to the normal vectors as were performed to the shape itself.

4.3 Computing Scatter Point Specular Reflection

Once the surface normal vectors for a shape are known, all that remains to determine the specular reflectivity for a scatter point is to compute the angle between the scatter point's surface normal vector and the electromagnetic energy from the radar. The radar wave can be represented as a vector by taking the coordinates of the radar and subtracting off the location of the scatter point per Equation 8. This vector can then be converted to a unit vector per Equation 9. From Equation 10, the angle between the surface normal vector and the radar wave is determined. If the resulting angle is between 0 and 90 degrees, the scatter point is exposed to the radar. If the resulting angle is between 0 and –90 degrees, the scatter point faces away from the radar and is shadowed (reflecting no energy back to the radar).

Equation 8:

$$V_{radar\ ray} = P_{radar} - P_{scatter\ point}$$

where:

P_{radar} *is position of radar*

$P_{scatter\ point}$ *is position of scatter point*

Equation 9: $$U_{radar\ ray} = \frac{V_{radar\ ray}}{\left\| V_{radar\ ray} \right\|}$$

Equation 10: $$cos(\theta) = \frac{U_{radar\ ray} \bullet N_{scatter\ point}}{\left\| U_{radar\ ray} \right\| \cdot \left\| N_{scatter\ point} \right\|}$$

4.4 Ray Tracing Techniques

In order to determine the radar signature from a complex target, one needs to understand the interactions and interference between the various scatter points that compose the different target sections. For example, in modeling the radar signature from an airplane, the wings and/or the fins may shadow part of the fuselage. In modeling the radar cross-section of a ship, the mast and superstructure shadows portions of the deck from the radar's view. Determining which scatter points are shadowed by other target features poses an interesting problem. If all of the scatter points that compose the target are treated independently, it is difficult to determine if a "ray" from the radar intersects one scatter point before it reaches another. In addition, since each scatter point represents some surface area, how does one determine if the ray intersects the small area which each point represents? It is a very expensive operation to cycle through all the scatter points on a target, checking whether each one blocks the scatter point from the radar's view.

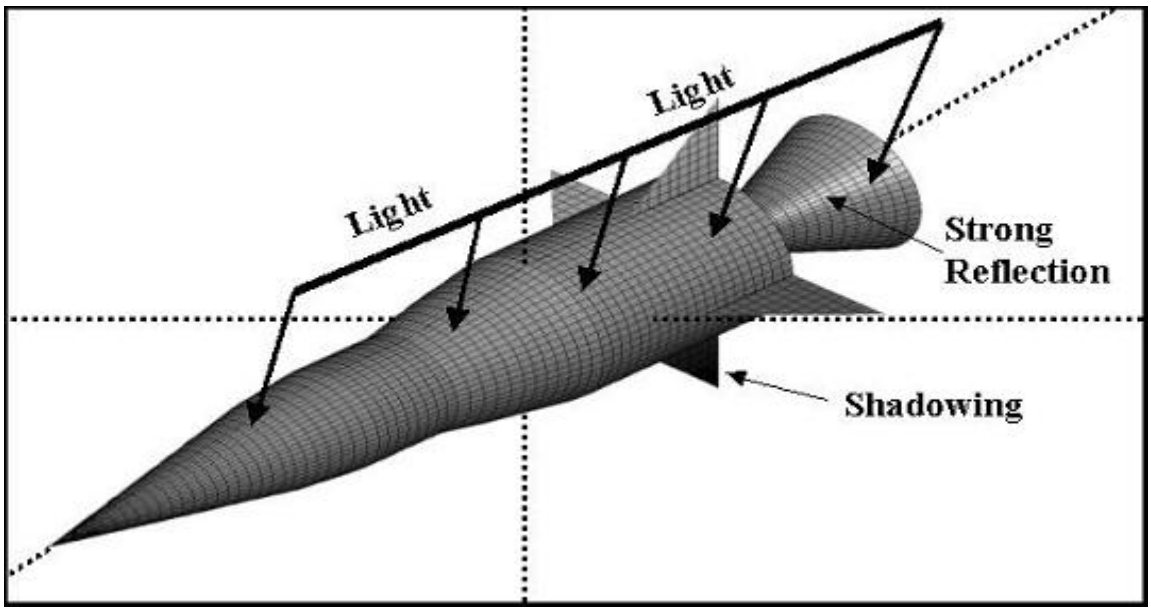

Figure 19. Shadowing Caused by Target Features.

With object-oriented design, the problem of determining if a scatter point is obscured by another portion of the target has a simple and elegant solution. Under an OO framework, each scatter point is encapsulated within a shape object. That shape may be a cone, facet, sphere, cylinder, or other basic shape. Instead of checking all scatter points to see if they cause a scatter point to be obscured, it is much more efficient to check if any of the *shapes* that make up the target cause a scatter point to be obscured. The techniques that can be used to determine the shadowing between shapes that make up the target is a technique borrowed from rendering known as ray tracing.

In general, ray tracing is done in the following manner. First, a ray is traced from the radar to the scatter point in question. This ray may be described by a simple mathematical equation. Each simple shape that makes up the target can also be described by a mathematical equation. To check if the ray intersects a shape that composes the target, simply plug the ray equation into the shape equation and solve. By examining the solution, one can easily determine the location(s) of any intersections between the shape and the ray traced from the radar. If an intersection occurs, it must be determined if the intersection point occurs before or after the radar wave reaches the scatter point in question. To

determine this, the range from the radar to the scatter point and intersection point are compared. If the range to the scatter point is greater, then it is shadowed by another part of the target.

While the general procedure for performing ray tracing is fairly straightforward, there are some nuances and tricks that can be used for different types of shapes. The following sections provide a detailed description of the techniques used in the simulation to compute the intersections between rays and a small library of basic shapes.

4.4.1 Ray versus Sphere Intersection

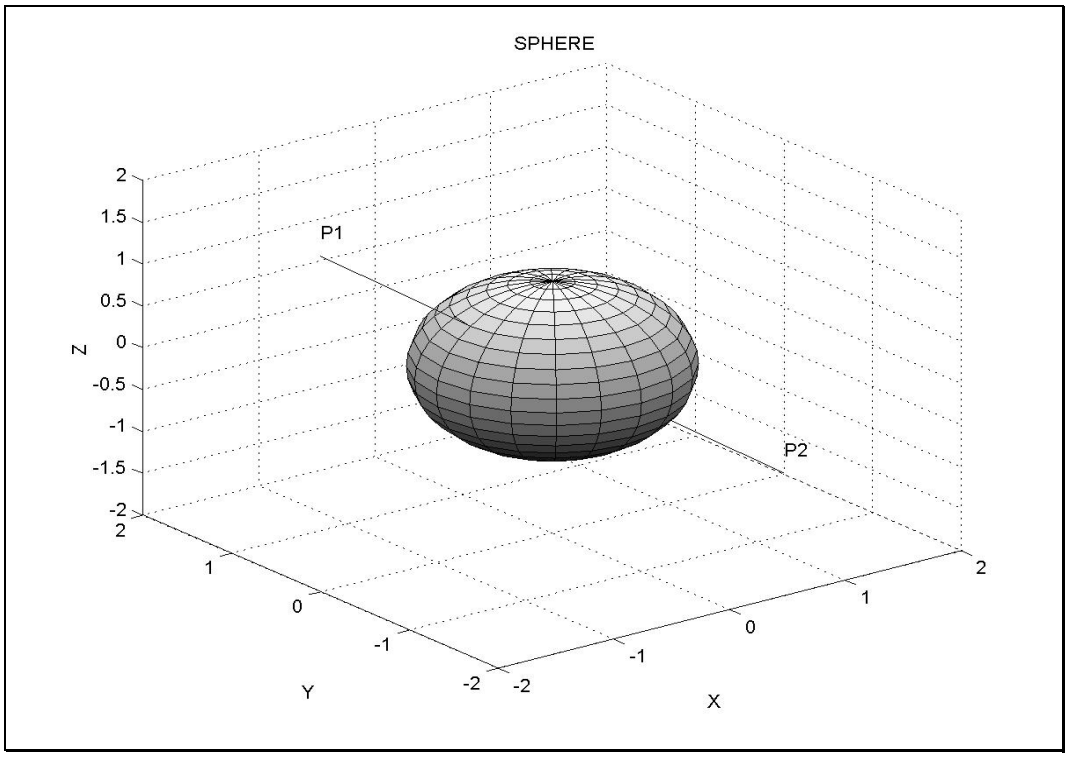

Figure 20. Ray-Sphere Intersection.

To check if a sphere shadows a scatter point on a target, the following algorithm may be used. The ray is defined by two points, $P_1 = (X_1, Y_1, Z_1)$ and $P_2 = (X_2, Y_2, Z_2)$. In the model, P_1 is the location of the target scatter point and P_2 is the location of the radar. The behavior of the ray is mathematically described by Equation 11. The sphere is mathematically described by Equation 12. Note from this description that the sphere is located at its *primitive* location, centered about the origin with a radius R. A graphical illustration of the ray-sphere intersection is provided by Figure 20.

Equation 11:
$$X^2 + Y^2 + Z^2 = R^2$$

Equation 12:
$$\begin{aligned} X &= X_1 + U(X_2 - X_1) \\ Y &= Y_1 + U(Y_2 - Y_1) \\ Z &= Z_1 + U(Z_2 - Z_1) \end{aligned}$$

Equation 13:
$$\begin{aligned} &((Z_2 - Z_1)^2 + (Y_2 - Y_1)^2 + (X_2 - X_1)^2) \cdot U^2 + \\ &2 \cdot (Z_1(Z_2 - Z_1) + Y_1(Y_2 - Y_1) + X_1(X_2 - X_1)) \cdot U + Z_1^{\,2} + Y_1^{\,2} + X_1^{\,2} - R^2 = 0 \end{aligned}$$

Equation 14:
$$\begin{aligned} U &= \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \\ A &= ((Z_2 - Z_1)^2 + (Y_2 - Y_1)^2) + (X_2 - X_1)^2) \\ B &= 2 \cdot (Z_1(Z_2 - Z_1) + Y_1(Y_2 - Y_1) + X_1(X_2 - X_1)) \\ C &= Z_1^{\,2} + Y_1^{\,2} + X_1^{\,2} - R^2 \end{aligned}$$

Substituting the equation for the ray into the equation for a sphere yields a quadratic equation given by Equation 13. The behavior of the ray-sphere intersection can be understood by examining the quadratic formula given by Equation 14:

- If the discriminate is < 0, the ray does not intersect the sphere
- If the discriminate = 0, the ray intersects the sphere at one point
- If the discriminate is > 0, the ray intersects the sphere at two points

If the ray intersects the sphere, the intersection point(s) are found by solving the quadratic formula and substituting the roots back into Equation 12. This gives at most two points of intersection, P_3 and P_4. Equation 29 can then be used to check if the scatter point is shadowed by the sphere. If the range P_1 (from the radar to the scatter point) is less than range P_3 or P_4 (from the radar to the point(s) of intersection on the sphere), then the sphere shadows the scatter point.

Equation 15:
$$\begin{aligned} P_1\ Range &= \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2 + (Z_2 - Z_1)^2} \\ P_{3,4}\ Range &= \sqrt{(X_{3,4} - X_1)^2 + (Y_{3,4} - Y_1)^2 + (Z_{3,4} - Z_1)^2} \end{aligned}$$

4.4.2 Ray versus Cylinder Intersection

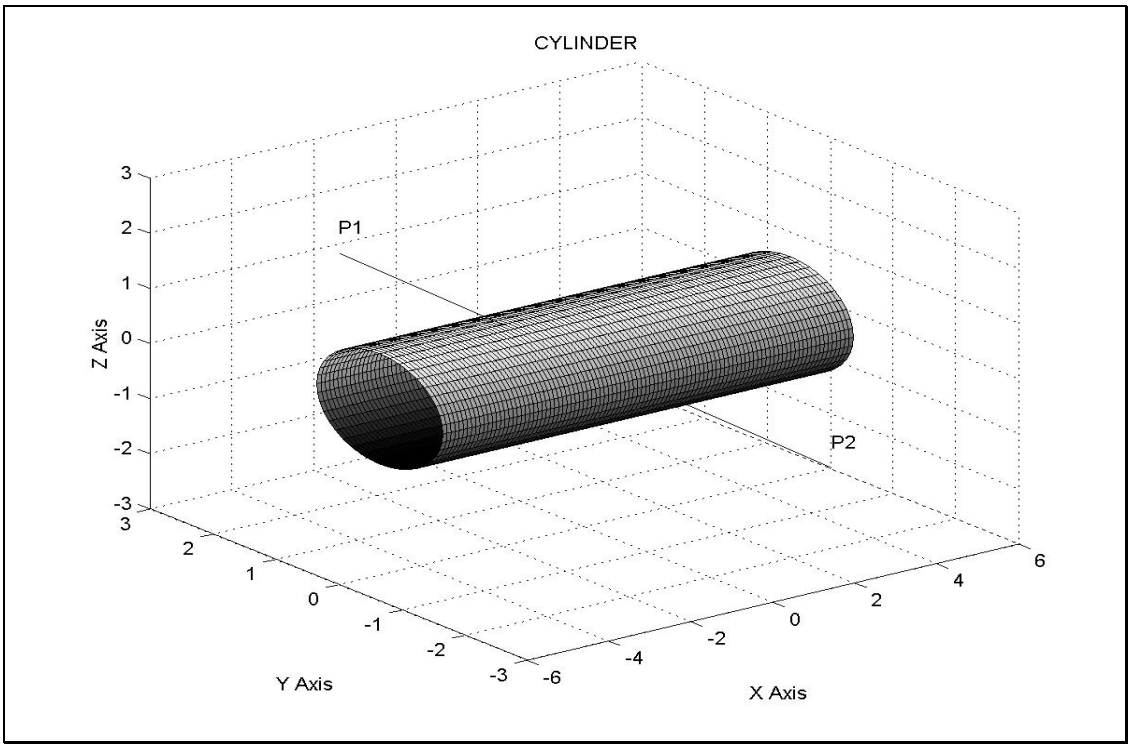

Figure 21. Ray-Cylinder Intersection.

To check if a cylinder shadows a scatter point on a target, the following algorithm is used. The ray is defined by two points, $P_1 = (X_1, Y_1, Z_1)$ and $P_2 = (X_2, Y_2, Z_2)$. In the model, P_1 is the location of the target scatter point and P_2 is the location of the radar. The behavior of the ray is mathematically described by Equation 17. The cylinder is mathematically described by Equation 16. Note from this description that the cylinder is in its *primitive* location, aligned along the X-axis and centered about the origin. A graphical illustration of the ray-cylinder intersection is provided by Figure 21.

Equation 16: $$Z^2 + Y^2 = R^2$$

Equation 17:
$$\begin{aligned} X &= X_1 + U(X_2 - X_1) \\ Y &= Y_1 + U(Y_2 - Y_1) \\ Z &= Z_1 + U(Z_2 - Z_1) \end{aligned}$$

Equation 18:
$$\begin{aligned} &((Z_2 - Z_1)^2 + (Y_2 - Y_1)^2) \cdot U^2 \\ &+ 2 \cdot (Z_1(Z_2 - Z_1) + Y_1(Y_2 - Y_1)) \cdot U + Z_1^{\,2} + Y_1^{\,2} - R^2 = 0 \end{aligned}$$

Equation 19:

$$U = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$
$$A = ((Z_2 - Z_1)^2 + (Y_2 - Y_1)^2)$$
$$B = 2 \cdot (Z_1(Z_2 - Z_1) + Y_1(Y_2 - Y_1))$$
$$C = Z_1^{\,2} + Y_1^{\,2} - R^2$$

Substituting the equation for the ray into the equation for a cylinder yields a quadratic equation given by Equation 18. The behavior of the ray-cylinder intersection is understood by examining the quadratic formula given by Equation 19:

- If the discriminate is < 0, the ray does not intersect the *infinite* cylinder
- If the discriminate = 0, the ray intersects the *infinite* cylinder at exactly one point
- If the discriminate is > 0, the ray intersects the *infinite* cylinder at two points

If the ray intersects the infinite cylinder, the intersection point(s) are found by solving the quadratic formula and substituting the roots back into Equation 17. This gives at most two points of intersection, P_3 and P_4. To determine whether or not the ray intersects the *finite* cylinder at P_3 or P_4, the conditions specified by Equation 20 are checked.

Equation 20:

$$For\ P_3 : X_{\min} < X_3 < X_{\max}$$
$$For\ P_4 : X_{\min} < X_4 < X_{\max}$$
$$where : X_{\min} = -0.5 \cdot CylinderLength$$
$$X_{\max} = 0.5 \cdot CylinderLength$$

If P_3 and/or P_4 intersect the finite cylinder, the final step is to check if the range from the radar to P_1 is greater than the range from the radar to P_3 or P_4, as calculated by Equation 21. If the range to P_1 is greater, the cylinder does not shadow the scatter point; otherwise, the scatter point is shadowed.

Equation 21:

$$P_1\ Range = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2 + (Z_2 - Z_1)^2}$$
$$P_{3,4}\ Range = \sqrt{(X_{3,4} - X_1)^2 + (Y_{3,4} - Y_1)^2 + (Z_{3,4} - Z_1)^2}$$

4.4.3 Ray versus Frustum Intersection

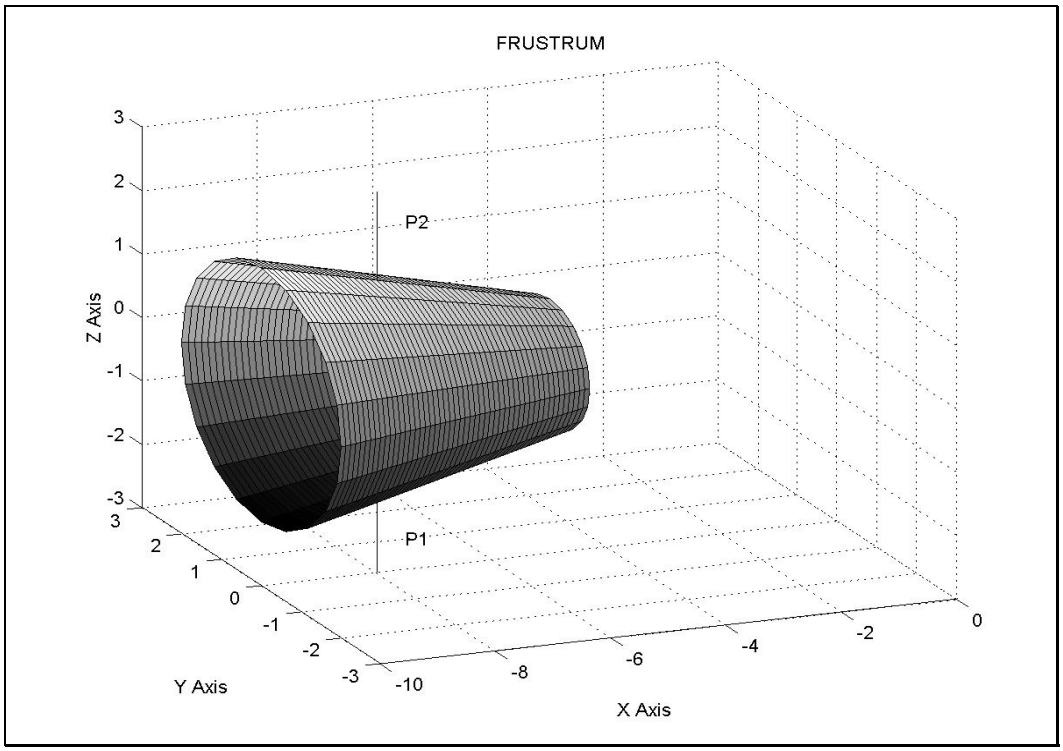

Figure 22. Ray-Frustum Intersection.

The algorithm for computing the ray-frustum intersection is similar to the previous algorithms with a few minor exceptions. As before, a ray is defined by two points, $P_1 = (X_1, Y_1, Z_1)$ and $P_2 = (X_2, Y_2, Z_2)$. P_1 is the location of the target scatter point and P_2 is the location of the radar. The ray's behavior is mathematically described by Equation 22. The frustum is essentially an infinite double cone, as described by Equation 23, with restrictions placed on its minimum and maximum extent. Note that the frustum has a slope associated with it which is determined by Equation 24. The frustum is geometrically described by its base and cap radius and the length between its base and end caps. For the intersection algorithm, the frustum needs to be at its primitive location, aligned along the X-axis with the tip of the frustum (if it were extended to be cone) at the origin of the coordinate system. A handy formula for determining the location of the frustum "tip" is Equation 25, which calculates the length from the base to the tip of the frustum if it were extended to be a cone. In performing these calculations, it may be useful to refer to Figure 23, which provides a graphical illustration of a two-dimensional frustum slice.

Equation 22:
$$\begin{aligned} X &= X_1 + U(X_2 - X_1) \\ Y &= Y_1 + U(Y_2 - Y_1) \\ Z &= Z_1 + U(Z_2 - Z_1) \end{aligned}$$

Equation 23:
$$\frac{Z^2 + Y^2}{c^2} = X^2$$

Equation 24:
$$c = \frac{baseRadius - capRadius}{FrustrumLength}$$

Equation 25:
$$ConeLength = \frac{baseRadius \cdot FrustrumLength}{(baseRadius - capRadius)}$$

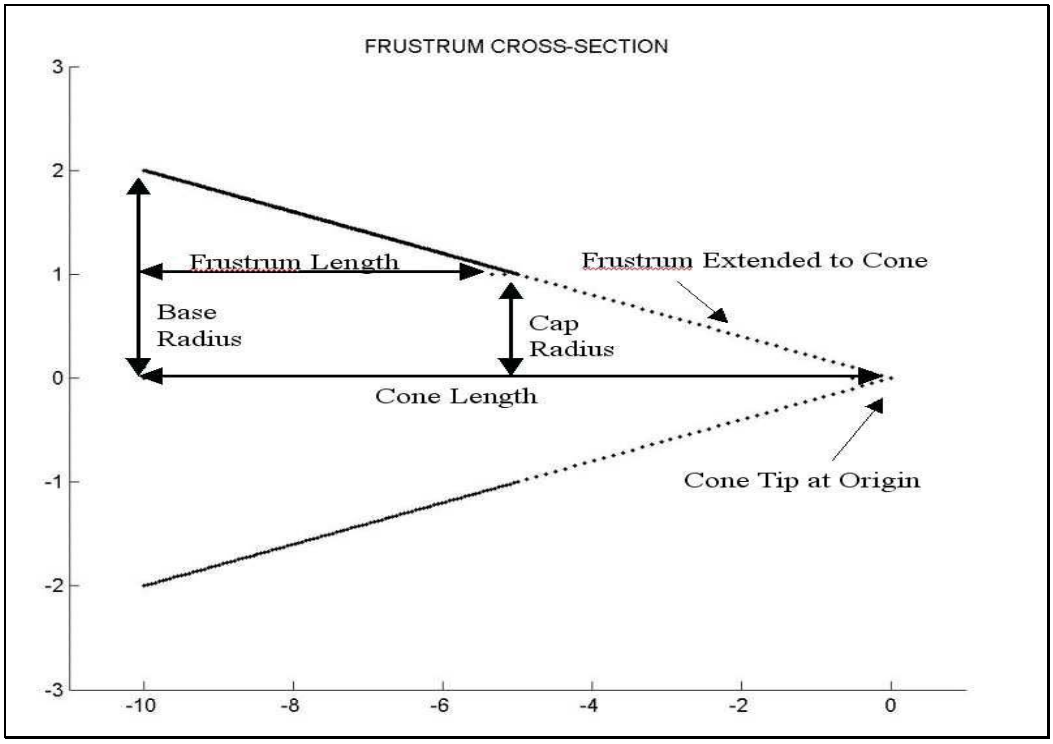

Figure 23. Cross-section of Frustum Primitive.

Substituting the equation for the ray into the equation for the frustum yields a quadratic equation given by Equation 26. The behavior of the ray-frustum intersection is understood by examining the quadratic formula given by Equation 27:

- If the discriminate is < 0, the ray does not intersect the *infinite double cone* (containing the frustum section).
- If the discriminate = 0, the ray intersects the *infinite double cone* (containing the frustum section), at exactly one point.
- If the discriminate is > 0, the ray intersects the *infinite double cone* (containing the frustum section) at exactly two points.

If the ray intersects the *infinite double cone* (containing the frustum section), then the point(s) of intersection are found by solving the quadratic formula using Equation 27 and substituting the roots back into Equation 17. This gives at most two points of intersection, P_3 and P_4. To determine whether or not the ray intersects the frustum at P_3 or P_4, the corresponding conditions specified by Equation 28 must be met.

Equation 26:

$$\begin{aligned}&((Z_2 - Z_1)^2 + (Y_2 - Y_1)^2 - c^2(X_2 - X_1)^2) \cdot U^2 + \\ &2 \cdot (Z_1(Z_2 - Z_1) + Y_1(Y_2 - Y_1) - c^2 X_1(X_2 - X_1)) \cdot U + \\ &Z_1^{\,2} + Y_1^{\,2} - c^2 X_1^{\,2} = 0\end{aligned}$$

Equation 27:

$$\begin{aligned}&U = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \\ &A = (Z_2 - Z_1)^2 + (Y_2 - Y_1)^2 - c^2(X_2 - X_1)^2 \\ &B = 2 \cdot (Z_1(Z_2 - Z_1) + Y_1(Y_2 - Y_1) - c^2 X_1(X_2 - X_1)) \\ &C = Z_1^{\,2} + Y_1^{\,2} - c^2 X_1^{\,2}\end{aligned}$$

Equation 28:

$$\begin{aligned}&For\ P_3 : X_{\min} < X_3 < X_{\max} \\ &For\ P_4 : X_{\min} < X_4 < X_{\max} \\ &where : X_{\min} = Minimum\ X\ Extent\ of\ Frustrum\ \mathrm{Pr}imitive \\ &\qquad\quad X_{\max} = Maximum\ X\ Extent\ of\ Frustrum\ \mathrm{Pr}imitive\end{aligned}$$

If P_3 and/or P_4 intersect the frustum, then the final step of the intersection algorithm is to use Equation 29 to check if the range from the radar to P_1 is greater than the range from the radar to points P_3 or P_4. If the range to P_1 is greater, then the frustum does not shadow the scatter point; otherwise, the scatter point is shadowed.

Equation 29:

$$P_1 \ Range = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2 + (Z_2 - Z_1)^2}$$
$$P_{3,4} \ Range = \sqrt{(X_{3,4} - X_1)^2 + (Y_{3,4} - Y_1)^2 + (Z_{3,4} - Z_1)^2}$$

4.4.4 Ray versus Disc Intersection

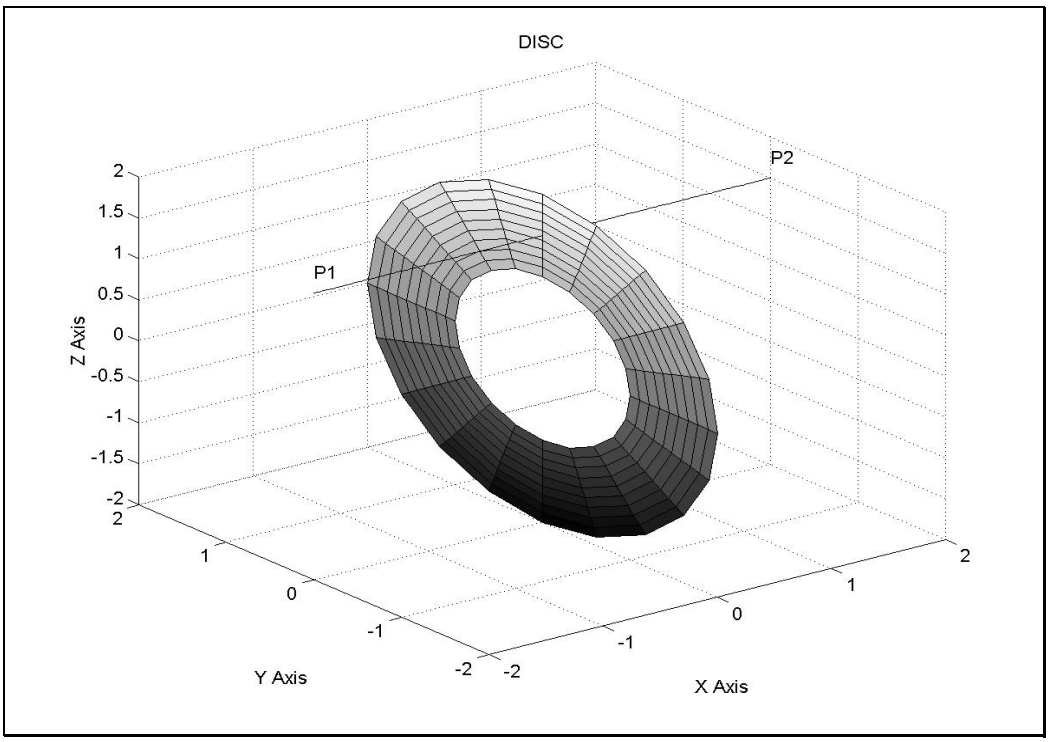

Figure 24. Ray-Disc Intersection.

The ray-disc intersection algorithm is especially useful since many times cylinders, cones, or frustums have base and/or end caps. As before, the ray is defined by two points, $P_1 = (X_1, Y_1, Z_1)$ and $P_2 = (X_2, Y_2, Z_2)$ and is mathematically described by Equation 30. In the simulation, P_1 is the location of the target scatter point and P_2 is the location of the radar. The disc is geometrically described by its inner and outer radius as shown in Figure 24. The plane that *contains* the disc is mathematically described by Equation 31. For the intersection algorithm, it is *not* important for the facet to be located at some primitive position.

Equation 30:

$$X = X_1 + U(X_2 - X_1)$$
$$Y = Y_1 + U(Y_2 - Y_1)$$
$$Z = Z_1 + U(Z_2 - Z_1)$$

Equation 31:
$$Ax + By + Cz + D = 0$$
$$where: \ A, B, C \ are \ components \ normal \ to \ the \ plane$$

Equation 32: $x = 0 \ \ for \ a \ plane \ containing \ a \ disc \ primitive$

The first step in the intersection algorithm is to determine the value for "D" in Equation 31 (assuming the normal vectors to the plane containing the disc have already been determined). This is done by simply substituting the coordinates for the origin of the disc into the equation for the plane and solving for D. To determine the point of intersection of the ray and the plane containing the disc, substitute Equation 30 into Equation 31 and solve for U using Equation 33. Once the value for U is found, it is plugged back into the equation for the ray to determine the point of intersection P_0.

Equation 33:
$$U = \frac{d + aX_1 + bY_1 + cZ_1}{(a \cdot (X_1 - X_2) + b \cdot (Y_1 - Y_2) + c \cdot (Z_1 - Z_2)}$$

To determine if P_0 intersects the disc, Equation 34 is used to compute the distance between P_0 and the disc's origin. If this distance is greater than the disc's inner radius but less than the disc's outer radius, then the ray intersects the disc.

Equation 34:
$$P_0 \ \ Radius = \sqrt{(X_0 - X_{origin})^2 + (Y_0 - Y_{origin})^2 + (Z_o - Z_{origin})^2}$$

For the intersection point P_0 to shadow the scatter point P_1, the range from the intersection point to the radar must be less than the range from the scatter point to the radar, as computed in Equation 35.

Equation 35:
$$P_1 \ \ Range = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2 + (Z_2 - Z_1)^2}$$
$$P_0 \ \ Range = \sqrt{(X_2 - X_o)^2 + (Y_2 - Y_0)^2 + (Z_2 - Z_0)^2}$$

4.4.5 Ray versus Three Vertex Facet Intersection

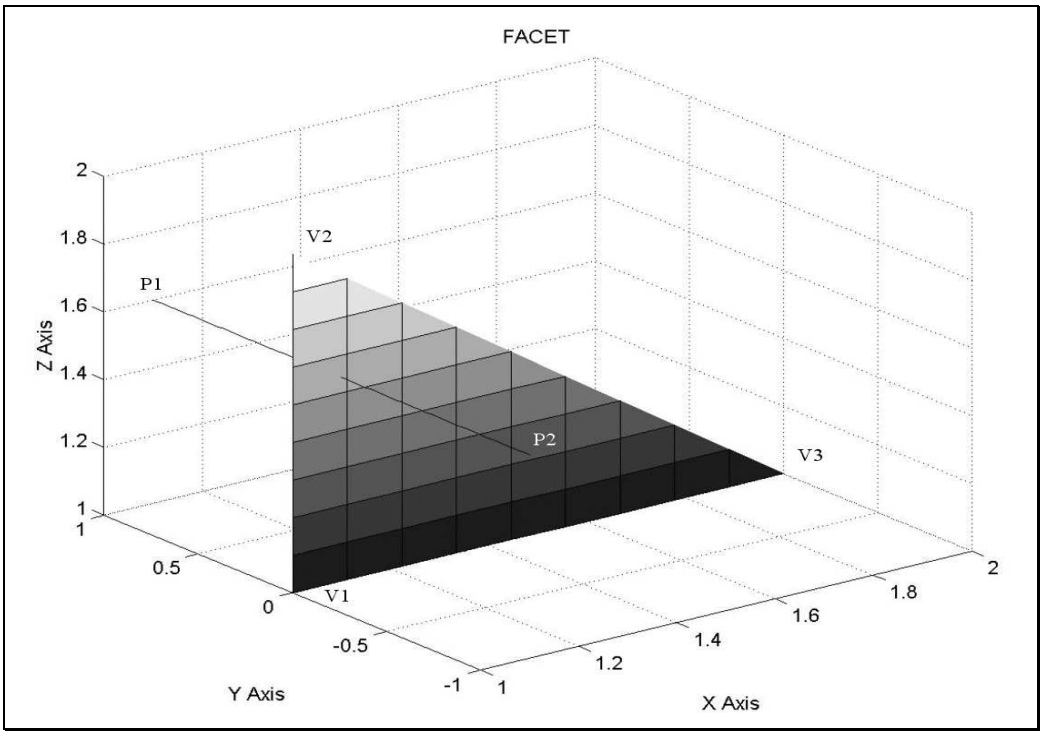

Figure 25. Ray-Three Vertex Facet Intersection.

The ray-facet intersection algorithm can be used to determine if a ray intersects an arbitrarily positioned, three vertex facet. As before, the ray is defined by two points, $P_1 = (X_1, Y_1, Z_1)$ and $P_2 = (X_2, Y_2, Z_2)$ and is mathematically described by Equation 36. In the simulation, P_1 is the location of the target scatter point and P_2 is the location of the radar. The facet is geometrically described by its three vertexes, V_1, V_2, and V_3. The plane that *contains* the facet is mathematically described by Equation 37. For the intersection algorithm it is *not* important for the facet to be located at some primitive position.

Equation 36:
$$\begin{aligned} X &= X_1 + U(X_2 - X_1) \\ Y &= Y_1 + U(Y_2 - Y_1) \\ Z &= Z_1 + U(Z_2 - Z_1) \end{aligned}$$

Equation 37:
$$aX + bY + cZ + d = 0$$
$$where: \ a, b, c \ are \ components \ normal \ to \ the \ plane$$

The first step in the intersection algorithm is to determine the value for "d" in Equation 37 (assuming the normal vectors to the plane containing the facet have already been determined). This is done by simply substituting one of the vertices, V_1, V_2, or V_3 into the equation for the plane and solving for D. To determine the point of intersection, if any, of the ray and the plane containing the facet, substitute Equation 36 into Equation 37 and solve for U using Equation 38. Once the value for U is found, it is plugged back into the equations for the ray to determine the point of intersection P.

Equation 38:
$$U = \frac{d + aX_1 + bY_1 + cZ_1}{(a \cdot (X_1 - X_2) + b \cdot (Y_1 - Y_2) + c \cdot (Z_1 - Z_2)}$$

To check if the point P_o intersects the facet, the geometric principle that the sum of the internal angles of a point on the interior of a triangle is 2Π can be used. The internal angles α_1, α_2, and α_3 can be computed by forming the unit vectors $P_{\alpha 1}$, $P_{\alpha 2}$, and $P_{\alpha 3}$ using Equation 39. The internal angles are then given by Equation 40. If the sum of the internal angles is 2Π, the ray interests the facet at point P_o.

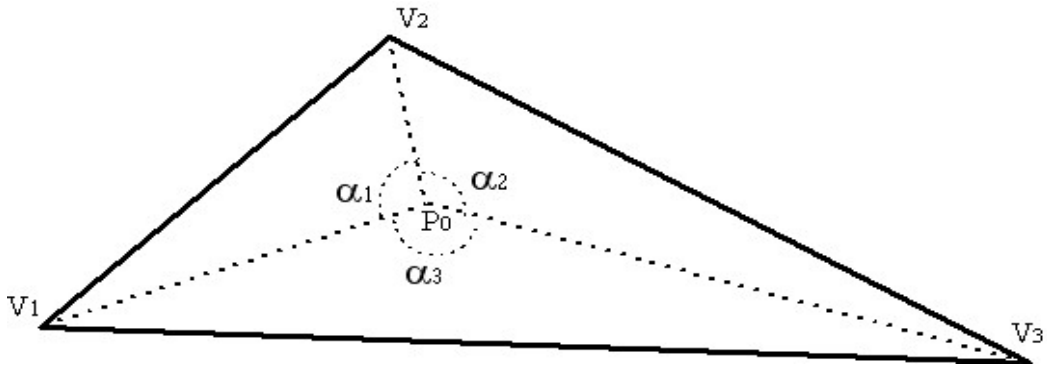

Figure 26. Sum of the Interior Angles of a Point in a Triangle.

Equation 39:
$$P_{\alpha 1} = \frac{(V_1 - P)}{|V_1 - P|}$$
$$P_{\alpha 2} = \frac{(V_2 - P)}{|V_2 - P|}$$
$$P_{\alpha 3} = \frac{(V_3 - P)}{|V_3 - P|}$$

Equation 40:

$$\begin{aligned}\alpha_1 &= \cos^{-1}(P_{\alpha 1} \bullet P_{\alpha 2}) \\ \alpha_2 &= \cos^{-1}(P_{\alpha 2} \bullet P_{\alpha 3}) \\ \alpha_3 &= \cos^{-1}(P_{\alpha 3} \bullet P_{\alpha 1})\end{aligned}$$

For the point P_0 to shadow the target scatter point P_1, the range from intersection point to the radar must be less than the range from the target scatter point to the radar, as computed in Equation 41.

Equation 41:

$$\begin{aligned}P_1\ Range &= \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2 + (Z_2 - Z_1)^2} \\ P_0\ Range &= \sqrt{(X_2 - X_0)^2 + (Y_2 - Y_0)^2 + (Z_2 - Z_0)^2}\end{aligned}$$

4.4.6 Intersections with Arbitrarily Positioned Shapes

Of all the ray tracing techniques discussed so far, only the ray-disc and ray-facet intersection algorithm allow for the shape to be arbitrarily positioned. The sphere, cylinder, and frustum intersection algorithms all have the restriction that the shape is centered about the origin and aligned parallel to the X-axis. There are two possible solutions to determine the point of intersection when one of these shapes is arbitrarily located:

1. Find the general intersection algorithm between the ray and the arbitrarily located shape.
2. Use geometric transformations to rotate and translate the shape into its "primitive location", meaning the shape is centered about the axis origin and parallel to the X-axis.

While both solutions are equally valid, the second one is implemented in the simulation because it allows for the relatively simple intersection algorithms discussed above to be used as opposed to more complex equations for handling shapes at arbitrary positions.

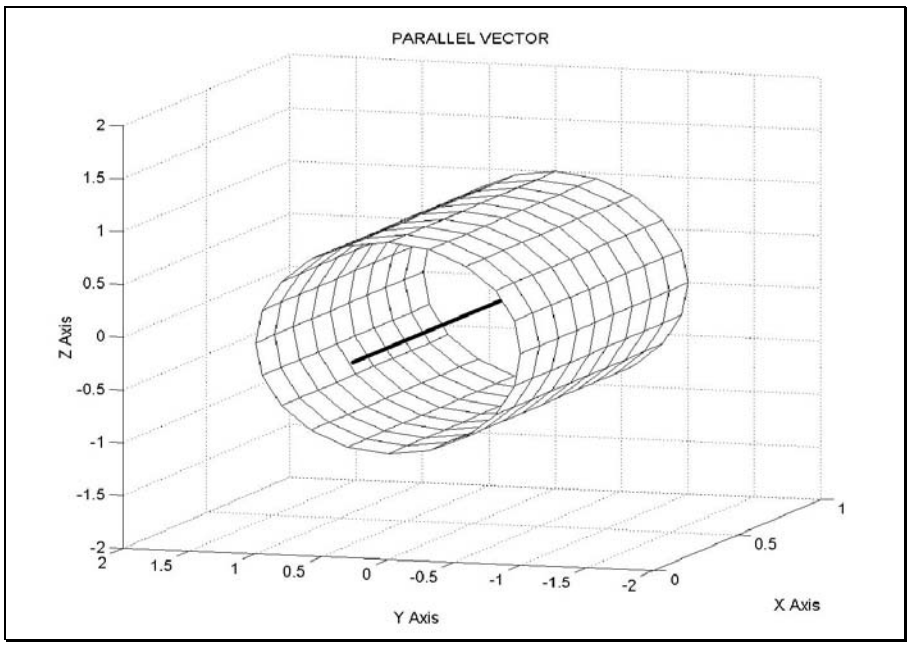

Figure 27. Vector Parallel to Object Body.

To aide in the determination of the translation and rotation needed to move an arbitrarily positioned shape to its primitive position, the coordinates of the shape's origin and a unit vector describing its orientation should be encapsulated within each shape object. Whenever the shape is rotated or translated, the shape's origin and orientation should be updated accordingly. Using the information about the shape's origin, Equation 42 can be used to determine the geometric translation required to center the shape about the axis origin.

Equation 42:
$$(X', Y', Z') = (X, Y, Z) - (X_o, Y_o, Z_o)$$
$$where: (X_o, Y_o, Z_o)\ is\ the\ location\ of\ the\ shape's\ origin$$

Once the shape is positioned about the axis origin, it needs to be oriented so that it is symmetrical about all three axes. Certain shapes like spheres, once centered about the axis origin, require no rotation since they are already symmetrical about all three axes. Other shapes, like cones, cylinders, and frustums, need to be rotated about the Y and Z-axis to make them symmetrical about all three axes. To determine the angles of rotation required for these types of shapes, the vector components describing the shape's orientation are plugged into Equation 43 and Equation 44.

Equation 43:

$$\alpha = -\tan^{-1}\left(\frac{N_y}{N_x}\right) \qquad \textit{Angle of rotation about Y axis}$$

$$\textit{where}: N \textit{ is unit vector parallel to object's body}$$

$$\tan^{-1} \textit{ is the four quadrant } \arctan\ (-\pi < \tan^{-1} < \pi)$$

Equation 44:

$$\beta = \tan^{-1}\left(\frac{N_z}{\sqrt{N_x{}^2 + N_y{}^2}}\right) \qquad \textit{Angle of rotation about Z axis}$$

$$\textit{where}: N \textit{ is unit vector parallel to object's body}$$

$$\tan^{-1} \textit{ is the four quandrant } \arctan\ (-\pi < \tan^{-1} < \pi)$$

Once the angles of rotation α and β are determined, they can be plugged into the direction cosine matrices provided by Equation 46 and Equation 47 to rotate the shape about the Y and Z-axis. It should be noted that shapes like cylinders, cones, and frustums need not be rotated about the X-axis to position them into their primitive locations. This is due to the geometry of these shapes, whereby they are symmetrical along their body axis. If these shapes were asymmetrical about their body axis, then a rotation about the X-axis would be required to correctly orient the shape.

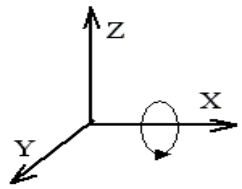

Figure 28. Rotation about X Axis.

Equation 45:

$$\begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

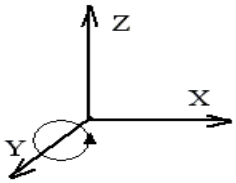

Figure 29. Rotation about Y Axis.

Equation 46:

$$\begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

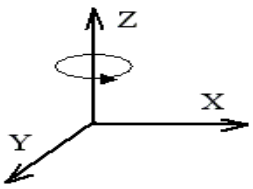

Figure 30. Rotation about Z Axis.

Equation 47:

$$\begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

CHAPTER V
SIMULATING THE RADAR

5.1 Setting up the Radar

After setting up the target model, the next step in building the simulation is modeling how the radar perceives targets. Like the target, the radar can be modeled at several different layers. To truly simulate the radar's behavior, aspects of the radar like the characteristics of the radiation pattern, beam forming, spectral side lobes, frequency mixing, receiver noise, and countless other radar characteristics should be included. For the purposes of this report, however, a minimal model is used to simulate the range-Doppler data captured by the radar. This model consists of the properties and methods required to generate the phase history known as the "in-phase and out-of-phase" data that forms the basis for any down stream radar signal processing. As with the target model, the radar model was designed using object-oriented principles. The model is extensible and additional features or characteristics can be added in the future to enhance the model's fidelity and capability.

5.2 Generating Phase History

Radar perceives targets by measuring their range and angular change in range, also known as range rate. To determine the target's range, the elapsed time between the transmission of the radar pulse and the reception of the target's echo is measured. The radar detects the target's range rate by measuring the change in frequency of the received echo due to the Doppler effect. The Doppler effect is a shift in frequency of a wave radiated, reflected, or received by an object in motion. The greater the object's speed in relation to an observer, the greater the Doppler effect. A common example of the Doppler effect is the observed change in frequency of an ambulance's siren as the vehicle approaches and passes a bystander. As the ambulance approaches, the sound waves emitted by the siren are compressed, translating to a higher frequency sound. As the ambulance passes, the siren's sound waves expand, observable to the bystander as a lower frequency sound.

In the simulation program, the range-Doppler information from the target must be determined and stored in a format suitable for down stream radar signal processing algorithms. The computation of this data, commonly known as "In-Phase and Out-Of-Phase" or "IQ" data is discussed in the following sections.

5.2.1 Computing Slant Ranges

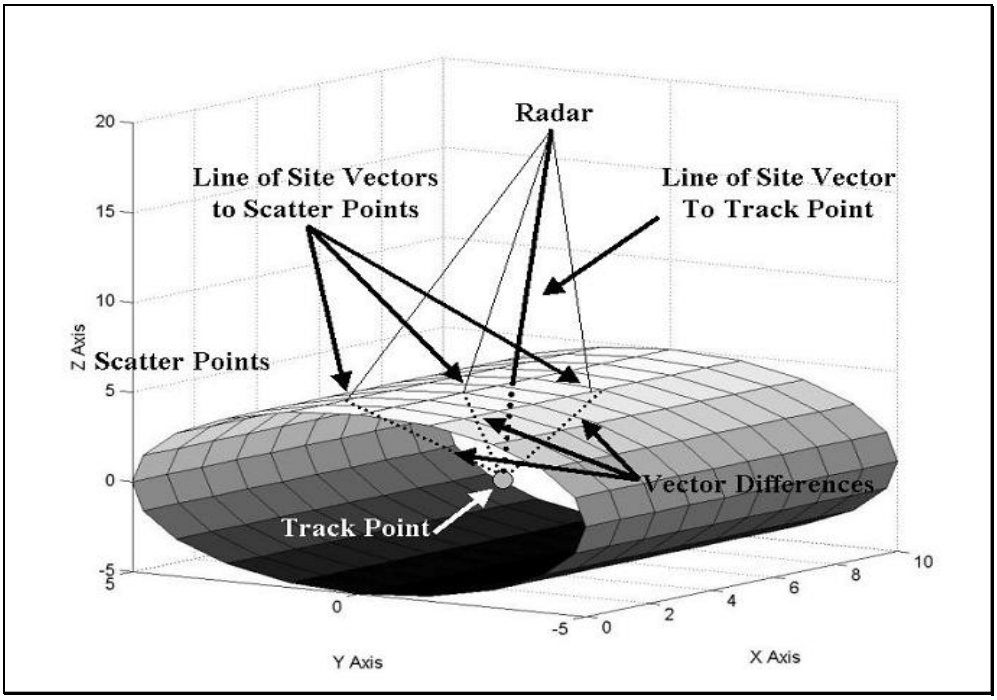

Figure 31. Line of Site Vectors to Scatter Points and Track Point.

The first step toward computing the IQ data is to determine the line of sight vectors from the radar to the target's scatter points and to the centroid of the target, known as its "track point." The slant range to each scatter point is computed per Equation 48 by taking the magnitude of the line of sight vector from the radar to each scatter point. The slant range to the track point is computed per Equation 49 by taking the magnitude of the vector from the radar to the track point. The range difference between the scatter point line of sight vectors and the track point line of sight vector is then computed using Equation 50. Figure 31 provides a graphical description of the track point line of sight vector, a small sampling of

scatter point line of sight vectors, and the vector differences between the scatter points and track point line of sight vectors.

Equation 48:
$$ScatPtSlntRng = \sqrt{\left(X_{ac} - X_{scatpt}\right)^2 + \left(Y_{ac} - Y_{scatpt}\right)^2 + \left(Z_{ac} - Z_{scatpt}\right)^2}$$

Equation 49:
$$TrkPtSlntRng = \sqrt{\left(X_{ac} - X_{trkpt}\right)^2 + \left(Y_{ac} - Y_{trkpt}\right)^2 + \left(Z_{ac} - Z_{trkpt}\right)^2}$$

Equation 50:
$$RngDiff = ScatPtSltRng - TrkPtSlntRng$$

5.2.2 Computing Scatter Point Amplitudes

The next step in determining the phase history is to compute an amplitude for each scatter point on the target. The amplitude determines the amount of energy reflected by the scatter point back to the radar. The first piece of information required is the angle of reflection of the radar wave off the scatter point, as discussed in the target modeling section. The second piece of information required is the wavelength of the radar pulse. The radar's wavelength is inversely proportional to its frequency. The final piece of information needed is the surface area represented by each scatter point, as was determined using the algorithms discussed in the target modeling section. Once this information is known, Equation 51 is used to compute each scatter point's amplitude.

Equation 51:
$$Amp = \left(\frac{4 \cdot \pi \cdot a^4}{\lambda^2} \cdot \frac{sin\left(k \cdot a \cdot sin\left(\theta\right)\right)}{k \cdot a \cdot sin\left(\theta\right)} \right)^2$$

where:

$$a = \sqrt{area\ of\ scatter\ point}$$

$$k = \frac{2}{\lambda}$$

$$\lambda = Radar\ Wave\ Length$$

$$\theta = Angle\ of\ Reflection\ off\ Scatter\ Point$$

From the equation for the amplitude, it is interesting to note how the various parameters affect the magnitude of the radar energy returned by a scatter point. If a scatter point represents a large surface

area, more energy is reflected. Small angles of reflection (close to 0 degrees) reflect much more energy than large angles (closer to 90 degrees). Recall that for negative angles of reflection, the scatter point faces away from the radar, meaning it is not seen by the radar and its amplitude is zero. Note also that a scatter point may be "shadowed" if some other portion of the target blocks the radar from seeing it. In this case the amplitude for the scatter point is zero. The algorithms and ray tracing techniques discussed in the previous sections are used to compute whether or not a scatter point is shadowed by another portion of the target. As a side note, to optimize the ray tracing routines, ray tracing need only be performed for scatter points facing the radar (with an angle of reflection between 0 and 90 degrees).

5.2.3 Computing Scatter Point Phases

In addition to amplitude, phase must also be computed for each scatter point. The phase term provides the Doppler information about the target to the radar. The first phase calculation for the scatter point, given by Equation 52, depends on the radar's wavelength and the magnitude of the vector difference between the scatter and track point line of sight vectors. The initial phase value for the scatter point (set to some uniformly distributed random number at the beginning of the simulation) is then added on to give a phase value for each scatter point.

Equation 52:

$$ScatPtPhs = \frac{4 \cdot \pi \cdot RngDiff}{\lambda} + InitialPhs$$

where:
InitalPhs is set to some uniformly distributed random number at beginning of simulation

The phase value for each scatter point is then plugged into Equation 53. From this equation, the scatter point phase value computed in Equation 52 is added to an array of "range bin terms" that help determine the ranges where the majority of the reflected energy from the scatter point falls into. The final phase terms for each scatter point, therefore, is actually an array of phase values created by adding the value in each cell of the "range term" to the scatter point phase computed in Equation 52.

Equation 53:

$$Phs(1\ldots nSmpBins) = \frac{2 \cdot \pi \cdot RngDiff \cdot SmpBins(1\ldots nSmpBins)}{rpixspace \cdot nSmpBins} + ScatPtPhs$$

where:

$rpixspace = Radar\ Pixel\ Spacing$

$$SmpBins(1\ldots nSmpBins) = -\frac{nSmpBins}{2}\cdots\frac{nSmpBins}{2}$$

$nSmpBins = Number\ of\ Sample\ Bins$

5.2.4 Target Amplitude and Phase for a Radar Pulse

To determine the IQ data for a given radar pulse, the amplitude and phase terms for all scatter points must be summed according to Equation 54. Recall that each scatter point has an amplitude and an array of "phase values" associated with it. This means the summation performed over all the scatter points is repeated for all phase array terms. The final IQ data, therefore, consists of an array of complex valued numbers, with the number of elements in the array equal to the number of samples taken by the radar for each pulse. Figure 32 provides a graphical illustration of how the I and Q terms from each scatter point are summed to create the IQ data for the target for a given radar pulse.

Equation 54:

$$IQ(1...nSmpBins) = \sum_{iScatPt=1}^{nScatPt} Amp \cdot e^{j \cdot Phs(1\ldots nSmpBins)}$$

where:

$nScatPt = Total\ Number\ of\ Scatter\ Points$

$\rightarrow$ *Note: Sum is performed over all scatter points*

$\rightarrow$ *Note: Each sum is repeated for each element in the phase array*

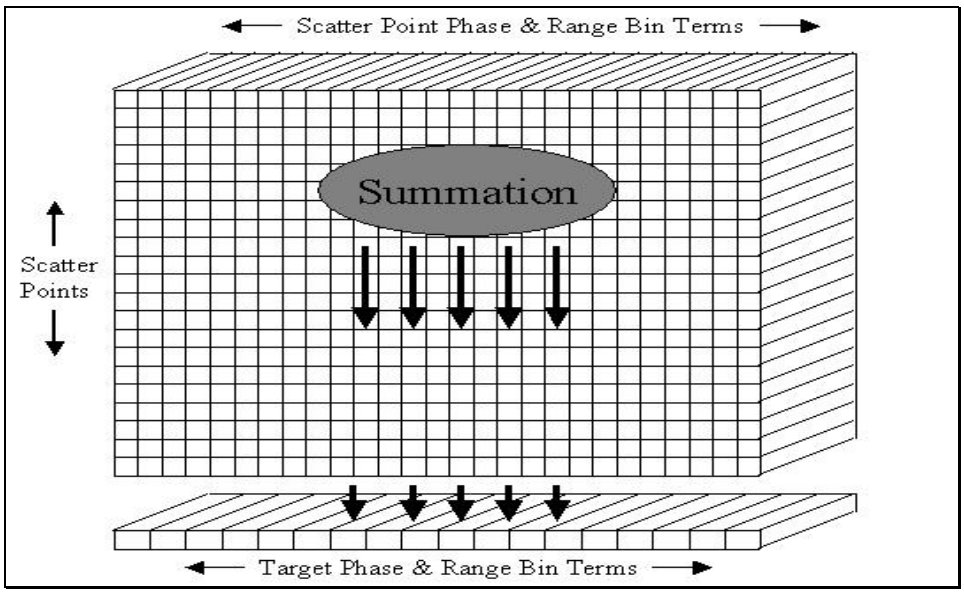

Figure 32. Summation of Scatter Point Phase and Range Terms.

A plot of the IQ data generated for a radar pulse is shown in Figure 33. At first glance, it appears that the plot does not reveal any information about the target's range or range rate. However, by examining the frequency components of the IQ data, information about the target's range and range rate can be extracted. The algorithms and techniques available for extracting this information are discussed in the next chapter.

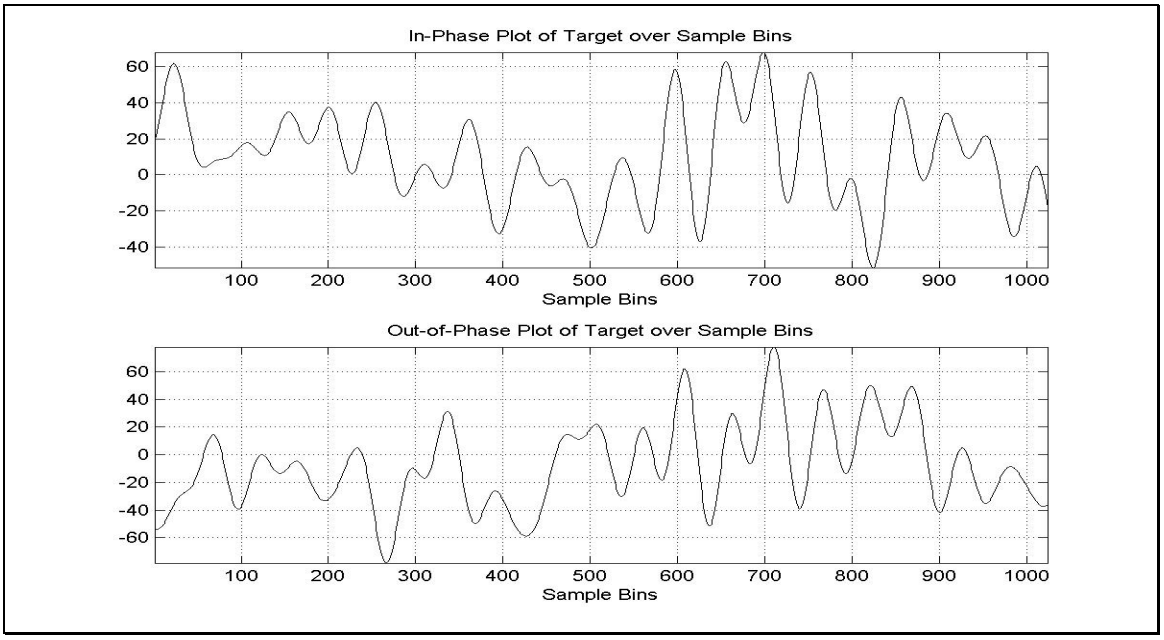

Figure 33. In-Phase and Out-of-Phase Plots for Target for a Radar Pulse.

CHAPTER VI
PROCESSING IQ DATA

6.1 Radar Signal Processing

The IQ data generated in the previous chapter is the most basic data required to perform any downstream radar signal processing. With signal processing techniques, one can improve the signal to noise ratio, perform moving target indication (MTI), pulse-Doppler processing, target range rate measurement, Doppler beam sharpening, synthetic aperture radar (SAR), and inverse synthetic aperture radar processing (ISAR). For this report, the signal processing techniques for creating range magnitude plots (A-Scans) and range-Doppler images are briefly explained.

6.2 Non-Coherent Processing

The range magnitude plot, or "A-scan", illustrates the relative intensity of the target at different ranges from the radar. The trick to creating the A-Scan is extracting the magnitude of the frequencies encoded in the IQ data. This is done by taking the FFT of the sampled IQ data collected by the radar over a single pulse, as shown by Equation 55. The FFT effectively translates the IQ data from the time domain to the frequency domain. By examining the magnitudes of the various frequencies present, and correlating them to the "range bins" they occur in, the ranges at which the target signature occurs can be determined.

Equation 55:

$$IQ_{rngcmp}(1\ldots nSmpBins) = fftshift\left(fft\left(IQ_{weighted}(1\ldots nSmpBins)\right)\right)$$

where:

$$fft(x) = X(k) = \sum_{j=1}^{nSmpBins} x(j)\cdot\omega_{nSmpBins}^{-(j-1)(k-1)}$$

$$\omega_{nSmpBins} = e^{\frac{(-2\pi i)}{nSmpBins}} \text{ is an } nSmpBins_{th} \text{ root of unity}$$

$$fftshift\left(x(1\ldots nSmpBins)\right) \text{ swaps right and left halves of array } x$$

shifting zero frequency component to center

In computing the FFT of the IQ data, the following observations should be noted. The first is that the samples are first weighted by a "hamming window" that is computed according to Equation 56. Hamming windows have the desired property that their Fourier transforms are concentrated around ω=0. The Hamming window is calculated using Equation 56. A graphical illustration of the Hamming window in the time domain is provided by Figure 34. With windowing, side lobes are greatly reduced; however, the price paid is a much wider main lobe. After the FFT is performed, the right and left halves of the resulting array are swapped to shift the zero-frequency component to the center of the frequency spectrum (Oppenheim, 1989).

Equation 56:

$$IQ_{weighted} = Hamming(1 \ldots nSmpBins) * IQ(1 \ldots nSmpBins)$$

$$where:$$

$$Hamming[k+1] = 0.54 - 0.46 \cdot \cos\left(\frac{2 \cdot \pi \cdot k}{nSmpBins - 1}\right), \quad k = 0, \ldots, nSmpBins - 1$$

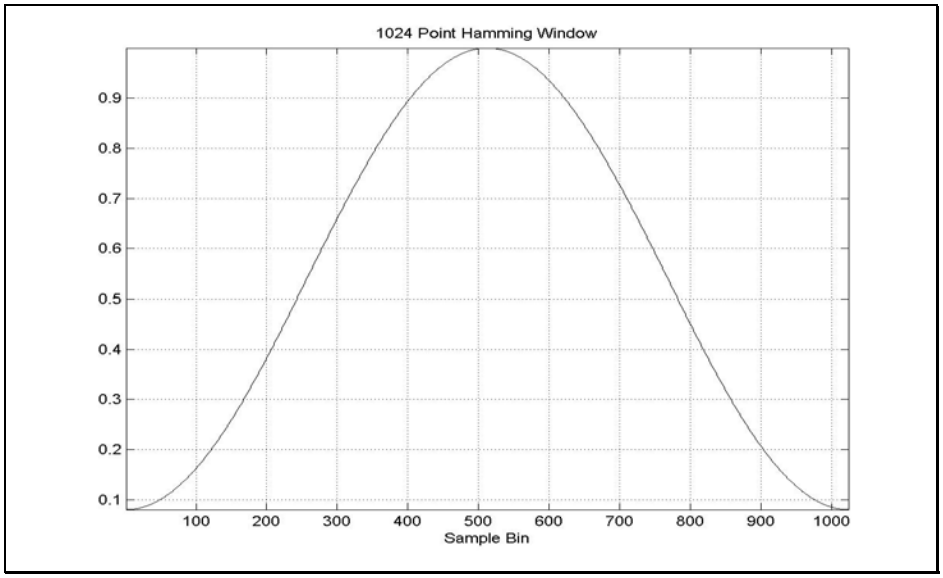

Figure 34. Hamming Window.

A sample A-scan plot of a missile target is provided by Figure 35. From this plot it is evident that the target signature is strongest in range bins 480-550. Each range bin corresponds to some range from the radar. For each pulse of the radar, a new A-scan plot can be generated according to the new IQ data samples collected from the radar returns. Over time, the target migrates through range bins as it moves

closer to or further from the radar. Downstream processing uses the range data computed for each pulse, or for a collection of pulses, to "track" the target and keep it centered about the display. Processing the range data by performing an FFT across the IQ samples for each pulse is known as non-coherent processing since only range information is kept. In the next section, coherent processing is discussed, where both the range and Doppler information is retained.

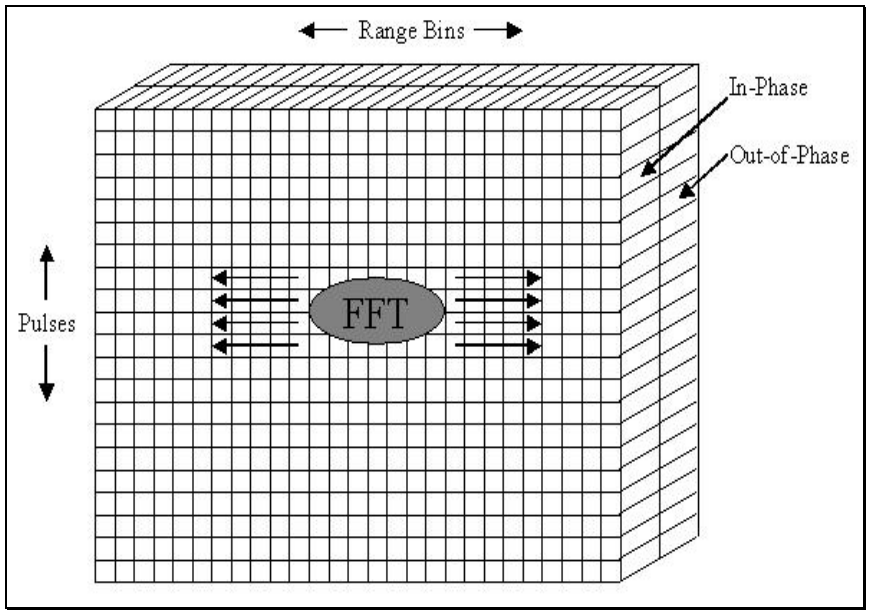

Figure 35. Non-Coherent Processing of Video Phase History.

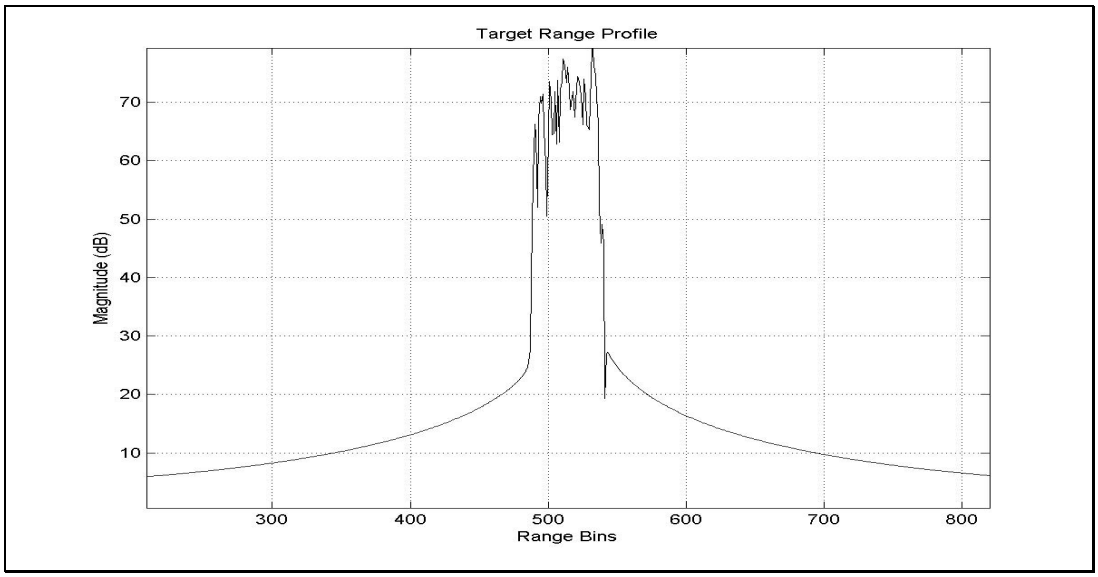

Figure 36. Range Profile for Target (no noise) Occupying Range Bins 480-550.

6.3 Coherent Processing

With coherent processing, both the range and Doppler information about the target is retained. From the previous section, it was shown by taking the FFT of the IQ samples for each pulse, that range information about the target is extracted. If in addition to the range FFT, an FFT is taken of the range compressed data over a series of pulses, then the *range rate* or Doppler behavior of the target may be extracted. This can be done by performing a two dimensional FFT over the IQ samples collected for a collection of pulses as shown in Figure 37 and Figure 38. After performing the 2D FFT, complex video data is produced which can be used to generate range-Doppler images of the target. Processing the IQ data in this way, where both the target's range and Doppler are extracted, is known as coherent processing and forms the basis for SAR, ISAR, and many other more advanced radar signal processing techniques.

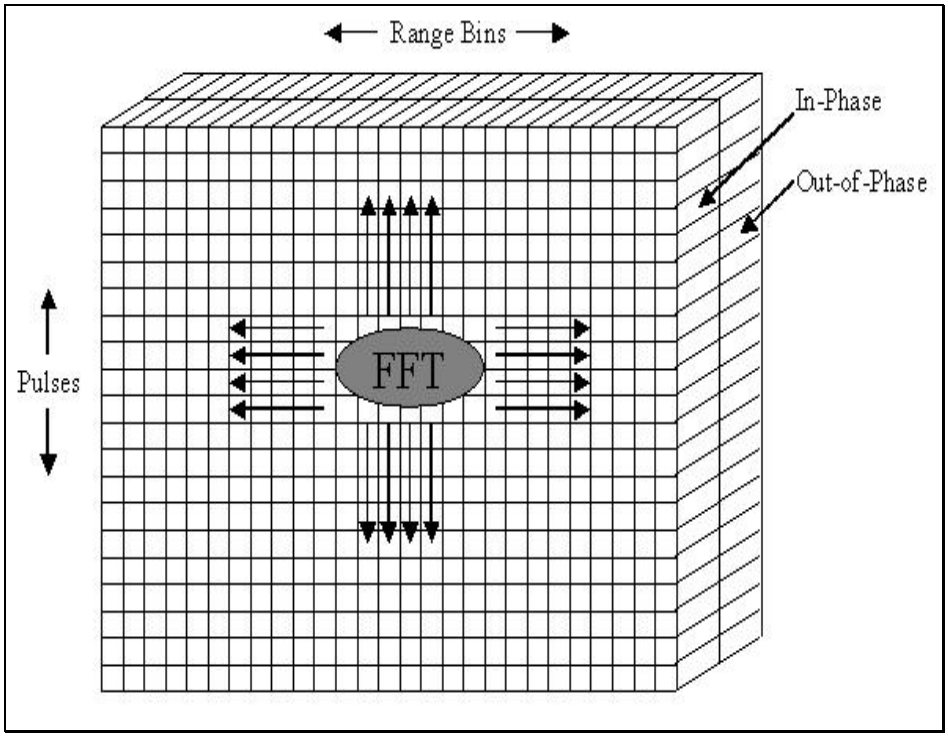

Figure 37. Coherently Processing IQ Data.

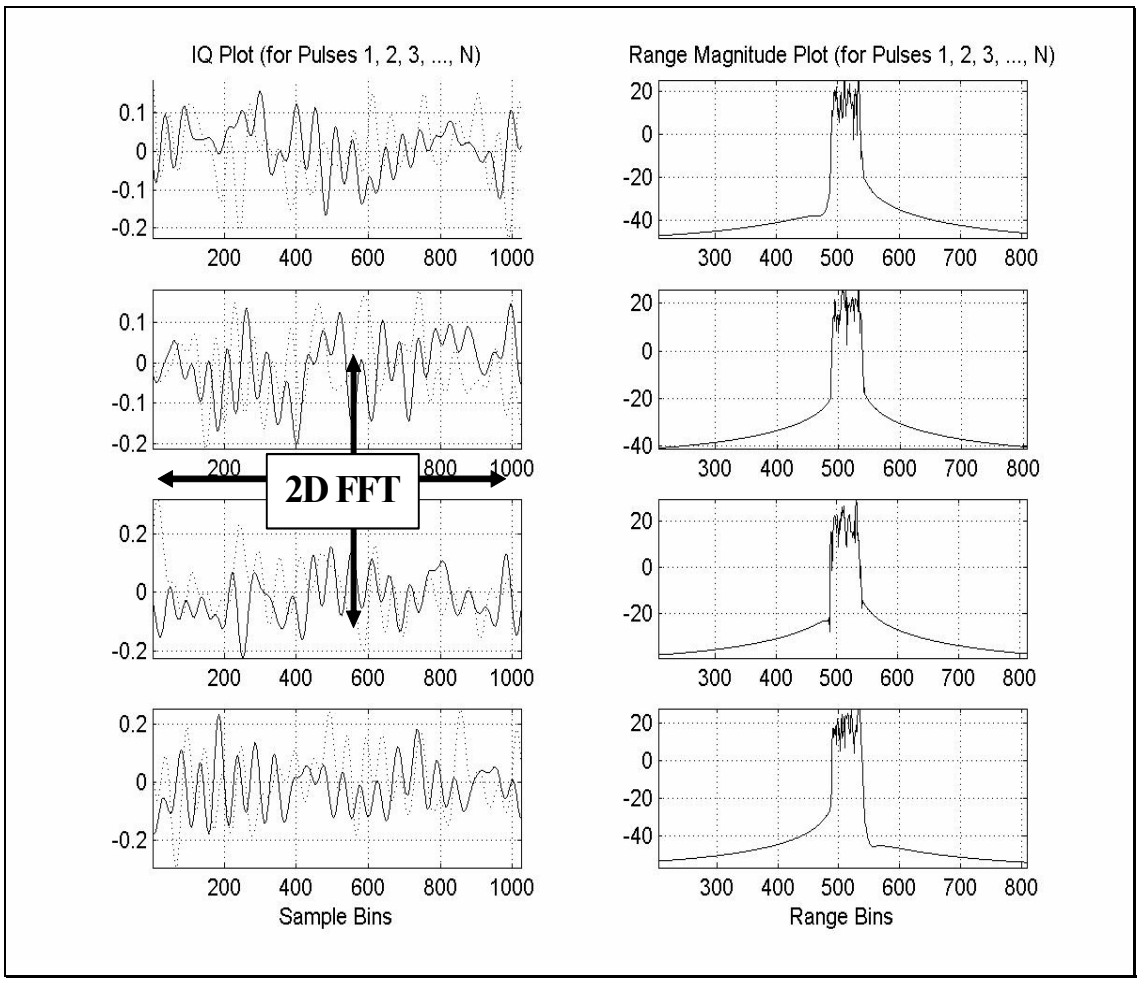

Figure 38. Coherently Processing IQ data over Multiple Pulses.

6.4 Sample Range-Doppler Maps

Some examples of range-Doppler maps produced by the simulation are shown below. For the simulation test runs, the geometry for a missile was input into the simulation. The missile basically consists of an assortment of cones, frustums, discs, cylinders, and facets connected together. The missile was given an initial position, rotation, and velocity. The simulation began with the missile approaching the target at some altitude below the radar and at some large positive X offset away from the radar. At the halfway point, the missile was directly beneath the radar. At the conclusion of the simulation run, the missile was beneath the radar at some large negative X offset from the radar.

6.4.1 An Approaching Target

Figure 39 provides a picture of what the missile would like to an observer located at the radar at the beginning of the simulation. Figure 40 provides a picture of what the radar would see at this same instance in time. From the range-Doppler image, it is clear that the radar resolves the different ranges for the different sections of the missile, with the missile's cone being closest to the radar and its exhaust being furthest in range. Along the Y-axis of the range-Doppler map, it is evident that the rotational motion of the missile translates to a high Doppler component for the fins and a lower Doppler component for the missiles body (since Doppler bins at the top and bottom of the range-Doppler image correspond to high positive and negative Doppler shift). Shadowing of the missile's exhaust nozzle by the missile body and of the fins beneath the missile can also be observed from the range-Doppler map. The change in shadowing of the fins, while not apparent from the static images of the missile, is readily apparent when subsequent image frames are stacked together to create a movie of the missile as it completes its flight.

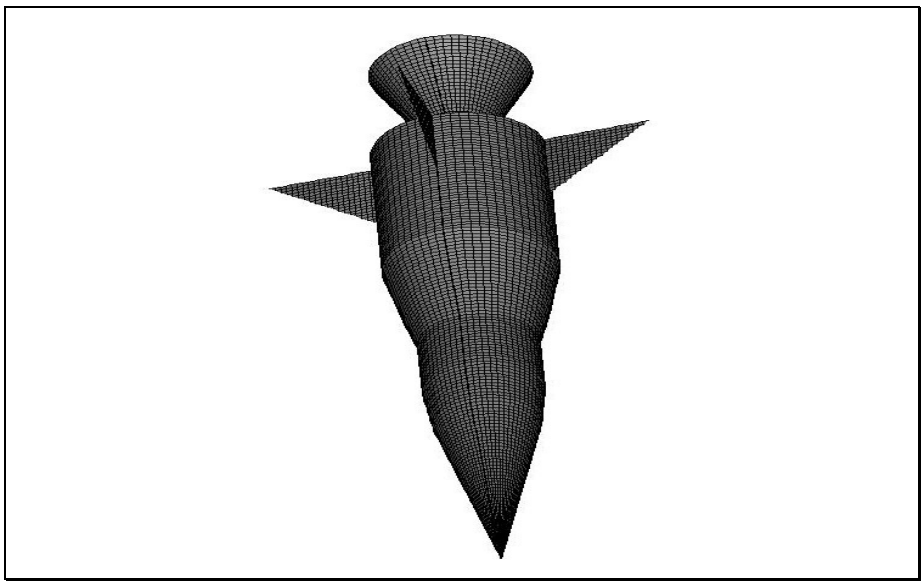

Figure 39. View from Aircraft for Approaching Missile.

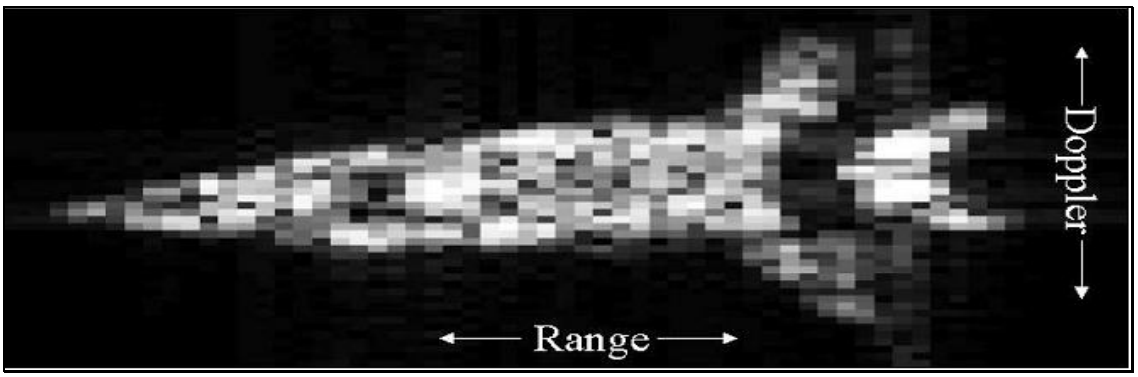

Figure 40. Radar Range-Doppler Image for Approaching Missile.

6.4.2 A Broadside Target

Figure 41 provides a picture of what the missile would like to an observer at the radar's location when the missile is directly beneath the radar. Figure 42 provides a picture of what the radar would see at this same location. From the range-Doppler image, it is clear that the radar has difficulty distinguishing the ranges for the different sections of the missile, since they are "compressed" into a small number of range bins. Along the Y-axis of the range-Doppler map, the rotational motion of the missile translates to a high Doppler component for the fins and a lower Doppler component for the rest of the missile body. Since much of the missile is compressed into a small number of range bins and is oriented broadside to the radar, the phenomena of specular return manifests itself in the smearing in the range and Doppler dimension.

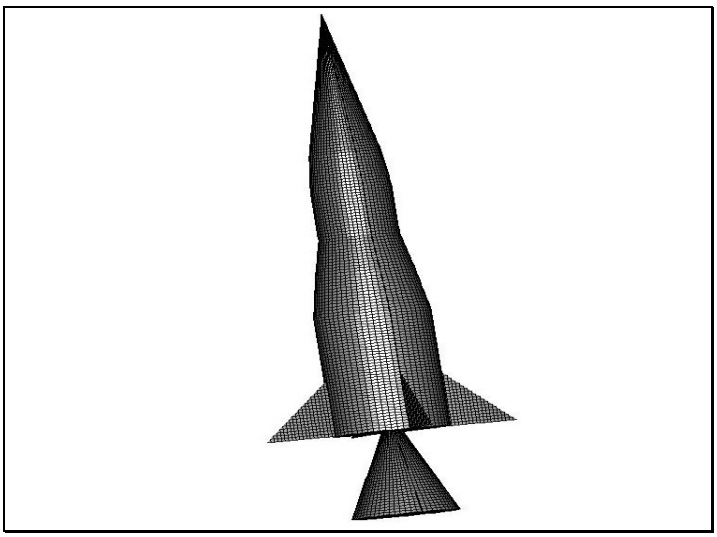

Figure 41. View from Aircraft when Missile Directly below Airplane.

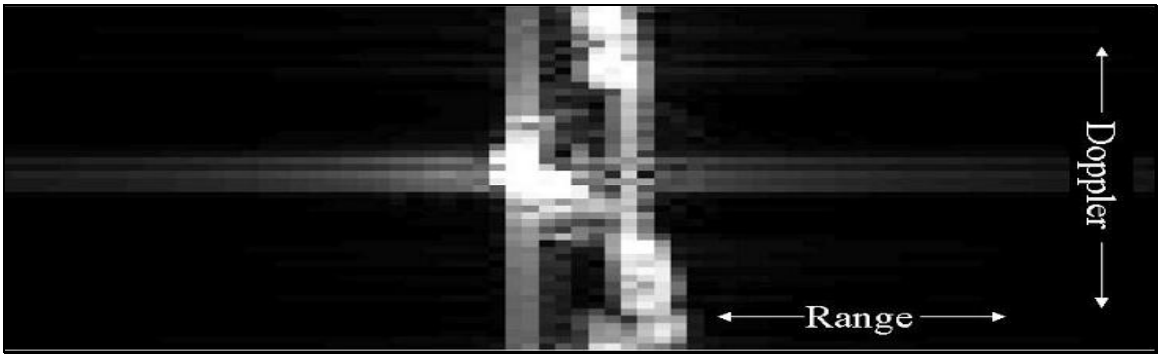

Figure 42. Radar Range-Doppler Image when Missile Directly Below Airplane.

6.4.3 A Receding Target

Figure 43 provides a picture of what the missile would like to an observer at the radar's location when the missile has passed the radar. Figure 44 provides a picture of what the radar would see. From the range-Doppler image, it is clear that the radar image is the exact opposite case of where the target is approaching the radar. The missile is spread across a large number of range bins since its cone and tail are at near and far range. The fins still appear in the Doppler bins representing a high positive or negative Doppler shift and the missile's body still appears in Doppler bins corresponding to a smaller Doppler shift.

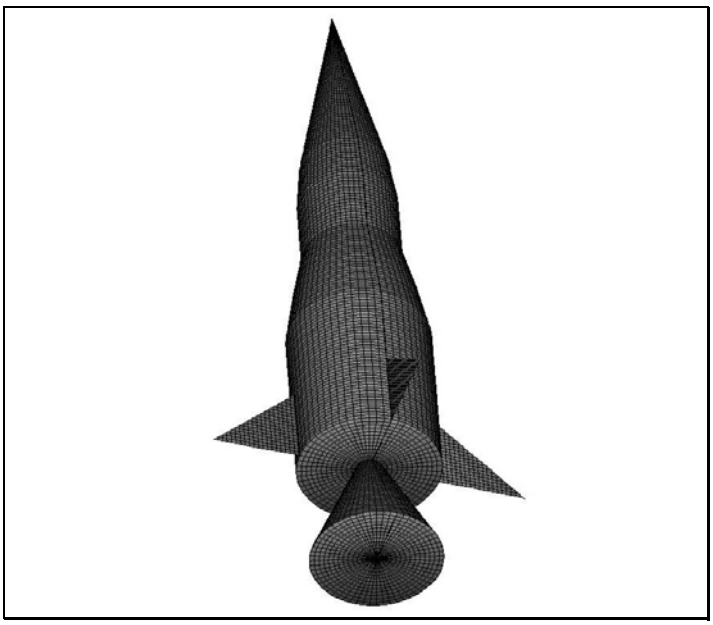

Figure 43. View from Aircraft for Receding Missile.

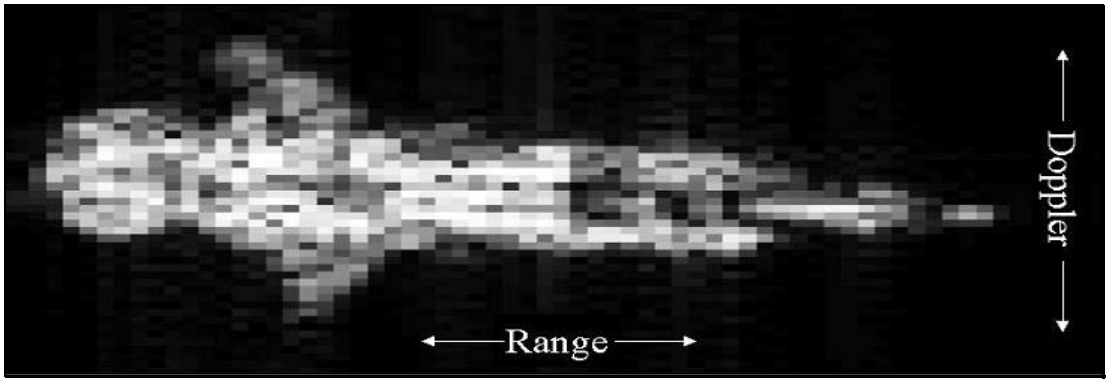

Figure 44. Range Doppler Image for Receding Missile.

CHAPTER VII
SUMMARY AND CONCLUSIONS

Object oriented programming claims to provide many benefits and advantages over structured design. In the business and financial sectors, object oriented techniques have proven to be beneficial in the creation of databases, spreadsheets, and other data driven software. Microsoft has leveraged off of object oriented techniques to build its lucrative windows operating system and office tools suite. The next big thing in the internet are web-based applications written in JAVA that allow users to track stocks, listen to music, watch streaming video, or perform live teleconferencing. Software engineers are evolving to object oriented design, and as a result, are capable of developing ever more complex and powerful software while minimizing complexity and maximizing maintainability.

In the realm of science and engineering, however, object oriented techniques have been slow to catch on. While systems engineers are starting to adopt new OO techniques in the formulation of system requirements and specifications, they are still mired in the mindset of structured design when it comes to creating software. Whereas in the past such structured languages as Fortran, ADA, and MATLAB have provided the necessary tools to perform trade studies, simulations, and analysis, the increasing complexity of modern systems demands higher levels of abstraction and design. It is now crucial for the systems engineer to adopt OO techniques in his software, not only to manage the increasing complexity, but also to bridge the gap between the systems and software engineering world, whereby both types of engineers are capable of viewing the system using the paradigm of object oriented design.

From the design and implementation of the simulation program for this report, it is clear that the systems engineer can evolve from structured to object oriented design without forsaking the mathematical analysis tools he depends on. The systems engineer need not transition to traditional OO languages such as C++ and Java, which lack comprehensive mathematical toolboxes, nor must he forsake OO capabilities by using non-object oriented mathematical analysis software. The solution, as demonstrated in the implementation of the simulation, is to use MATLAB as the development platform of choice for systems engineers in designing simulations, models, and analysis software.

MATLAB is clearly capable of handling the signal processing, numerical analysis, mathematical algorithms, graphing, and plotting required by modern mathematically intense simulations. With version 5, MATLAB now offers the capability of object-oriented programming. Since MATLAB is historically a structured language, however, it is a concern as to its object oriented prowess. Can software written in MATLAB closely resemble OO software written in C++ or Java? Is MATLAB true to object-oriented design or were unacceptable compromises made? The following sections answers these questions and evaluates the success or failure of MATLAB as a mathematically rich, object oriented capable programming language.

7.1 Encapsulation in MATLAB

An important feature of any object oriented programming language is encapsulation. The ability to encapsulate methods and properties within objects reduces software complexity. Restricting communication with an object to its interface methods shields object's properties from being improperly set. If code is modified, encapsulation limits the number of potential knock-on effects to other parts of the program and reduces the amount of coupling throughout the software.

MATLAB does an excellent job of implementing encapsulation. A class is built by creating and storing its methods in a separate subdirectory. Within a class subdirectory, a constructor must be created that initializes class properties to some known value. In MATLAB, properties are initialized at runtime as opposed to compile time. As in C++ and Java, MATLAB allows various arguments to be passed to the constructor, providing control over what values an object's properties are initialized to. An object's properties cannot be directly outside of the object itself. In this respect, MATLAB is more true to encapsulation than some other OO programming languages. Whereas in C++ object properties can be directly accessed through the tricky use of pointers, in MATLAB object properties must be accessed through the object's interface methods.

Methods within an object can be made public or private. Public methods can be called from outside the object. As in C++, public methods within a class have the permission to modify, change, or

retrieve properties from other objects of the same class type. Logic can be embedded within class methods to ensure that an object's properties are correctly accessed. Private methods may also be created for an object. As in C++, private methods can be called only by other methods within the same class. They can be called by any other method defined within the class directory, but not from the MATLAB command line or by methods outside of the class directory, including any parent methods.

One advantage of MATLAB over C++ is that a destructor does not have to be called to "clean up" memory after objects are finished executing. MATLAB is also capable of automatically managing memory allocation for arrays, matrices, vectors, and any other data type desired.

7.2 Function and Operator Overloading in MATLAB

A capability present in some OO languages is function and operator overloading. While function and operator overloading are not required for a language to be "object oriented", they are usually considered as desirable features to have. With function overloading, methods may be "overloaded" to perform different tasks depending upon the type of arguments passed to the function. For example, separate "add" functions may be created that perform either an arithmetic sum or a string concatenation depending upon whether numbers or strings are passed in as arguments.

MATLAB does a good job of implementing function overloading. An added benefit of MATLAB is that most built in functions can be overloaded.. This means that the plot, FFT, transpose, or any other MATLAB function can be overloaded to perform a more specific task defined by the user. Function overloading proved especially useful in the simulation program in that MATLAB'S built in "surf" function, which draws three dimensional surface plots, was overloaded to draw either a target or a unique shape such as a cylinder, sphere, or facet, based upon the type of object passed to the surf function.

There are a couple major differences between MATLAB and C++ in regards to function overloading. Whereas in C++ method dispatching is syntax based, in MATLAB, when the argument list contains objects of equal precedence, the left-most object is used to select the appropriate method to call.

This means that within a function, conditional logic statements must sometimes be used to determine the type and number of arguments passed in and to choose the proper code stream to execute. The need to use conditional logic to implement certain types of function overloading proved to be more unwieldy than the more efficient method of function overloading offered in C++.

In addition to function overloading, MATLAB provides operator-overloading capabilities. Operator overloading is the ability to override built in operators like '+', '-', '/', '*', and '()' to perform user defined tasks. This again proved useful in the simulation program whereby the '.' Operator was overloaded to "set" or "get" an object's properties in a more readable manner. To get the value of the radius property within a cylinder object, for example, with an overloaded "." operator, one can merely type "radius = cylinder.radius". As with any other set or get method, the overloaded dot operator contains the proper logic to ensure the object's properties are properly accessed.

One issue that is often associated with operator overloading is determining the operator precedence within an operation. For example, consider the expression objectA + objectB where the "+" operator has been overloaded to perform some user defined operation. Usually for this scenario MATLAB assumes the objects have equal precedence and calls the method associated with the leftmost object first. There are however two exceptions (Mathworks, 2000). First, user-defined classes have precedence over built–in MATLAB classes. Second, user-defined classes can specify their relative precedence with respect to other user-defined classes using the inferiorto and superiorto functions. While these two functions help manage the order of operations when different objects are used, it can become confusing very quickly. When implementing operator overloading, it is typically best within MATLAB to keep expressions as simple as possible in order to prevent any unnecessary confusion regarding operator precedence.

It should be noted that function and operator overloading are not considered by all to be a "desirable feature" to have. For some, function and operator overloading is considered a dangerous practice that increases the potential for bugs and adds complexity to code. While overloading provides

for slicker more elegant code, it can cause confusion if not implemented carefully. For any software project, it should be clearly decided upon front whether or not operator overloading should be used.

7.3 Inheritance in MATLAB

Inheritance is another key feature of object-oriented programming. With inheritance hierarchies of parent and children classes may be created. A child class can inherit properties and methods from one parent or from many parents (single of multiple inheritance). Inheritance can span one or more generations and enables the sharing of common functions and enforces common behavior among all children classes.

MATLAB provides inheritance capabilities similar to the C++ language. In MATLAB, children can inherit methods and properties from parents. The child object includes all of the properties present in its parent's class and can use all of its parent's public methods. While the parent can access properties the child inherits, it cannot access new properties or methods created within the child itself. This implies that children have the same properties and methods as their parent, plus any additional properties and methods they might require. Methods associated with the parent can operate on child objects, but methods associated with children cannot operate on objects belonging to their parent's class. This means that for a child to access properties belonging to its parent, it must work through the methods provided by the parent class. It should also be noted that unlike in C++, in MATLAB there is no such thing as a "protected" method.

In creating child objects, MATLAB requires all properties inherited and created within the child to be initialized. In C++ and JAVA, this is done through the child object's constructor. In these traditional OO languages, the child constructor knows to automatically call its parent's constructor to properly initialize inherited properties. Once the parent's constructor is finished, control returns to the child constructor where any new properties defined by the child object are initialized. In MATLAB, a similar procedure is used whereby the child constructor calls its parent's constructor to initialize any inherited properties. In MATLAB, however, the call to the parent constructor must be explicitly made

and is done after the child's properties are initialized. Since logic is sometimes required to implement "function overloading" within MATLAB, care must be taken to ensure the parent constructor is called correctly for all possible branches of execution within the child's constructor.

MATLAB provides for both single and multiple inheritance. In single inheritance, objects inherit characteristics from one parent class. In multiple inheritance, objects inherit characteristics from multiple parent classes. While multiple inheritance is a powerful feature, many software engineering experts recommend against its use due to the added complexity and potential for bugs. One particular problem with multiple inheritance is when a child inherits similarly named methods from multiple parents. From which parent will that method be inherited? MATLAB solves this problem by inheriting the method from the parent whose constructor is called first from the child's constructor.

7.4 Aggregation in MATLAB

In addition to inheritance, MATLAB provides support for aggregation within objects. Any MATLAB object can contain another object as one of its properties. In the simulation program, for example, the target object contained an array of shape objects such as fins, frustums, cones, discs, and cylinders. In MATLAB, methods for an object encapsulated in another object can only be called from a method within the container object. Objects can also be passed as arguments to other objects and returned as outputs from methods.

7.5 Polymorphism in MATLAB

No object oriented programming language would be complete without the ability to do polymorphism. Polymorphism is the ability of an object to automatically select the correct method to invoke at run time. In the simulation, polymorphism was used to automatically choose the correct intersection algorithm to use for ray tracing based upon the type of shape encountered. Polymorphism was also used to "draw" three dimensional perspective drawings for the different shape types.

MATLAB supports polymorphism, with some slight differences from C++ and JAVA. When looking for the appropriate method to call, MATLAB follows specific rules in determining function

precedence. These rules allow for polymorphism to be implemented, whereby there exists many methods with the same name located in their various class directories, and based on the type of object passed to a function, the appropriate method is used. For example, in the simulation program a display routine was formulated, whereby the values for all the properties from an object are output. A display routine was written for the target, shape, facet, cylinder, frustum, sphere, and cone classes. If the display method is invoked on a target, the target object's properties are automatically displayed. If invoked on a shape, only those properties from the shape object are displayed. If invoked on the sphere, the properties unique to the sphere class, along with the properties inherited from its parent shape class, are displayed.

One negative aspect of MATLAB'S method for implementing polymorphism and function overloading is that it is somewhat confusing. MATLAB uses the following rules to determine function precedence:

For built-in functions the precedence order is as follows as taken from the MATLAB user manual:

1. Overloaded methods: If there is a method in the class directory of the dispatching argument that has the same name as a MATLAB built-in function, then the method is called instead of the built-in function (Mathworks, 2000).
2. Non-overloaded MATLAB functions: If there is no overloaded method, then the MATLAB built-in function is called (Mathworks, 2000).
3. MATLAB built-in functions take precedence over both sub-functions and private functions. Therefore, sub-functions or private functions with same name as MATLAB built-in functions will never be called (Mathworks, 2000).

For functions not built-in to MATLAB

1. Sub-functions: Sub-functions take precedence over all other M-file functions and overloaded methods that are on the path and have the same name. Even if the function is called with an argument of type matching that of an overloaded method, MATLAB uses the sub-function and ignores the overloaded method (Mathworks, 2000).
2. Private Functions: Private functions are called if there is no sub-function of the same name within current scope. As with sub-functions, even if the function is called with an argument of type matching that of an overloaded method, MATLAB uses the private function and ignores the overloaded method (Mathworks, 2000).

3. Class Constructor Functions: Constructor functions take precedence over other MATLAB functions. Therefore, if you create an M-file called polynom.m and place it on the path before the constructor @polynom/polynom.m, MATLAB will always call the constructor version (Mathworks, 2000).
4. Overloaded Methods: MATLAB calls an overloaded method if it is not masked by a subfunction of private functions (Mathworks, 2000).
5. Current Directory: A function in the current working directory is selected before one elsewhere on the path (Mathworks, 2000).
6. Elsewhere On Path: Finally, a function anywhere else on the path is selected (Mathworks, 2000).

Clearly MATLAB'S method of implementing polymorphism through function precedence is pretty confusing. Given the legacy of MATLAB as a structured language, however, their techniques of implementing polymorphism and inheritance are understandable. If one is careful about placing MATLAB functions in their proper directory location, the whole hierarchy of function precedence should be transparent. MATLAB does provide the *which* command to determine the precedence of existing methods.

7.6 MATLAB as the Solution

From the experience of designing, coding, and executing the simulation program, it is clear that MATLAB is a possible solution for bridging the gap between the algorithmic world of the system's engineer and the object-oriented world of the software engineer. MATLAB is one of the premier software tools for implementing advanced mathematics, signal processing, image processing, and statistical analysis. Its graphing capabilities are a real boon to the system's analyst. With the added capability of OO, MATLAB now offers the possibility of object-oriented design as well.

With OO, the simulations, trade studies, and models created by the system's engineer can now more closely mirror the software design of object oriented system code. Even if the simulation or tool developed is not going to be implemented in a real system, the new OO capabilities of MATLAB allows the software tool to be implemented in a manner that saves rework, minimizes complexity, promotes readability, and maximize maintainability.

One must keep in mind that while the Mathworks did a decent job of adding OO capabilities to MATLAB, it is still not in the same league as C++ and JAVA as being a truly OO software language. Since MATLAB is historically a structured language, there were many "workarounds" that had to be made, including the determination of object precedence, the overloading of constructors and functions, and the many rules needed to determine function precedence.

While learning the new syntax for OOP in MATLAB is not difficult, it is recommended that one first have a basic understanding of OO concepts. No language can produce good OO code if the designer does not understand the philosophy behind object oriented design. At a minimum, to utilize the object oriented capabilities of MATLAB, the ideas of encapsulation, inheritance, aggregation, and polymorphism should be understood. With an understanding of these concepts, and a thorough understand of MATLAB, the system's engineer has at his command a truly powerful tool capable of architecting, designing, simulating, and analyzing today's modern hardware and software systems.

REFERENCES

Lee & Tepfenhart, UML and C++, a Practical Guide to Object-Oriented Development, 1997, Upper Saddle River, Prentice Hall, 1st Ed.

Wilkie, Object Oriented Software Engineering, 1993, New York, Addison Wesley, 1st Ed.

Anton & Rorres, Elementary Linear Algebra, 1991, New York, John Wiley & Sons, INC.6th Ed.

Horton, I. Beginning C++, 1998, Birmingham, Wrox Press Ltd, 1st Ed.

Oppenheim, A. Discrete-Time Signal Processing, 1989, Englewood Cliffs, Prentice-Hall, Inc.

Skolnik, M. Introduction to Radar Systems, 1980, New York, McGraw-Hill, Inc. 2nd Ed.

Rihaczek, A, & Hershkowitz, S., Radar Resolution and Complex-Image Analysis, 1996, Artech House, Inc. 1st Ed.

Stimson, G., Introduction to Airborne Radar, 1998, New Jersey, Scitech Publishing, Inc. 2nd Ed.

The Mathworks, MATLAB - The Language of Technical Computing, 2000, Natick, Mathworks, Inc.1st Ed.

Kim, J, Introduction to Radar Fundamentals, 2002, McKinney, Raytheon