NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

HIERARCHICAL TASK NETWORK PROTOTYPING IN UNITY3D

by

David Miller

June 2016

Thesis Advisor: Imre Balogh
Second Reader: David Reeves

This thesis was performed at the MOVES Institute
Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 2016	Master's Thesis 07-08-2014 to 06-20-2016

4. TITLE AND SUBTITLE

HIERARCHICAL TASK NETWORK PROTOTYPING IN UNITY3D

5. FUNDING NUMBERS

6. AUTHOR(S)

David Miller

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Operations Analysis Division of the Analysis Directorate, Deputy Commandant, Combat Development and Integration

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

The Combined Arms Analysis Tool for the 21st Century, or COMBATXXI, is the primary analytical combat simulation model in use by the Marine Corps' Operations Analysis Division (OAD) and the Army's Training and Doctrine (TRADOC) Analysis Center for weapon system and force effectiveness analysis.

The bottleneck in the COMBATXXI scenario production process is the behavior development process. Analytically useful scenarios demand complex and dynamic behaviors that react to the unique circumstances of the simulation's current state. Hierarchical Task Networks (HTN) are the state-of-the-art methodology in COMBATXXI used to describe dynamic behaviors. Although HTNs decrease the scenario development time, they are difficult to conceptualize, validate, and troubleshoot. The long iteration cycle is due, in part, to the complex development environment, the necessity of a large simulated infrastructure to test behaviors, and an inability to visually debug.

Here we present a solution for prototyping HTNs by extending an existing commercial implementation of Behavior Trees within the Unity3D game engine prior to building the HTN in COMBATXXI. Existing HTNs were emulated within this prototyping environment to test transferability of the behaviors, and new HTNs were prototyped in Unity3d prior to being built in COMBATXXI as a proof of concept. Prototyping HTNs in a 3D development environment may prove useful by reducing the iteration time and improving the overall quality of the behaviors. The interactive nature of Unity3d reduces the iteration time, and the ability to rapidly test many different cases improves the quality of the behaviors.

14. SUBJECT TERMS

hierarchical task network, HTN, dynamic behaviors, behavior prototyping, agent-based simulation, entity-level combat model, game engine, discrete event simulation, virtual environments

15. NUMBER OF PAGES 123

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

HIERARCHICAL TASK NETWORK PROTOTYPING IN UNITY3D

David Miller
Captain, United States Marine Corps
B.S., United States Naval Academy, 2009

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MODELING, VIRTUAL ENVIRONMENTS, AND SIMULATION (MOVES)

from the

NAVAL POSTGRADUATE SCHOOL
June 2016

Approved by: Imre Balogh
Thesis Advisor

David Reeves
Second Reader

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The Combined Arms Analysis Tool for the 21st Century, or COMBATXXI, is the primary analytical combat simulation model in use by the Marine Corps' Operations Analysis Division (OAD) and the Army's Training and Doctrine (TRADOC) Analysis Center for weapon system and force effectiveness analysis.

The bottleneck in the COMBATXXI scenario production process is the behavior development process. Analytically useful scenarios demand complex and dynamic behaviors that react to the unique circumstances of the simulation's current state. Hierarchical Task Networks (HTN) are the state-of-the-art methodology in COMBATXXI used to describe dynamic behaviors. Although HTNs decrease the scenario development time, they are difficult to conceptualize, validate, and troubleshoot. The long iteration cycle is due, in part, to the complex development environment, the necessity of a large simulated infrastructure to test behaviors, and an inability to visually debug.

Here we present a solution for prototyping HTNs by extending an existing commercial implementation of Behavior Trees within the Unity3D game engine prior to building the HTN in COMBATXXI. Existing HTNs were emulated within this prototyping environment to test transferability of the behaviors, and new HTNs were prototyped in Unity3d prior to being built in COMBATXXI as a proof of concept. Prototyping HTNs in a 3D development environment may prove useful by reducing the iteration time and improving the overall quality of the behaviors. The interactive nature of Unity3d reduces the iteration time, and the ability to rapidly test many different cases improves the quality of the behaviors.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1 Improving the Behavior Development Process **1**
1.1 Agent-based Simulations in the Marine Corps. 2
1.2 Complexity Theory in Simulations for Analysis 3
1.3 Analytic use of COMBATXXI 4
1.4 The Scenario Design Bottleneck 6
1.5 Commercial Game Development Best Practices 7
1.6 Unity3D as a Development Environment 8
1.7 Benefits of This Thesis 9

2 Achieving Dynamic Agent Behavior **11**
2.1 Hierarchical Task Networks 11
2.2 Behavior Trees 16

3 Managing Dynamic Behaviors in COMBATXXI **25**
3.1 Testing Behaviors in COMBATXXI 25
3.2 The Need for Robust Testing of Behaviors 29
3.3 Options for Improved Testing in COMBATXXI 30

4 Best Practices from Commercial Game Development **33**
4.1 Applicability of Game Development Techniques 34
4.2 Identifying Applicable Best Practices 38
4.3 Benefits of 3D Development Environments 39
4.4 Prototyping Models and Behaviors 40
4.5 Test Driven Development 43
4.6 Selecting the Most Applicable Methods 46

5 Unity as a Behavior Prototyping Environment **47**
5.1 Introduction to Unity 47
5.2 Scripting with Unity 53

5.3 Limitations of Unity 56
5.4 Unity as a Surrogate Development Environment. 57

6 Using Behavior Trees in Unity to Emulate HTNs 63
6.1 Utilizing the Prototyping Environment 63
6.2 Emulating COMBATXXI HTNs with Behavior Trees 65
6.3 Bounding Overwatch 68
6.4 Prototyping New Behaviors 70

7 Recommendations and Conclusions 77
7.1 Recommendations 77
7.2 Future Work 81

Appendix A The Prototyping Environment 85
A.1 Behavior Designer Behaviors 85
A.2 Prefab GameObjects 90
A.3 Scenes for Testing Basic Behaviors 96
A.4 Scenes for Implementing Existing Behaviors 97
A.5 Scenes for Building New Behaviors 98

Appendix B MonoBehaviour Script Lifecycle 99

List of References 101

Initial Distribution List 105

List of Figures

Figure 2.1 Five node types used in HTNs 13

Figure 2.2 An HTN to get to the store. 15

Figure 2.3 An HTN to make a sandwich. 16

Figure 2.4 Three major types of behavior tree composite nodes 17

Figure 2.5 Examples of behavior tree decorator nodes 18

Figure 2.6 Examples of behavior tree conditional nodes 18

Figure 2.7 Examples of behavior tree action nodes 19

Figure 2.8 GoToStore Implemented as a BT in Opsive's Behavior Designer 20

Figure 2.9 GoToStore HTN Overlayed onto BT 22

Figure 3.1 COMBATXXI logger selection user interface 26

Figure 3.2 3D Viewer tool in COMBATXXI 27

Figure 3.3 Distributed Interactive Simulation (DIS) enumerations entry in a COMBATXXI entity profile 28

Figure 3.4 Viewing the model output panel adjacent to the viewer during a simulation run. 28

Figure 4.1 Example of a simple Game Loop Pattern 36

Figure 5.1 A generic Unity editor configuration 51

Figure 5.2 The Unity pause, play, and frame-by-frame buttons 53

Figure 5.3 An HTN to get to the store. 58

Figure 5.4 The agent walks past the key within plain sight. 58

Figure 5.5 A behavior tree which sends an event after detecting an enemy. 61

Figure 6.1 Emulating a COMBATXXI replan event in a behavior tree . 67

Figure 6.2 Bounding overwatch behavior tree owned by the squad leader which makes the decision to conduct bounding overwatch or patrol along a direct route 69

Figure 6.3 Behavior trees owned by the squad leader and assistant squad leader to coordinate bounding overwatch 70

Figure 6.4 An overview of the Platoon Security Patrol provided by Operations Analysis Division (OAD). 71

Figure 6.5 Developmental behavior tree for the platoon security patrol scenario 73

Figure 6.6 A Unity scene built to prototype the Platoon Security Patrol scenario behaviors 73

Figure 6.7 Platoon Security Patrol scenario, implemented in COMBATXXI. 75

Figure 7.1 Developers may choose to build entire scenarios in the surrogate environment first, incurring the risk that some behaviors won't transfer. Ideally, this process is conducted once per scenario. 79

Figure 7.2 Developers may choose to frequently transfer existing behaviors to the target simulation in order to prevent excessive refactoring. This process is repeated as many times as is necessary, depending on the number of behaviors required. 79

Figure A.1 Type "custom" into the search bar of the task tab within the Behavior Designer editor to find all uploaded tasks built for this thesis. 86

Figure A.2 The "EmbarkVic" behavior tree loads a squad into a vehicle. 87

Figure A.3 The "EngageUntilDeadorGone" behavior engages the nearest enemy within range. 87

Figure A.4 The "DetectCloseWithEngage" behavior results in a search and destroy type of behavior after detecting an enemy. 88

Figure A.5 The "MoveToEngage" behavior moves the agent to within engagement range. 89

Figure A.6 The "DestroyWithinRange" behavior engages the nearest enemy within range. 90

Figure A.7 The class hierarchy of scripts attached to soldiers and vehicles. 91

Figure A.8 The "BlueSoldier_AgentState" prefab 91

Figure A.9 The class hierarchy of scripts attached to munitions 93

Figure A.10 A proximity IED which detonates when an enemy agent enters the empty box 93

Figure A.11 The red agent on the left is calling for fire, using the 60mm-Mortar prefab, while blue agents maneuver through the impact zone. 94

Figure A.12 The basic scene includes only terrain and several buildings. . 96

Figure A.13 The complex scene is larger than the basic scene, and includes a debugging user interface built into the game window. 96

Figure B.1 MonoBehaviour Script Lifecycle 99

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AI	Artificial Intelligence
AoA	analysis of alternatives
BT	Behavior Tree
CAS	close air support
CASTFOREM	The Combined Arms and Support Task Force Evaluation Model
CNA	Center for Naval Analysis
COTS	commercial off the shelf
CD&I	Deputy Commandant, Combat Development and Integration
COMBATXXI	Combined Arms Analysis Tool for the 21st Century
DES	Discrete Event Simulation
DIS	Distributed Interactive Simulation
DoD	Department of Defense
FDC	fire direction center
FSM	finite state machine
HTN	Hierarchical Task Networks
IDE	integrated development environment
IDF	indirect fires
IED	improvised explosive device
ISAAC	Irreducible Semi-Autonomous Adaptive Combat
LOS	line of sight

MAGTF	Marine Air Ground Task Force
MANA	Map Aware Non-uniform Automata
MCCDC	Marine Corps Combat Development Center
MEU	Marine Expeditionary Unit
MOVES	Modeling, Virtual Environments, and Simulation
M&S	Modeling and Simulation
NPS	Naval Postgraduate School
OAD	Operations Analysis Division
OOP	object-oriented programming
PDU	Protocol Data Unit
SISO	Simulation Interoperability Standards Organization
SITS	Scenario Integration Tool Suite
SME	subject matter expert
STRIPS	Stanford Research Institute Problem Solver
TDD	Test Driven Development
TRAC-WSMR	Training and Doctrine Command Analysis Center, White Sands Missile Range
TRADOC	Training and Doctrine Command
UI	user interface
USMC	United States Marine Corps
VS	Visual Studio
V&V	verification and validation

Acknowledgments

This work would not have been possible without the help of several groups of individuals. First of all, the MOVES faculty deserves my gratitude for taking a group of Marines and getting us up to speed in the complex and changing world of modeling and simulation in such a short time. Thanks specifically to Dr. Imre Balogh, Curtis Blais, Kirk Stork, and David Reeves. Their technical expertise, historical perspective, and willingness to provide a novice with expert guidance in the realm of COMBATXXI were instrumental in the guidance, conduct, and completion of this thesis.

The valuable guidance received from the COMBATXXI group within the Marine Corps' Operations Analysis Division helped in understanding the benefit to the Marine Corps. I look forward to working with you all.

Without the support of a family that values education, the labors involved in pursuit of an advanced degree may seem idle. Thanks to my parents for instilling in me the value of education, and for supporting me through this process. The most important thanks goes to my wife, Maiah. Without her support I could not have made it through this program. Her encouragement and willingness to happily let me work away while taking care of our two newly born baby girls will never be forgotten. To Blair and Ivy, thanks for not screaming too loudly while I was working!

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Improving the Behavior Development Process

The Combined Arms Analysis Tool for the 21st Century (COMBATXXI) is the primary multiagent-based simulation tool in use by the Marine Corps' Operations Analysis Division (OAD), under the purview of the Deputy Commandant for Combat Development and Integration (CD&I), as well as the Army's Training and Doctrine Command (TRADOC) Analysis Center. Analytically useful scenarios demand complex and dynamic behaviors which allow agents to react to the unique circumstances of the simulation's current state. Hierarchical Task Networks (HTN) are the state-of-the-art methodology in COMBATXXI used to describe dynamic behaviors. Although HTNs greatly decrease the scenario development time, they are difficult to conceptualize, validate, and troubleshoot. The long iteration cycle is due, in part, to the complex development environment, the necessity of a large simulated infrastructure to test behaviors, and an inability to visually debug.

In this thesis, we suggest a construct for prototyping HTNs by extending an existing commercial implementation of Behavior Trees within the Unity3D game engine prior to building the HTN in COMBATXXI. Prototyping HTNs in a 3D development environment may prove useful by reducing the iteration time and improving the overall quality of the behaviors. The iteration time is reduced due to the interactive nature of Unity3D and the availability of low-cost models. The quality of the behaviors is improved due to the ability to rapidly test many different cases.

This chapter provides a historical and contextual overview of agent-based simulations in the Marine Corps analysis community, and a look ahead at the chapters to follow.[1]

[1]We recognize that varying definitions of "agent-based" simulation exist in the Modeling and Simulation (M&S) community. Our interpretation is that of a simulation which revolves around sensing and reacting agents, which some have called "entity-level combat simulations" [1, p. 608]. These models "represent individual combat vehicles (e.g. tanks or helicopters), platforms (e.g. radar sets), and personnel (e.g. infantry squads or soldiers) as distinct simulation entities. Necessary state information, such as location, velocities, and ammunition load, is maintained for that entity separately and each entity is capable of independent action." [1]

1.1 Agent-based Simulations in the Marine Corps

Agent-based simulations provide a framework in which analysis can be conducted without the limiting assumptions made in many conventional analytic models. In recent decades, the computational power available to simulation developers has increased many times over, and simulations are becoming a primary analytic tool rather than "a method of last resort" [2, p. 294]. A primary the benefit of multi-agent based simulations of combat is the agent-based behaviors, regardless of whether they are explicitly scripted or dynamic in nature.

The Marine Corps has embraced the analytic potential of agent-based simulations since their infancy, most famously in the study titled *Complexity & Combat*, sponsored by Lieutenant General Van Riper [3]. Currently, this banner is carried by the Marine Corps' Operations Analysis Division (OAD), within the Analysis Directorate of the Deputy Commandant, Combat Development and Integration (CD&I). The primary mission of OAD is to provide analytic support to decision makers "about weapon systems, equipment acquisition and resource allocation" [4]. One way in which this is achieved is through the use of agent-based combat simulations, such as the COMBATXXI. COMBATXXI is a "high-resolution, closed-form, stochastic, discrete event, entity level structure analytic combat simulation", which focuses on land-based combat with some littoral and amphibious operations support [5].

COMBATXXI is not the first agent-based simulation that the Department of Defense (DoD) has used to support its analysis.[2,3] However, COMBATXXI and its predecessors alike have been guided by the Marine Corps' philosophy on combat and warfare at large. The primary bearer of the high-level philosophy on warfare for the Marine Corps is the Marine Corps Doctrinal Publication 1 (MCDP-1), titled *Warfighting*, which states:

"War is not governed by the actions or decisions of a single individual in any one place but emerges from the collective behavior of all the individual parts in the system

[2]The Combined Arms and Support Task Force Evaluation Model (CASTFOREM), for example, is the predecessor to COMBATXXI. Details of CASTFOREM can be found in [3, p. 42] and here: http://www.simscript.com/solutions/military/CASTFOREM.html

[3]An overview of previously existing combat simulations can be found in section 1.4 of Ilachinski's *Artificial War* [3, p. 39].

interacting locally in response to local conditions and incomplete information. A military action is not the monolithic execution of a single decision by a single entity but necessarily involves near-countless independent but interrelated decisions and actions being taken simultaneously throughout the organization." [6]

The Marine Corps defines war as a violent clash of wills, wherein each opponent is not a single or homogeneous entity, but a collection of independent actors [6]. It is this philosophy of war that drives the Marine Corps' outlook on analytic simulations, and makes its analysis community ever more inclined to the use of multi-agent based simulations. It is with this philosophy in mind that this thesis aims to improve upon the behavior-based methods which make those simulations useful.

This chapter provides a brief historical overview of the emergence of dynamic behaviors within analytic simulations used by the United States Marine Corps (USMC) and the underlying motivation for further research in this area.

1.2 Complexity Theory in Simulations for Analysis

In Dr. Andrew Ilachinski's book titled *Artificial War: Multiagent-Based Simulation of Combat*, he details the history behind how the Marine Corps came to use multiagent-based simulations. From 1996-2003, the Center for Naval Analysis (CNA) directed a project called *Complexity & Combat* which was sponsored by the Commanding General of Marine Corps Combat Development Center (MCCDC), Lieutenant General Van Riper. The primary goal of the project was to determine the "general applicability of nonlinear dynamics and complex systems theory to land warfare." Thus the Marine Corps' relationship with complexity theory was born. Although the Marine Corps' philosophy of war is quite similar to complexity theory, no organizational efforts had been made prior to the CNA project to implement complexity theory within models of warfare. This advancement towards using complexity theory within models of warfare as a peer or replacement for traditional, attrition-based models, represents the aggressively revolutionary mindset of Lieutenant General Van Riper for which he is so well known.

1.2.1 Complexity Theory, Defined

Complexity theory, also known as complex systems theory, is the idea that "complicated systems can generate simple behavior." [3, p. 5] This concept applies to the study of complex organisms, including humans. According to Dr. Ilachinski, research in the area of complex systems theory can be summarized by two themes:

Complexity emerges from simplicity. This is the concept that relatively few and simple behaviors have the ability to result in seemingly complex individual or group behaviors. Perhaps the most famous example of this is the Boids simulation, first created by Craig Reynolds in 1986, in which complex flocking behaviors emerge as a result of three simple parameters: separation, alignment, and cohesion [7].

Simplicity emerges from complexity. This is the concept that relatively stable results arise from an otherwise complex model. This may occur through so-called "self-organization" of agents within the simulation, or through guided tuning by the scenario or behavior designer. An example may be a behavior at the appropriate echelon of leadership level which dictates that a unit reorganizes in a smaller perimeter as it takes casualties, in order to fill the gaps of its defense. This is the portion of complexity theory that agent-based simulations rely on, and this is the portion of complexity upon which this thesis is focused [3, p. 5].

It should be noted that agent-based simulations are only one tool which aims to answer research questions in the area of complexity theory. Readers are directed to Dr. Ilachinski's book for a more comprehensive list of tools that are used in this domain [3, p. 6].

1.3 Analytic use of COMBATXXI

COMBATXXI is the simulation tool currently used and jointly developed by the Marine Corps' OAD and the Army's TRADOC Analysis Center. COMBATXXI has many of the same goals as Dr. Ilachinski's original combat simulation tool, known as Irreducible Semi-Autonomous Adaptive Combat (ISAAC). Both tools aim to leverage the insights gained from complexity theory as it applies to military simulations for analysis. COMBATXXI is capable of modeling an entire Marine Air Ground Task

Force (MAGTF) of various sizes, including manned and unmanned aviation assets, complex land-based combat operations, and amphibious operations. There is support for simulating urban and asymmetric combat as well as more conventional combat operations. It provides simulationists with a wide range of options for scenario development, and additional modules can be developed to suit the requirements of specific studies. COMBATXXI is the focus of this thesis because it is the primary simulation used by OAD. All succeeding discussions regarding "simulations" and "models" can be assumed to pertain most specifically to COMBATXXI.

1.3.1 The Role of Behaviors in Agent-Based Simulations

Ilachinski's latter theme of complexity theory states that simplicity has the potential to emerge from complexity. The clarity of analysis which we seek from multi-agent based simulations of combat aims to leverage this theme. Due to the closed-loop nature of COMBATXXI, simulationists must have the ability to manually script behaviors or build dynamic behaviors which can determine appropriate actions at runtime.[4] These behaviors must sufficiently emulate doctrinal military behaviors in order to be valid. In practice, current methodologies embody both scripted and dynamic behaviors. For example, the observation process in COMBATXXI is dynamic in that it is always running, once activated, and requires no further explicit commands from the simulation developer. On the other hand, maneuver is typically scripted in a manual fashion by stringing together way-points into routes which the agents explicitly follow.

Maneuver-related behaviors may also be dynamic. For example, a dynamic behavior may measure which route is the fastest and provides the most cover from potential enemy observations and fields of fire. These behaviors, be they dynamic or scripted, are a primary enabler for multi-agent based simulations of combat. This thesis examines ways in which the development methods for the latter type of scenario-specific dynamic behavior may be improved upon.

[4]Scripted behaviors are those which do not change based on varying stimuli, while dynamic behaviors will adjust based on different input according to their programmed algorithm.

1.4 The Scenario Design Bottleneck

Manual scripting of behaviors, such as those that dictate maneuver routes, is a tedious process that gets more complicated whenever its scope is expanded in space, time, or number of agents. While there are many potential bottlenecks in the design of COMBATXXI scenarios, the primary limiting factor is the development of scripted agent-based behaviors which adequately emulate military doctrine and robustly apply that doctrine to a wide variety of situations.

1.4.1 Refactoring without Dynamic Behaviors

Refactoring scenarios in COMBATXXI without the use of behaviors which dynamically adjust to the new scenario can be excessively wasteful. Even small changes can have drastic ripple effects throughout the simulation, leading to adversely large developer hour investments for seemingly trivial changes, particularly later in the development process. For example, if a COMBATXXI scenario is being built to support an analysis of alternatives (AoA), the scenario designer may be tasked with swapping out different types of a certain armored vehicle. Perhaps one variant has a longer range primary weapon system, and therefore would take a different route in open terrain in order to leverage this advantage. The scenario designer would have to build a new set of routes for each armored vehicle in the scenario. This is where the advantage of dynamic behaviors lies.

Dynamic behaviors typically have parameterized input which allows for ease of refactoring. For example, consider a formation which has been manually built to include 5 meters of dispersion for a simulation scenario in open terrain. If the scenario manager wants to examine how the outcome changes based on dispersion, several different formations would have to be manually built. A dynamically programmed formation may include a parameter for dispersion, allowing for that same formation to be useful in many different scenarios and to be easily changed within a scenario. Refactoring scenarios becomes significantly easier with dynamic behaviors.

1.4.2 COMBATXXI Dynamic Behavior Development

Despite the significant positive benefits that can be achieved from the use of dynamic behaviors within COMBATXXI, they are often difficult to conceptualize, validate, and troubleshoot. The long iteration cycle is due, in part, to the complexity of the development environment, the necessity of a large simulated infrastructure to test behaviors, and an inability to visually debug in real-time. A detailed discussion of dynamic behavior development and testing in COMBATXXI will follow in chapter 3.

1.5 Commercial Game Development Best Practices

Many practices from the commercial game industry have the potential to benefit the development of agent-based behaviors for simulations. This work draws inspiration from those practices for a variety of reasons. For example, both domains reward software optimization, code re-use, and the ability to quickly refactor or extend existing code. The goals of game design may be to maximize a customer's interest while minimizing the computational cost of running the game on a variety of platforms [8, p. 155]. Similarly, simulation software must be computationally tractable, albeit for a different purpose. Instead of entertaining customers, simulationists aim to provide output data that is ready for consumption by military and civilian analysts. In order for this output to be statistically useful, it must be sufficient in quantity, so a simulation which takes too long to run will be of no use to anyone. These are just a few examples. Chapter 4 provides an overview of some best practices from the commercial game development industry which may prove beneficial to the development of dynamic behaviors for COMBATXXI.

1.5.1 The Impact of the Behavior Development Environment

The development environment is perhaps the most important tool for a scenario designer who is tasked with the development of dynamic behaviors. Due to the complex nature of these behaviors and the stimuli to which they respond, intuition and judgment alone are not sufficient to predict the resulting emergent behavior for a wide variety of use cases. User-friendly development environments allow designers to fit in more tests in a shorter amount of time, leading to more robust, stable behaviors.

The development environment should allow behavior designers to test their behaviors early and often, in order to prevent incorrect or unstable behaviors from creating complex bugs later in their life cycle [9, p. 237]. Section 4.2.1 details the qualities which the ideal behavior development environment should possess, as revised from previous work generally defining high-quality behaviors [10, p. 8].

1.5.2 Benefits of 3D Development Environments

While the attributes specified in section 4.2.1 are generic, a specific attribute of an efficient behavior development environment is the inclusion of a 3D model viewer. A modern, interactive 3D model viewer leverages the "kinetic depth effect", which conveys 3D information to a user through a 2D interface–the computer screen [11]. It is not only intuitive, but has been shown in user studies that an interactive 3D environment allows users to better understand spatial relationships between objects [12].

The effect of a 3D model viewer on the resulting quality of behavior development has not been studied in depth. However, we submit that the demonstrated ability of subjects to understand models better with 3D information translates to a better understanding of a model or agent's behavior in a simulation. This then leads to a more thorough understanding of the implications of details within the behavioral logic, and therefore a better development process.

1.6 Unity3D as a Development Environment

Game engines are growing in popularity due to their accessibility, affordability, and ease of authoring. There are large online communities which provide guidance, starter code, and support to anyone with a basic understanding of programming and the willingness to spend some time learning the nuance of that particular game engine. Unity3D in particular is free to use for small projects, and does not even require the payment of royalties for small productions. [5] Furthermore, the Unity Asset Store provides access to a plethora of low-cost models and development tools which help build artificially intelligent behaviors.

[5]Further details on the business model used by Unity can be found here: https://unity3d.com/legal/eula

Unity3d was selected for purposes of this thesis due in part to its accessibility and the resident expertise within the Naval Postgraduate School. However, there may be other game engines which are also worth exploring. This thesis is a proof of concept, and does not serve as a singular solution for development of dynamic behaviors. For purposes of conciseness, Unity3D will be referred to as Unity.

1.6.1 Behavior Designer

The tool selected for the development of Behavior Tree (BT)s in Unity is Behavior Designer, version 1.5.5. Behavior Designer is a commercial off the shelf tool built by a small group of developers and sold in the Unity Asset Store. It is a BT authoring tool that ships with a large set of task nodes which can be used to build relatively complex behaviors without building any new tasks. It is authored in C#, which is the most common form of scripting in Unity.

Other tools, such as Rain AI, were examined, but Behavior Designer was determined to be the most suitable for several reasons. The primary reason is that the source code is available upon purchase–enabling behavior designers to fully understand, debug, and extend the shipped software. Behavior Designer also provides an authoring tool which eases the authoring of advanced behaviors from the modular compilation of simple behaviors.

1.7 Benefits of This Thesis

From a practical perspective, this thesis explores a specific methodology for developing a specific type of artificial intelligence for agents in COMBATXXI. It seeks to answer, in part, the Naval Research Program's Broad Area Study NPS-N16-M159, which urges researchers to "examine the gaming industry for innovations below the individual application level that are applicable to the DoD modeling and simulation community." This thesis focuses on the second potential research question, which asks "What technology advances does the commercial game industry implement that can be leveraged by USMC modeling and simulation programs of record?"

Ultimately, the primary goal of this thesis is to explore the benefit to agent behavior designers in using a 3D prototyping environment for the development of their agents'

behaviors. This is demonstrated using Unity3D as a prototyping environment for behaviors that will be ported for use in COMBATXXI.

CHAPTER 2:
Achieving Dynamic Agent Behavior

Artificially intelligent, or dynamic behavior systems fall into two major categories: planners, or reactive. Planners are used as a means of abstracting the deliberate human planning process prior to an agent taking action within a simulation, or game, and looks into the future to do so [13, p. 1]. Planners take into account the world state and the methods with which they can change the world state in order to achieve a goal. The process of determining which methods to use to achieve the goal is known, in a formal sense, as planning. Often the plan is represented as a stack of tasks, or methods, to use in order to achieve the higher level goal. Contrastingly, reactive AI systems do not plan ahead, and only choose the most appropriate action given the current world state.

These two broad categories encompass many different specific types of Artificial Intelligence (AI) systems, of which there are too many to list here. Their implementations vary depending on the application domain, and the *AI Game Development* website provides a good overview of those used in the commercial game industry [14]. This chapter examines only a few AI methods that are either currently used in COMBATXXI or could be useful for emulation of techniques employed within COMBATXXI.

2.1 Hierarchical Task Networks

Hierarchical Task Networks (HTN)s decompose a high-level task into a set of subtasks until no further decomposition is needed. This decomposition results in a stack of primitive tasks which, when executed, is expected to achieve the high-level goal of the HTN. Their popularity is due, in part, to their task decomposition method of planning, which is similar to human planning.

HTNs, as they are typically described in academic sources, assume that the state of the world will not change in a way that will impact the plan unless changed by the agent [13, p. 229]. This allows for the entire plan to be created, from start to

finish, without concern of a condition which enables a primitive task later becoming invalid. This basic assumption is likely to cause problems in a computer simulation or game in which many agents are affecting the world state during the execution of a plan. Therefore, HTNs as implemented in commercial video games and analytic simulations tend to include a way to allow the agent to replan to avoid this problem. This methodology was first introduced to simulations for analysis by work conducted at the Naval Postgraduate School (NPS) [15].

2.1.1 HTNs in Video Games

HTNs have become somewhat commonplace in video games over the last ten years as a means of achieving more realistic AI. The first major title to implement HTNs was *Killzone 2* in 2009, although Stanford Research Institute Problem Solver (STRIPS)-like planners were introduced as early as 2005 with *F.E.A.R.* Planners were found to be more efficient to manage than reactionary AI mechanisms, such as finite state machine (FSM)s, due to their modularity and readability. Even large HTNs are easy to read, and small changes don't have the wide-reaching effects on the behavior like they have a tendency to do with large FSMs [16].

HTNs are only one type of planner which have made their way into the commercial game development industry from academia and elsewhere. The other major type of planner, developed at Stanford in 1971, is called STRIPS [17]. Planners in games often deviate from the academic definition, but most can be categorized as either HTN-based or STRIPS-based [14].

2.1.2 HTNs in COMBATXXI

Traditional HTNs assume that the world state will only be changed by the planning agent. A planning system that only accounts for the agent's actions is unlikely to be useful in the context of complex agent-based simulations. This is a well-known shortcoming of classic planning architectures in general. Ghallab et al. state that "a more realistic model interleaves planning and acting, with plan supervision, plan revision, and replanning mechanisms" [13, p. 9]. This is precisely the intent behind the replan triggers in the HTN implementation in COMBATXXI. The term HTN

will be used from this point forward to describe HTNs as they are implemented in COMBATXXI, unless otherwise noted.

2.1.3 Anatomy of an HTN

HTNs are represented as directed acyclic graphs composed of five types of nodes representing the following concepts: compound tasks, constraints, primitive tasks, goal tasks, and interrupt goal tasks [15], [18]. These fives task types are depicted in figure 2.1.

Figure 2.1: Five node types used in HTNs. Adapted from [15].

Compound Task

Compound task nodes contain no task-oriented code and are used for syntactic structure only. They may contain code which gathers information for use in following, subordinate nodes, but this is not necessary. Compound task nodes represent a higher-level task which needs to be broken down into primitive tasks for execution [15, p. 2].

Constraint

A constraint node directs the flow of execution within the HTN by the use of conditional logic. These nodes execute some code which resolves to a Boolean. The child nodes of the constraint node are executed only if the Boolean resolves to a value of true.

Primitive Task

Primitive tasks contain code representing some action, and require no further decomposition. Primitive tasks are added to the agent's task stack to be scheduled as any other event in COMBATXXI.

Goal Task

Goal tasks are a type of primitive task which, when finished, represent the goal which the HTN has attained. They "correspond to the achievement of the desired world state," for which the HTN was built [15]. The execution of the HTN ceases after completion of the goal task.

Interrupt Goal Task

Interrupt goal tasks are specialized goal tasks which contain actions that are expected to take some time to execute, and will likely change the world state in some way that may require the HTN to be re-evaluated. The execution of the HTN is stopped until the actions in the interrupt goal task is completed. This prevents the entire HTN from being evaluated when it can be reasonably assumed that the conditions will change before the rest of the HTN is executed. This method is necessary for the scalability of HTNs within combat simulations, such as COMBATXXI, which may have thousands of agents using HTNs at the same time [15, p. 3].

2.1.4 Replanning with HTNs

COMBATXXI is an event-driven simulation, and the events provide possible points at which the HTN planner could be run again. Due to the high number of agents whose behavioral processes need to be modeled in a combat simulation, replanning at each event would be computationally wasteful [15, p. 3]. Furthermore, high-frequency replanning could lead to the same sort of oscillating bugs that are a weakness of reactionary AI systems [19]. However, not replanning at all, like academic HTNs, would produce unsuccessful plans and unrealistic behaviors. Deciding what middle ground to achieve when implementing HTNs is clearly an important factor to their success.

A primary benefit of the implementation of HTNs in COMBATXXI is their ability to replan based only on events which will affect the plan. This is accomplished by the HTN designer identifying "replan events" which the agent will listen for, and replan only when those events occur. Replan events are typically used in conjunction with "interrupt goal tasks," which pause the planner until execution up to that point has

finished. While "interrupt goal tasks" pause the tree to prevent planning too far into the future, they also identify points at which replanning will likely be necessary.

2.1.5 GoToStore: An Example HTN

As an explanatory example, we present an application agnostic scenario adjusted from [15], in which an agent is attempting to move from his home to the store. The scenario consists of three locations: the home, the store, and all points in between, and two objects: a car and a set of keys. The agent is at one of the locations, and either has the keys or not. The HTN representing this scenario is displayed in Figure 2.2.

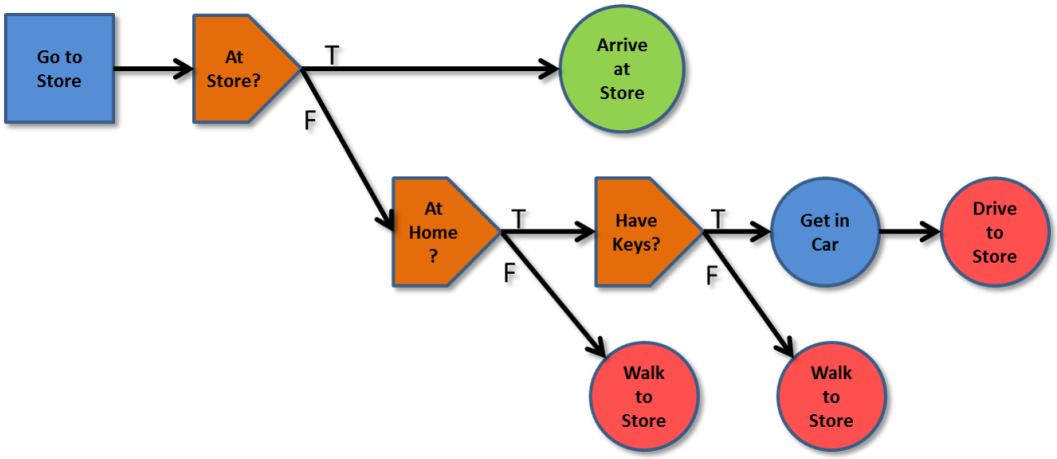

Figure 2.2: An HTN to get to the store.

In this scenario, the agent first checks if he is already at the store. If so, then the goal node “Arrive at Store” is executed, and the HTN is complete. Otherwise, the agent checks to see if he is at home. If he isn’t, he walks to the store, thereby executing an interrupt goal task which pauses the rest of the tree until completion of this task. Upon completion of the “Walk to Store” interrupt goal task, the agent re-evaluates from the leftmost node, realizes he is at the store, and the HTN is again complete. If the agent started at home, he checks to see if he has the keys to the car. If so, he gets in the car, and drives to the store. Like the “Walk to Store” node, the “Drive to Store” node is also a goal interrupt task, which pauses the tree until its completion. At this point, the agent is at the store, and the HTN is complete.

The "Go To Store" HTN is built to achieve the high-level goal of getting to the store, and can stand alone in its current form. However, it could also be included within a more complex HTN as an interrupt goal task node, perhaps when the agent is out of bread and is trying to make a sandwich. An example is shown in Figure 2.3.

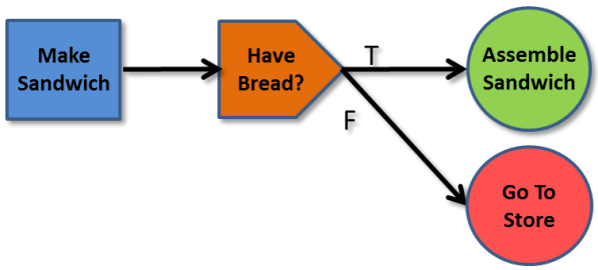

Figure 2.3: An HTN to make a sandwich.

It is in this way that the modularity of HTNs can be leveraged to create increasingly complex behaviors. Yet if we need to change the logic of the "Go To Store" portion of the behavior, we only need to change its HTN. This is a maintainability advantage over FSMs, and part of the reason that HTNs have been widely adopted.

2.2 Behavior Trees

Behavior trees (BT) are a means of achieving artificially intelligent behaviors popularized in the video game industry. Their expressiveness, extensibility, and ease of use have led them to gain popularity over other, less extensible methods such as FSMs [20, p. 159]. Although the exact origin of behavior trees is unclear, the first mainstream game in which behavior trees were used was Halo 2, which was released in 2004 [21].

2.2.1 Behavior Tree Basics

BTs are composed of nodes and the connective branches. Nodes represents either a task or structural syntax of the tree, and are classified as one of three general types: composites, decorators, and actions [22, p. 262]. Composites and decorators define the logic and syntax of the tree, while the actions run code which typically aims to change the game state. BTs are read from top to bottom and left to right, and use a depth-first logic. Every node in a BT either returns success or failure to its parent

node, even if it takes some time to execute. Traversal of a branch ceases when a node returns failure. BTs can be as expressive as the language in which they are implemented, and are inherently more modular than simple scripting alone [23].

The nodes discussed in this section are as general as possible, but apply most directly to the BT implementation in Opsive's Behavior Designer, version 1.5.5, in Unity version 5.1 [24].

Composites

Composite nodes determine the manner in which their child nodes will be executed. There are three primary types of composite nodes: sequences, selectors, and parallels, as seen in Figure 2.4.

Figure 2.4: Three major types of composite nodes. Source: [24].

- **Sequence nodes.** Sequence nodes execute their child nodes in left to right order, returning "success" back up the tree only when all child nodes are successfully completed. They will immediately return "failure" to their parent node if any children fail. In this way, they are similar to "and" logic, as all children must succeed for the parent sequence node to succeed.
- **Selector nodes.** Selector nodes also execute their child nodes in sequence, left to right, but return success as soon as any child returns success. In this way, they are most similar to "or" logic, as only one child needs to return success to the parent selector node in order for the selector node to itself return success.
- **Parallel nodes.** Parallel nodes are the exception to the "rule" that BTs are read from left to right, because parallel nodes execute their child nodes at the same time. They are similar to the sequence nodes in that they only return "success" when all child nodes succeed; otherwise they return "failure."

Decorators

Decorators are essentially composite nodes with only one child [22, p. 263]. Decorators do things like repeat a subordinate branch until a condition is met, or provide a way to interrupt the branch below it from another part of the tree. Examples of decorator nodes are shown in Figure 2.5.

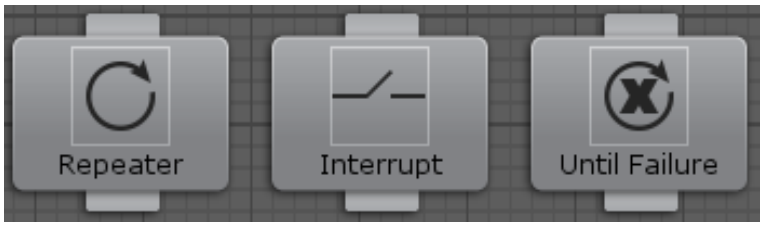

Figure 2.5: Examples of decorator nodes. Source: [24].

Conditionals

Conditionals restrict the flow of execution within a behavior tree in exactly the same way that if-else or switch-case statements do in traditional scripting. Their contents must eventually resolve to a Boolean. To be useful, they typically have a sequence as a parent and one or many action nodes as successors. This allows conditionals to act as gatekeepers to action-oriented nodes. Some basic examples of conditionals are shown in Figure 2.6.

Figure 2.6: Examples of conditional nodes. Source: [24].

Actions

Action nodes are the meat of BTs, as they are the only nodes which are meant to change the world state. The BT itself may have many nodes of complex logic in conditionals, decorators, and composites, but without action nodes, nothing would ever happen. The possibilities for action nodes are limited only by the programming language and the application domain, but typically describe tasks such as attack,

move, crouch, or send a message. They may also include tasks that change variables, store and represent knowledge, interrupt another task, or make calculations for other purposes. It is important for any implementation of BTs to include extension options for authors to build custom actions for their application.

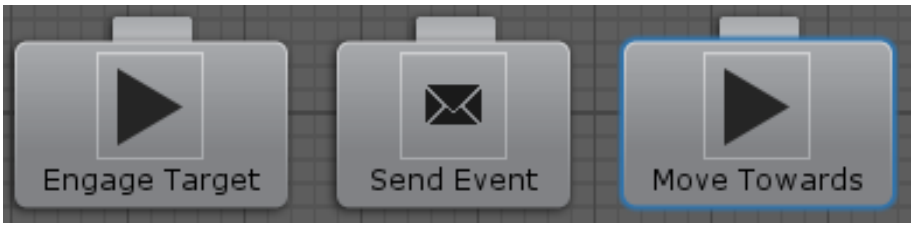

Figure 2.7: Examples of action nodes. Source: [24].

There is no limit on the code included within an action node. They may range from getting or setting a local variable to executing a complicated detection algorithm. This can be a potential pitfall, as nodes which represent complex tasks may be better broken down into several more simple nodes and then combined into a tree. They can then be inserted into the original tree not as a task, but as a behavior tree reference.

Behavior Tree References

BTs are inherently modular, meaning that they can be pieced together to make more complex BTs. Often this is accomplished with behavior tree references, where an entire BT is treated as a child node within another BT. This allows for efficient code re-use, because when the child BT referenced in the parent BT is changed, the parent BT does not have to be edited. In this way, behavior tree references leverage the same type of convenience and efficiency that is the cornerstone of object-oriented programming (OOP).

2.2.2 GoToStore as a Behavior Tree

This section details one possible implementation of the GoToStore HTN from section 2.1.5 as a behavior tree within Unity 5.1, using Behavior Designer version 1.5.5. Chapter 6 details the making of BTs in Unity; this tree serves only as a syntactic example. The objectives are to demonstrate through simple example the syntax of BTs and to provide an example suggesting that HTN-like constructs can be represented

as BTs. We will be using the basic components detailed in the previous section to walk through the BT shown in Figure 2.8, assuming the following starting state: the agent is at home and has the keys to his car. The initial execution of the tree can be seen by following the trace described by the numbered nodes in the image, which is detailed in the following section.

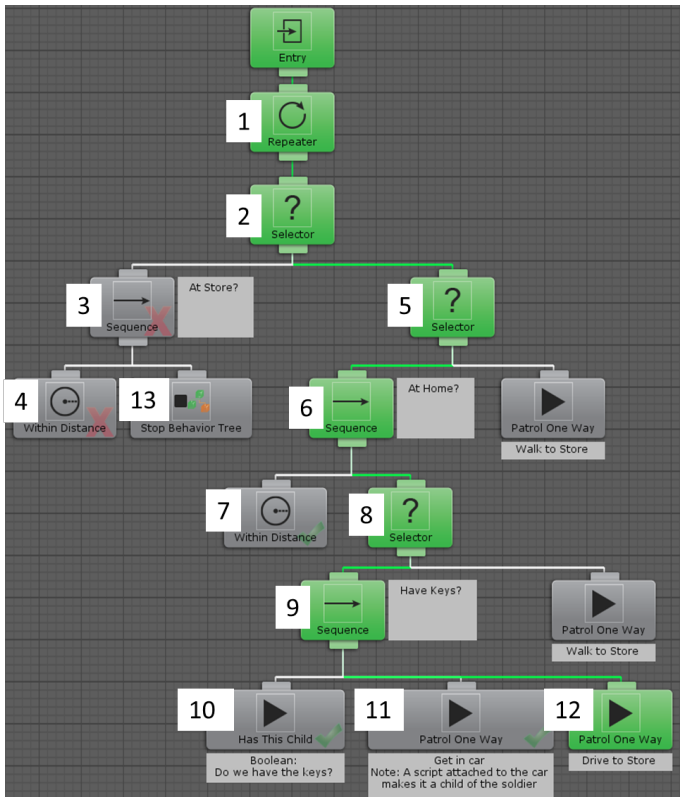

Figure 2.8: GoToStore Implemented as a BT in Opsive's Behavior Designer

1. Execution of the BT begins at the entry node, and proceeds to the repeater. The repeater ensures that this BT will not stop executing unless explicitly told to. Without the repeater node, the tree would stop after passing through all branches exactly once.
2. The next node is a "selector" node, which starts with its leftmost child. At the abstract level, this node dictates that the agent will only check for keys to drive to the store if it is at home. Otherwise, it will simply walk to the store.
3. This sequence node will check its children from left to right, which in this case is checking if the agent is at the store.

4. "Within distance" is a conditional node, and fails here because the agent is at home. It returns failure back to the parent sequence node labeled "3," which returns failure to the selector node labeled "2."
5. Execution passes to the next child of "2," which is another selector node.
6. This node represents the notion of checking to see if the agent is at home. It passes execution to its first child.
7. "Within distance" is a conditional node, which succeeds because the agent is at home. Because its parent is a sequence node, execution passes to its successor, also a child of node "6."
8. This selector node passes execution to its first child. This branch must be explored before the "Walk To Store" node is visited. This structure implicitly biases driving to the store over walking to the store.
9. Another sequence node, which passes execution to its children.
10. The conditional "Has This Child" is checking to see if the key is attached to the agent. It is, so this node returns success to its parent.
11. The agent then executes a movement task which directs it to get into the car. This task has been completed at the point in time represented here.
12. The BT is currently executing the "Drive to Store" task. Upon completion, it will return success back up the tree through nodes 9, 8, 6, 5, and 2, to the repeater node.
13. The repeater node then restarts evaluation of the tree, which finds that the agent is at the store. This time around, node 4 will return success, and node 13 will be visited, which stops the tree. Execution is complete.

In the same way that HTNs can be pieced together to create complex behaviors, this BT could also be built into more complex behaviors.

The following graphic, Figure 2.9, overlays the HTN nodes from Figure 2.2 on to the BT shown in Figure 2.8. In several cases, one node in the HTN equates to several nodes in the BT. For example, the sequence node labeled “2” and the conditional node labeled “3” combine to create the equivalent of the “At Store” constraint node from the equivalent HTN. Chapter 6 details the translation of BTs to HTNs, using both generic equivalencies and more complex examples.

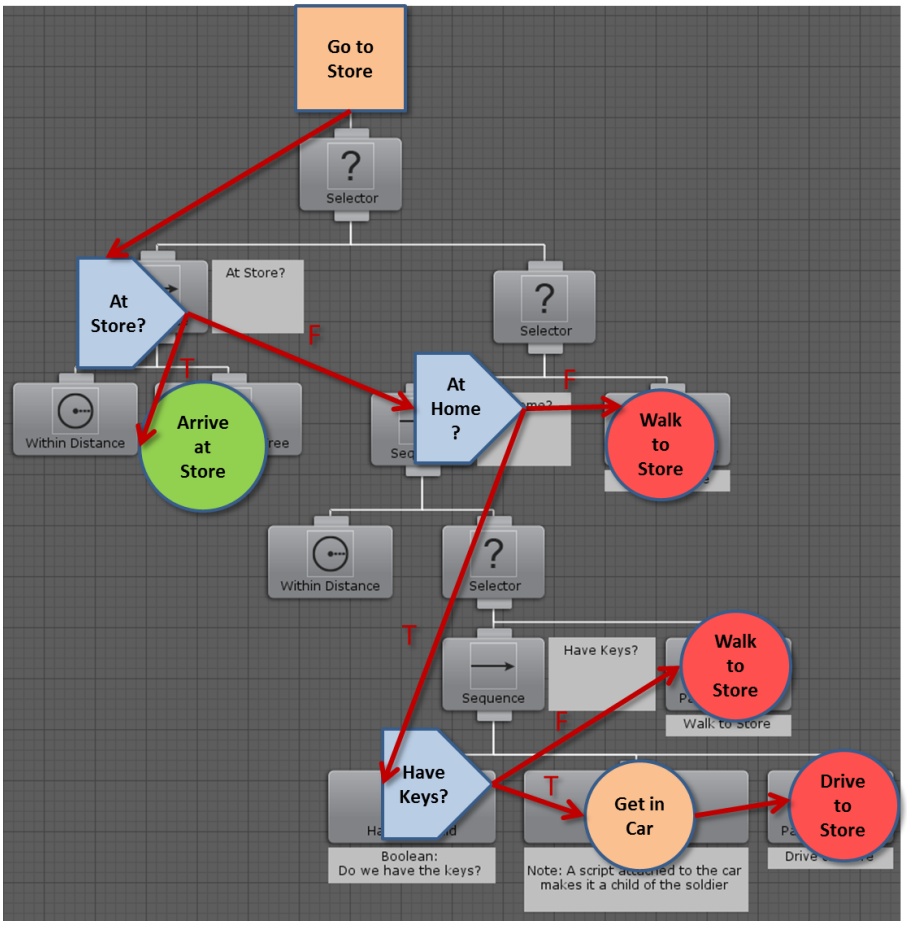

Figure 2.9: GoToStore HTN Overlayed onto BT

2.2.3 Using BTs to Emulate HTNs

The expressiveness of BTs means that their implementation can be as unique as needed. This also means that there are likely many ways to achieve a desired behavior. For example, one designer may design a tree whose first node is a parallel composite node. This means that each branch below the composite node will execute simultaneously. Another designer may build those two branches into two separate trees, and run them simultaneously. This effectively achieves the same behavior, but may appear significantly different. Yet it is the expressiveness of BTs that led to their selection for use in this thesis. After all, we are attempting to emulate a specific implementation of HTNs, and BTs provide the most versatile tool with which to do so. HTNs are a better fit for the precision needed in COMBATXXI than their be-

havior tree counterparts. However, while some major game development companies have used HTNs, there are not many implementations of HTNs available for use in the developers' tool market.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Managing Dynamic Behaviors in COMBATXXI

COMBATXXI is an agent-based simulation of combat, falling into the second highest of Dr. Ilachinski's eight tiers of complexity of combat models [3, p. 237]. COMBATXXI is one of the most detailed agent-based simulation of combat to date, and includes models of many of the complex processes which occur in combat, such as target acquisition, ballistics, variations in terrain mobility, tactical communications, and much more. The combat model itself has over 750,000 lines of code, and the primary development environment, Scenario Integration Tool Suite (SITS), is equally large. However, with this high level of fidelity comes complexity in building scenarios, an increase in the time required to run them, and more complex output data through which users must sort. Much of the complexity in COMBATXXI is not needed to test the basic behavioral logic of entities. The primary benefit of COMBATXXI–high fidelity–may inadvertently be the primary barrier to developing and managing agent behaviors.

3.1 Testing Behaviors in COMBATXXI

Testing agent behaviors in COMBATXXI is no trivial matter. Most analysis in COMBATXXI is conducted with the output data, which comes in two main forms. The largest form of output data is the "loggers". Although loggers can be difficult to sort through, they are the most reliable form of data output from COMBATXXI. Users can also utilize visual representations of troop movements and some events during scenario execution, but this is limited.

3.1.1 COMBATXXI Loggers

Loggers are typically .csv files that save certain specific types of data, and are only saved when specified in the scenario manager. There are currently 33 loggers that users can choose from, some of which have many more subordinate loggers with more specific information. The interface used for selecting loggers is shown in 3.1.

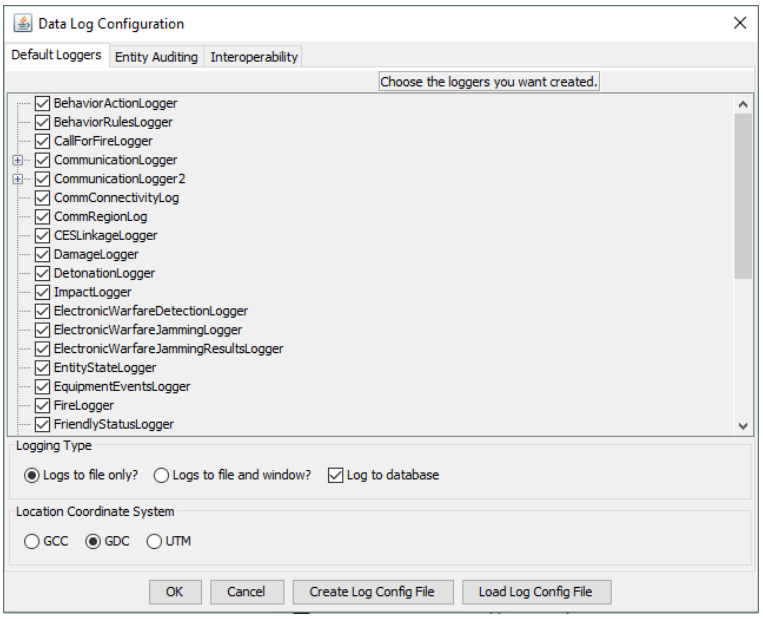

Figure 3.1: Loggers which save simulation run data are selected by the user.

Automated data processing tools are used by OAD and Training and Doctrine Command Analysis Center, White Sands Missile Range (TRAC-WSMR), the primary consumers of COMBATXXI. These tools may be built using common desktop software such as Microsoft Excel, or more flexible software such as Java. However, most of these tools are meant for post-processing of scenario output data, and do not focus on behavior debugging.

3.1.2 Visual Output

COMBATXXI output can be visualized with two different tools: a "viewer tool and a "3D viewer" tool. The viewer can be used while a simulation is running, or it can be used to view simulation output after scenario execution. The use of these tools within the work flow for a COMBATXXI scenario designer may proceed as detailed in the following sections.

3D viewer tool

The 3D viewer tool is primarily meant to visualize spatial orientation of terrain, obstacles, and buildings, and is meant to be used before running the simulation. A

scenario designer may use pre-existing terrain data, or they may build it on their own using other tools within the COMBATXXI architecture. The designer may use the 3D viewer at multiple points during the scenario development to ensure initial placement of entities is rational and as intended. There is no ability to use the 3D viewer during the simulation run or to view Distributed Interactive Simulation (DIS) data afterwards. An example of what a designer may see in the 3D viewer is shown in Figure 3.2. The 3D viewer was built to help terrain designers, who don't have the requirement to move around the scene as much as a behavior designer would. As a result of its initially intended purpose, it is not often used by behavior designers.

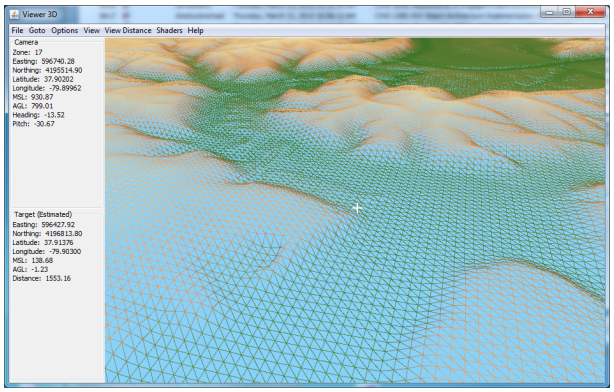

Figure 3.2: 3D Viewer tool in COMBATXXI. Source: [5].

Using the Viewer Tool

Before using the viewer, it is important to understand that it is portraying recorded DIS Protocol Data Unit (PDU)s as specified in the IEEE Standard 1278.1 [25]. This standard is meant to promote interoperability across simulation platforms, and is a natural fit for COMBATXXI. However, this standard can be limiting, for example, when soldiers have multiple weapons. The DIS standard only allows for the soldier's primary weapon system to be represented in a DIS PDUs, as shown in the soldier's profile in Figure 3.3. Additionally, the scaling of entities within the viewer is arbitrary and user-defined, so entity geometry is not accurately portrayed. There is no "rewind" functionality in the viewer–play and pause are the only options.

In spite of its apparent limitations, the viewer can be a useful tool for debugging a scenario. It does show some events, such as direct and indirect fire engagements, as

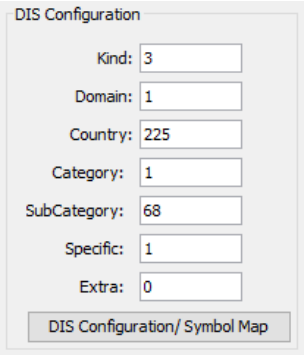

Figure 3.3: DIS enumerations are manually specified in the entity profile tab in SITS. The "SubCategory" field only holds one value for one weapon system.

well as fatalities. If COMBATXXI users choose to view the output from a single simulation run at run-time, it is typically done with the model output window open adjacent to the viewer. This allows users to see output and error statements, as shown in Figure 3.4. However, this output comes across the screen as fast as the simulation can be processed. This may be extremely fast in comparison to real-time if the simulation is relatively simple, or it may move slower than real-time if it is complex.

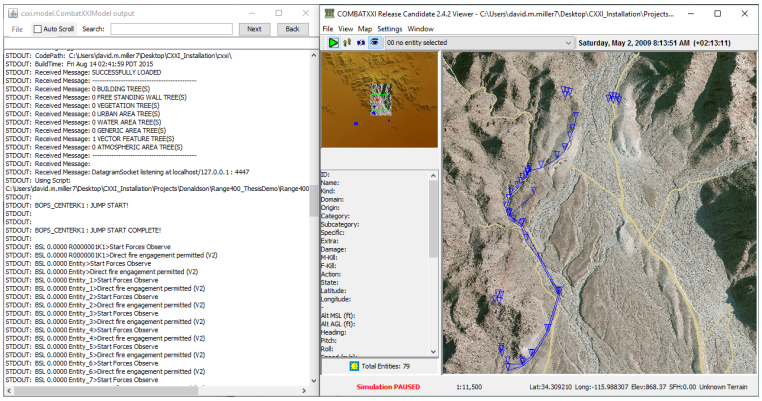

Figure 3.4: Viewing the model output panel adjacent to the viewer during a simulation run.

In practice, COMBATXXI users typically need to run many iterations of a simulation, and can do so from the "run manager" window without seeing each iteration's DIS output visualized. They then may analyze the output data using their preferred

method, and open the viewer retroactively only when something needs further investigation. Using the viewer retroactively produces the same result as viewing the DIS output at run time.

3.2 The Need for Robust Testing of Behaviors

Early testing is a crucial habit to prevent small bugs from causing large problems later in development [9, p. 238]. It is best to test often when developing code so that when bugs do occur, developers have a better idea of what may have caused them. This also prevents the developer from having to untangle the effects of having to change some earlier mistake on the rest of their code. This concept is equally applicable in COMBATXXI due to its complex structure. However, for all the reasons mentioned thus far, testing early and often in COMBATXXI is not easy.

3.2.1 Test Scenarios

Building specific scenarios for a variety of test cases is time-consuming manual work. For example, the basic scenario authoring tutorial is recommended to take 12 hours, and the terrain is already set up [26]. Building a set of scenarios with varying terrain is an advanced skill in COMBATXXI. For example, simply changing the orientation, size, or number of buildings in a given area requires new terrain to be built. This difficulty is prohibitive for developers who want to test their behaviors in a wide variety of terrain orientations.

The discussion in Chapter 1 regarding the nature of complexity theory focused on the inherent benefit of leveraging the complexity that emerges from simple behaviors, or the simple natures which emerge from complex behaviors [3, p. 5]. COMBATXXI aims to leverage the latter type of emergence, but the resulting macro-level behaviors are not always convergent. Sometimes, behaviors in COMBATXXI lead to divergent or at least unpredictable behaviors. This may be due to inconsistencies in the behavioral logic or the unpredictability of the external input to the behavior. Often, such inconsistencies arise later on the in the simulation, when the possibility for a wide range of world states prior to the instantiation of the behavior increases.

These are the most difficult behavior-related problems to solve in COMBATXXI,

because a simulation may take a long time to arrive at the point where the behavior is instantiated. The causal stimulus may not arise in each iteration of the simulation, further complicating the debugging process. Gathering data and recognizing a pattern that may be indicative of the underlying cause of the problem may take many more simulation iterations than the behavior designer has time to perform. One possible solution to this problem could be to build smaller scenarios which are similar to the scenario at hand, but this can be similarly time-consuming.

3.2.2 Modular Testing

It has been suggested that testing code in a manner similar to testing integrated circuits is a beneficial practice for software projects [9, p. 189]. That is, each atomic module of code should be tested on its own first, and then tested as a larger whole. This is a logical, and even intuitive way to test BTs and HTNs, particularly because a trademark of well-built behaviors is modularity.

Due to the potentially explosive complexity of agent behaviors in COMBATXXI, all possible emergent agent behaviors cannot be reasonably predicted through observation of the code alone. As with any software, early and ruthless testing are key to a successful outcome [9, p. 237]. Therefore, a thorough verification and validation (V&V) routine should include a wide array of test cases that diversely challenge and completely encapsulate the behavior requirements. Ruthless testing of any software helps to confirm two things: whether the code satisfies the requirements, and whether the requirements match the customer's intent. Working with subject matter experts to develop realistic test-cases may help to identify problems with individual behaviors before they are brought in to already complex scenarios.

3.3 Options for Improved Testing in COMBATXXI

There are three options going forward for behavior development in COMBATXXI. The status quo can always be maintained, and should be considered. If improvement efforts incur a significant manpower investment, the status quo may be the only tenable option. The other two options include development work, either by way of extending COMBATXXI's current development environment, or by building a

surrogate test environment for behavior developers.

3.3.1 Extending COMBATXXI

Extending COMBATXXI should not be ruled out. The primary benefit of investing development efforts to extend the COMBATXXI development environment is that those improvements would reside in the same code base as COMBATXXI. This would result in more direct translation of development efforts, as opposed to a surrogate environment from which behaviors must be translated.

There are three ways in which COMBATXXI could be extended. First, the viewer could be made to incorporate a modern 3D viewer–combining the top-down viewer and the current 3D viewer into one. The ideal 3D viewer would be comparable to modern game development environments' viewers. This could include more direct control of the playback speed, as well as a rewind option. A second option could be to build higher-level abstractions within COMBATXXI for current models which currently require burdensome detail. Lastly, reusable prototypes could be built which allow designers to quickly deploy highly complex networks of agents. An example for the latter two could be the use of indirect fires (IDF), which currently requires the existence of a unit to fire the IDF, a fire control center, and a forward observer. If a scenario designer only needs the effects of IDF, it would be useful to have an easily deployable IDF zone which can be easily initiated from a behavior.

3.3.2 Testing in a Surrogate Environment

The final option, and the focus of this thesis, is the development and testing of behaviors for COMBATXXI in a surrogate environment. The following chapter examines best practices from the commercial video game industry and how those methods may benefit the development of dynamic behaviors.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Best Practices from Commercial Game Development

In 2015, the commercial video game industry achieved a record $61 billion in worldwide sales, split between mobile, PC, and console games [27]. Although traditional console games are losing market share as a percentage, individual console games have brought in incredible amounts of revenue in recent years. For example, sales of *Grand Theft Auto V*, which was released in September 2013, totaled more than $1 billion in the three days following its launch, and high-grossing releases are no longer the exception. These major titles are incredibly complex, and require large amounts of investment up front to develop. *Grand Theft Auto V*, for example, reportedly cost over $265 million to produce [28]. More recently, it is estimated that *Final Fantasy XV* will have to sell 10 million copies just to break even after its release in late 2016. Considering that only one *Final Fantasy* release has sold that many copies thus far, this increased acceptance of high development costs is representative of the fact that the market for these types of games is still growing [29]. With this increased spending in commercial game development, it is reasonable to believe that there may be best practices for development which could be applied to particular challenges within the DoD M&S community.

Military simulations face many of the same development challenges as commercial video games, such as optimization, debugging, and code re-use. High-resolution military analytic simulations include detailed physics and system models of combat equipment such as tanks and artillery, however, cognitive or behavioral representation is more of a challenge. The concept of maneuver warfare has represented a challenge for the DoD M&S community since the advent of agent-based simulations and the corresponding divergence from attrition models [3, p. 25]. While military doctrine can serve as a starting point for maneuver-related behaviors, different subject matter expert (SME)s may have varying interpretations of that doctrine. This complicates the development and validation of maneuver-based behaviors, and military agent-based behaviors in general. Further complicating the task of behavior developers, the majority of military operations of the past decade have involved irregular, or asym-

metric warfare. Modeling these types of conflict is even more complex than traditional force-on-force combat [30].

However, at the core of these complex military simulations is the agent, be it a soldier, a tank, an airplane, or an insurgent. Nearly all commercial games require some form of artificially intelligent agents to act as enemies, companions, or other non-player characters, and all of those agents require behaviors. It is reasonable to posit that some best practices from the commercial game industry may be useful for the development of agent-based behaviors for military simulations. Adopting the most applicable methods may improve "modelers' ability to explore possibility spaces and potential outcomes" [31]. This chapter provides an overview of the techniques used in the commercial game development industry and the software engineering industry at large as they may apply to the development of dynamic behaviors for military simulations.

4.1 Applicability of Game Development Techniques

The increasingly large budgets of commercial video games makes it ever more likely that cutting-edge development techniques will emerge from the commercial domain. However, this does not mean that all techniques developed for real-time video games will transfer well to high-resolution, multi-agent based simulations of combat. In order to explore the applicability of commercial game development best practices to the development of behaviors for multi-agent based simulations of combat, we must first identify the traits which these two development domains have in common.

4.1.1 Analytic Simulations' Similarities with Video Games

The similarities between game development and simulation development far exceeds their differences. The fact that they are both rooted in highly complex code bases means that much of what applies to general software engineering applies to both game and simulation development. Both domains at least partially rely on OOP, behavior development and re-use, and managing complex models.

Object Oriented Programming. OOP is at the core of both game and simulation development. For example, the highly regarded *Game Programming Patterns*,

by Robert Nystrom, is essentially a modern, game development-focused update to the classic *Design Patterns: Elements of Reusable Object-Oriented Software* [8, p. 4]. *Design Patterns*, which was first published in 1994, is viewed as a classic for OOP development. If the fundamentals of OOP are still useful for the development of commercial games, it is likely also useful for the development of military simulations.

Behavior Development and Re-use. Behavior development can be a tedious process, and as with any software project, re-use can help to reduce the need for work duplication. This concept may fit in with the OOP paradigm, where two classes of agents which inherit from the same parent class behave in very similar ways. This concept applies to military simulations as well. For example, a tank and an armored personnel carrier are both types of vehicles, and should have some similarities in the way they behave. Modelers should not need to design completely new behaviors for actions, such as movement, which are basically similar between those two classes. Another way in which code re-use benefits military analytic simulations is the applicability of a behavior to multiple echelons of units. For example, an ideal maneuver behavior which orders a squad to move in a wedge could also be used to order a platoon to move in a wedge.

Managing Complexity. With approximately 1.5 million lines of code in the COMBATXXI development environment, and hundreds of millions of dollars going in to commercial game development, it is not a stretch to say that both domains have to be adept at managing complexity. Challenges that arise in highly complex software projects include version control, geographically dispersed development teams, interoperability, and architecting for forward compatibility. While both OOP and behavior re-use help manage this complexity, there are myriad other solutions that can help to streamline the development process.

Both the commercial gaming industry and the military simulation community demand reliable implementations of models. Bugs in games can prevent them from being distributed on a given platform, and errors in simulations can cause crashes that result in the loss of hours of processing and debugging time. Both industries leverage the OOP paradigm for code reuse and extension. However, that is mostly where

the similarities stop, at least within the specific domain of military simulations for analysis, where 3D visualization is not a priority.

4.1.2 Analytic Simulations' Differences with Video Games

The differences between commercial games and analytic simulations are important to note so that reasonable assessments can be made regarding which methods may not transfer well from the commercial game industry.

Number of agents. Typical, large-scale military simulations of combat include military units up to the Brigade level. A Marine Corps MAGTF, such as a ship-based Marine Expeditionary Unit (MEU), consists of approximately 2,300 Marines. The number of agents in a military analytic simulation will be greater than a typical commercial game. Therefore, it is beneficial to be aware of the scaleability of a given technique. The OOP nature of analytic simulations accounts for this, and the fact that analytic simulations can be run slower than real-time means that scaling will only increase the run time.

Time-step vs. Discrete Event. Modern video games revolve around the game loop pattern, which updates the state of the game as much as possible between each frame rendered [8, p. 128]. Typically, this means that the state of the world is updated 30 or 60 times per second under ideal conditions. The simplest variety of this pattern is depicted in Figure 4.1. Contrastingly, constructive military simulations such as COMBATXXI leverage discrete event simulation in order to maximize event timing precision, minimize computational resources, and decouple the simulation state update process from other processes, such as graphics rendering process.

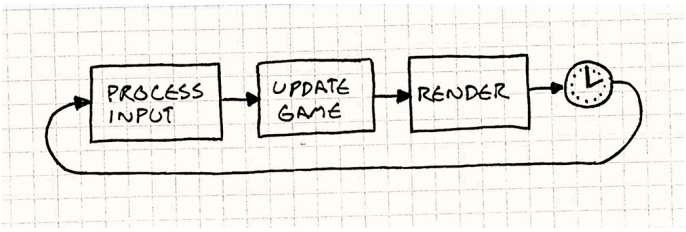

Figure 4.1: Simple Game Loop Pattern. Source: [8]

Graphics focus. Commercial video games are able to use more realistic graphics with each new iteration of hardware. Graphical representation of the game world is often the primary focus of commercial games. This means that a good deal of the computational resources must go to the "render" portion of the Game Loop depicted in Figure 4.1. COMBATXXI does not necessarily produce any graphical output, unless specified by the user. This is due to the intended use of COMBATXXI scenarios, which are typically run dozens of times in order to produce analytically tractable output. In an already complex simulation, coupling the simulation state update process to a rendering process would needlessly, and likely excessively slow down the output production process. However, the user may request visual representation before runtime, and DIS PDUs are logged and played back on a top-down map in order to spatially depict the events in the simulation–but only after they have occurred. Section 7.2.2 provides suggestions for future work which may leverage commercial game engines' graphics focus for the betterment of COMBATXXI model validation and analysis.

User input. Video games rely on user input in order to drive the gameplay. Without user input, there would be no game. Most constructive simulations are closed-loop, meaning that no user input is provided during run-time, or only have limited high level command input. This puts more of an impetus on the fidelity and realism of the agent behaviors in multi-agent based constructive simulations, but does not create any additional barriers for the use of commercial game development best practices.

Deployment Environment. Video game development teams are often tasked with deploying a game on to multiple different platforms. This requires foresight when architecting the game development environment that may not be necessary for the DoD's analytic simulations. In the case of COMBATXXI, the developers and the end users are often in direct communication, and they are able to ensure that new hardware purchases or potential changes to the architecture will not adversely affect the ability to use the model.

These differences do not mean that game development practices related to them will not be useful for the development of behaviors in analytic simulations. Instead, they serve as context to help focus the search for the development mechanisms which are

most likely to be applicable.

4.2 Identifying Applicable Best Practices

As with any software design, the commercial game development industry is subject to the sirens' call of buzz words, alleged short cuts to game design success, and fleeting design fads. Additionally, there are likely to be some highly successful techniques which simply do not help development of agent-based behaviors due to the reasons discussed in the previous section. Therefore, it is the imperative of simulation designers to attempt to qualify attributes of development techniques that will most readily apply to the development of agent-base behaviors. Here we examine those qualities of a behavior development methodology which may contribute to the betterment of the behavior development process, thereby increasing quality and decreasing development time.

4.2.1 Higher Quality Behaviors

Defining exactly what makes a behavior have higher quality is neither an exact nor an agreed upon science. However, a set of attributes has been suggested in previous thesis work at NPS to identify high quality behaviors [10, p. 8]. Here we examine those attributes with the intent of examining whether or not a given development methodology will lend itself to producing behaviors with these qualities.

Creation: Is the methodology sufficiently expressive to allow for the creation of complex military behaviors?

Efficiency: Does the methodology help to streamline the scenario development process, reducing the overall production time, and improve the ability to communicate the intent behind the behavior to other potential users?

Modularity: Does the methodology lead to behaviors which can be composed into progressively more complex behaviors?

Reusability: Does the methodology create behaviors which can be reused in other scenarios?

Scalability: Can behaviors built with this method be used at different echelons of military units within the simulation?

Although poorly written behaviors can be created in the best of development processes, a limiting development process can prevent the above criteria from being achieved even by experienced authors. It is with this in mind that we seek out the most enabling development methods and tools for our dynamic behavior development.

4.2.2 Faster Development Cycle

Aside from the overall quality of a behavior, the other most significant question to be asked of a development process is whether or not it helps to reduce the development time. Developing, distributing, and providing requisite instruction for a behavior prototyping environment requires a potentially large amount of effort up front, so it needs to provide significant development time reductions with increased use. If simple behaviors are needed for simple scenarios, the efforts required to build a prototyping environment may outpace their benefit.

While development time is more practical to measure than behavior quality, conducting a full user study is beyond the scope of this thesis. Although assessment of a faster development process is somewhat arbitrary, the following sections in this chapter aim to qualify characteristics of a behavior development environment or methodology which may help reduce the time required to develop new dynamic behaviors.

4.3 Benefits of 3D Development Environments

Cognitive modeling of agent behaviors is a difficult task to undertake. Similar to planning in real-life, unintended stimuli have a tendency to throw our plans, or in this case our agents' behaviors, off track. Due to the highly complex nature of COMBATXXI, it is not likely that our behaviors will work the first time. We will need to debug them, and to do so we must first grasp exactly what is breaking them. COMBATXXI users can leverage the loggers, the runtime output, or the visual output to do this, but game designers are able to leverage the visual nature of their games to design and debug.

Wallach, O'Connell, and Neisser's article about the so-called "kinetic depth effect" in the *Journal of Experimental Psychology* in 1953 posited that three-dimensional views of complex scenes enabled users to better understand what was happening

within those scenes [11]. Recent research into understand how people retain three-dimensional information has corroborated this hypothesis, arguing that this ability "enables the user to obtain a different view suitable to the task, reveals structures that may otherwise be occluded, and allows better perception of rigid 3D structure, through structure-from-motion, also called the kinetic depth effect" [12].

A similar logic applies to the development of artificially intelligent behaviors which play out in a three dimensional environment. As an example of this concept in the commercial game industry, the AI development team for *Sunset Overdrive* at *Insomniac Games* was able to visually debug their game's enemy behaviors within the levels in which they would be employed. This included visual representation of path planning and other parameters needed to fine tune the behaviors [32] of the game's antagonists. Had they needed to rely only on textual output, this certainly would have been a more difficult process.

4.4 Prototyping Models and Behaviors

In order for a prototyping environment to be useful, it must eventually save the developer time. In order to do so, it should be as fluid and dynamic as possible, enabling the developer to implement many designs in a short amount of time in order to find the most suitable design for the end use simulation. An ideal prototyping environment enables designers to safely explore combinations of models and behaviors without breaking prior existing models and behaviors.

Game development tools enable developers to easily reuse models that have already been built and quickly develop new models based on previously existing models. In order to accomplish this, three common concepts may be used: the object-oriented programming paradigm, the blueprint pattern, and the sand box pattern.

4.4.1 Object Oriented Programming for Prototyping

Object Oriented Programming (OOP) is a cornerstone of many software projects, including COMBATXXI, and it is worth exploring the benefits of OOP in relation to prototyping and transferring behaviors. OOP is more likely to be useful in a prototyping environment if the end use environment, in our case COMBATXXI, leverages

OOP. Mimicking the class structure of the end use environment will assist with transferability when a behavior is finalized.

Deviation from mimicking of the end use class structure should only be done when absolutely necessary. At best, it will make the transfer process more difficult and error prone. At worst, it may lead to behavioral constructs in the prototyping environment which don't transfer to the intended simulation environment.

For example, lets imagine that a machine gunner entity in inherits from the basic soldier entity in our end simulation. The majority of the functionality of the machine gunner is based on that of the basic soldier entity, and they share many of the same variables. If our prototyping environment develops the basic soldier and the machine gunner such that they have different public and private methods or variables, this could create unnecessary confusion or even lead to an infeasible transfer.

4.4.2 Reusable and Dynamically Adjustable Components

Game engines typically allow designers to build or import a model once and then use it many times within the game. These objects are called "blueprints" in the Unreal Development Kit, and "prefabs" in Unity3D [33].[6] For clarity, the concept will be referred to as prefabs from this point forward.

Prefabs are like blueprints to a model. The blueprint exist as a concept, but until it is brought into the simulation, it has no functional purpose. Prefabs are reusable, different instances of the prefabs may have different parameters at different points in time. For example, if a soldier is implemented as a prefab, the prefab may start out with 100 level health. Obviously each instance of the same prefab may have different levels of health. In this way, each instance of a prefab comes from the same source but exists as its own object.

Prefabs may often be similar to one another, and therefore it can be easy to build a new prefab based on a prior existing one. This is similar to classes in OOP with one key difference. A prefab which is based off of a prior prefab will break all ties with that previous prefab.

[6]More on Unity prefabs can be found here: http://docs.unity3d.com/Manual/Prefabs.html

This is a somewhat similar to profiles in COMBATXXI, with one key difference. In SITS, the representation of an entity must be placed on the force structure board prior to a profile being applied to it, while a prefab typically encapsulates the entity and the profile. They are similar in that they are both easy to make new deviations from prior existing profiles, and the new profiles can be independent of the original profile from which they were built.

This concept helps behavior developers to quickly alter, test, and revise behaviors that are attached to agents in the prototyping environment in a manner that does not affect previously built agents models or behaviors. If the new design is preferable to the old design, it can be made to replace it outright, or replace it in future designs.

4.4.3 The Sandbox Pattern

The "Sandbox" pattern is another way of providing a wide range of stable yet dynamic options for game designers. The example provided by Robert Nystrom in *Game Programming Patterns* of interchangeable "spells" which a hypothetical game uses to allow wizards to compete is similar to our desire to rapidly experiment with entity-level behaviors in COMBATXXI [8]. Game designers and players alike can experiment with different combinations of spells without breaking the game or inducing unforeseen bugs to the code.

This is similar to the concept of modularity in behavior development for simulations. Any given behavior, whether represented as a behavior tree, a hierarchical task network, or any other abstraction, will likely be composed of many sub-tasks. Ideally, those sub-tasks have the ability to stand alone or be combined in different combinations in a variety of situations without inducing bugs. This concept allows behavior designers to rapidly experiment with a variety of behavioral logic that meets their requirements.

The sand box pattern can be used throughout a game's design, including the area of behavior design. For example, game designers may want players or other designers to be able to build new characters. Perhaps the designers want to be able to easily swap out masks, clothing, or other visual components of their characters. The sand box method would build all these potential elements and allow the user to safely swap

them out without breaking anything.

The sand box pattern may be used in commercial games in abstract low-level code in to higher-level tasks. These higher-level tasks can then be combined in various ways by behavior designers who won't have to interface with source code [8]. If scoped properly, these designers, working at the abstract level, can explore many different types of behaviors without touching any code or breaking any other behaviors.

4.4.4 The Good in Bad Code

Although not a primary benefit of prototyping in a parallel development environment, the required transfer of code from one environment to the other has collateral benefit. The notion that a behavior developer should expect to throw away their first design is widely held, as summarized in Nystrom's *Game Programming Patterns*, and this section's namesake [8, p. 14]. Discarding any development work may seem wasteful, and can be resisted by developers and managers alike. However, faster iteration cycles beget a faster initial development, and this makes the scrapping of the initial design less painful.

The design schema proposed in this thesis involves writing behaviors in a programming language and software application which are different than the intended application. This method may seem inherently flawed by incurring the risk of transposing behavioral logic, yet it has potential benefits. Specifically, it ensures that prototypes must remain prototypes, and that no hastily built behaviors can be used as supposed production quality behaviors in COMBATXXI scenarios. Therefore, supposedly "bad code" is less likely to find its way in to production-level analytic simulations which will be used to make valuable decisions.

4.5 Test Driven Development

Test Driven Development (TDD) is a popular development technique in which no code is written without first building tests against which that code will be validated. The general process can be summarized by the following six steps [34]:

1. Add a test to check one or several components of the specification.

2. Run each test in the test set.
3. Write additional source code in order to pass the tests from step 2.
4. Run each test in the test set again.
5. Refactor the current source code to pass the tests.
6. If not all specifications are met, return to step 1.

As with any mature software development process, TDD for a behavior should not begin without a clear specification of the expected fidelity and level of detail to be included in that behavior. Fortunately, military doctrinal publications provide a standardized starting point for these behaviors. The use of military doctrinal publications, when available for a specific behavior, allows for the generation and approval of detailed requirements which are well-suited for testing. Additional techniques for building tests may include soliciting vignettes from SMEs who may be able to encapsulate information that bolsters the difficulty of the tests and the resulting robustness of the behavior.

4.5.1 TDD as Applied to Prototyping

While the end-validation of a scenario should be rigorously adhered to in the end use simulation environment, the TDD construct can be relaxed in the prototyping environment. The prototype behavior is built based on an abstraction of the model in which it will be eventually used. The entire benefit of the parallel prototyping schema is that it should speed up the process. Perhaps tests are written for the prototype, as in step 1 of the TDD process, but the designer building the prototype isn't able to build the components for a behavior which meet those tests in a timely manner. It may be necessary to further abstract the prototyped behavior to move the process along.

For example, imagine a designer is building a mounted patrol behavior. First, the soldiers need to mount in the vehicle. The designer may initially want to explicitly model the movement of the soldiers to the vehicle in the prototyping environment. However, the designers finds some development environment-imposed limitations which make this difficult, but he can "warp" them to be magically inside the vehicle. If the designer already knows how this "warp" will map to soldiers loading into the vehicle

in the end use simulation, time spent troubleshooting this is time wasted. Because the prototype is going to be discarded at the end of the process, a strict approach to TDD within the prototyping environment is not necessary.

4.5.2 TDD on the Decline

Although TDD is still a widely utilized programming paradigm, it does have valid and vocal detractors and should not be held as irrefutable gospel [35]. TDD is better utilized as a set of guidelines as opposed to hard-and-fast rules. For example, the behavior developer may not need to proceed all the way through the refactoring of current code to realize that they need to add additional code before the current code should be expected to work. As in all programming paradigms, common sense prevails, but TDD does provide a well-structured, forward-thinking paradigm that has the potential to enable developers to build more robust behaviors in a shorter amount of time.

Regardless of potentially waning acceptance and use of TDD, it holds particular potential for the development of agent-based behaviors. *Codingame.com* is a popular website which allows novice and expert programmers to hone their skills by building agent behaviors which accomplish certain tasks in a video game application. Each application has an agent which can be programmed using one of several major programming languages. The user can see the tests in advance, and must write code to meet the tests. Each test adds an element of complexity, and each iteration of code development must account for this new complexity. In this way, users of *Codingame.com* are executing TDD without actually writing the tests themselves. Combining this construct with military doctrinal specifications as a source for tests makes TDD a promising candidate for use in the development of military agent-based behaviors such as the HTNs used in COMBATXXI. A final note on *Codingame.com* is that the programmer is able to immediately test their code and see the results of any changes that have been made. This fast iteration cycle is likely meant to keep users engaged, but is also a best practice for behavior development, as discussed in section 4.2.2.

4.6 Selecting the Most Applicable Methods

The commercial video game industry, along with the programming community at large, provide myriad approaches for building a prototyping environment. Designers should leverage those methods which minimize their development time and maximize their ability to explore the design space and then transfer the behaviors to the end use environment. While some game development methods are meant to solve problems in less relevant areas such as graphics rendering and multi-platform deployment, the methods discussed here are applicable to the prototyping of behaviors for military agent-based simulations of combat.

CHAPTER 5:
Unity as a Behavior Prototyping Environment

Unity3D, known more simply as Unity, began development in 2001 as a small project by Unity Technologies–a company of only three developers. By 2005 it was released at Apple's Worldwide Developer's Conference, and quickly gained popularity as it leveraged an increase in game developer demand for a game engine which simplified deployment to multiple platforms [36]. By 2014, Unity had achieved a 45% share of the global game engine market, with over 4.5 million developers actively using the platform [37]. There is a free version of Unity that provides nearly full functionality, or a commercial version which costs $75 per month, and allows users to leverage advanced application deployment resources.[7] There is an active online community of developers and support personnel, a robust marketplace for reusable code and models, and an existing base of experts within the Modeling, Virtual Environments, and Simulation (MOVES) Institute at NPS. Unity was selected as the most appropriate game engine for use in this thesis for exactly these reasons.[8]

5.1 Introduction to Unity

Unity is simple enough that users with only basic scripting experience can learn to build deployment-ready games in a reasonable amount of time. This is accomplished through an intuitive user interface, a robust online community and plug-in market, and an abundance of free tutorials built by the Unity team.[9] Certain concepts need to be understood to fully grasp the potential of Unity as a behavior prototyping environment. Many of these concepts will be familiar to game developers who have developed with other game engines, but some concepts are unique to Unity. These descriptions only cover the level of detail required to grasp the utility of Unity for a behavior prototyping environment.

[7] Additional details on Unity3D licenses can be found here: http://unity3d.com/get-unity
[8] Unity 5.3.1 was used for this thesis.
[9] Tutorials for all skill levels can be found at: http://unity3d.com/learn

5.1.1 Basic Unity Concepts

The following sections cover only the key concepts within Unity which will help to understand its suitability for use as a behavior prototyping environment. For more in-depth explanations of the various aspect of Unity, please visit Unity3d.com.

GameObjects

GameObjects are the primary container within Unity that hold most other elements. Every GameObject must have a location, scale, and orientation, which are contained within the GameObject's transform, but otherwise a GameObject may be empty. The position, rotation, and scale of a GameObject are contained within that GameObject's transform, and transform hierarchies define the parent-child relationships within a scene. The parent-child relationship of GameObjects is what ties them together in the virtual environment. A child GameObject will change synchronously with its parent. For example, when a parent GameObject moves forward in the scene, the child will move along with it unless specifically programmed not to do so.[10]

A GameObject may be a plane which represents the ground or a mesh cube with a solid color material which represents a building. GameObjects may also be highly complex, containing not only the aforementioned elements which make the object visible in the scene, but also scripts that contain detailed behavior-related information. The GameObjects used in the course of this work were kept as simple as possible, as the focus was placed on behavior development. Cinematic effects were only added where necessary for debugging.

Game Loop Pattern

At its core, Unity is a game development engine, and therefore revolves around the frame update cycle or game loop pattern, as discussed in section 4.1.2. It is sufficient for the purposes of this thesis to understand that Unity, or any other game engine's reliance on the game loop means that simulations implemented in these game engines will update the world state once each frame and are therefore inherently time-stepped.

[10]For a detailed specification of the Unity transform, see the following: https://docs.unity3d.com/ScriptReference/Transform.html

This is not necessarily a limitation for our purposes, as no analysis is being conducted in Unity at this time.

MonoBehaviour

The primary enabler of GameObjects in Unity is the MonoBehaviour. Not every GameObject must have a script attached to it, but nearly every script that is attached to a GameObject is derived from the base class MonoBehaviour.[11] Scripting in Unity can be done in Javascript, Boo, or C#. Most people seem to use C# as their preferred language.

Prefabs

Prefabs are an asset type in Unity that acts as a template GameObject that designers can use for commonly reused GameObjects. Typically, a prefab is built by building it as an independent GameObject in the scene, and then saving it as a prefab. This preserves all components and characteristics of the original GameObject in a reusable form. A prefab is similar to a class, in that it defines what its spawned GameObject will be when it is instantiated. Once a prefab instance exists in the game, it exists as its own object, and may change its state without any ties to other instances of the same prefab. However, if the prefab is updated, it will affect each instance of the prefab in the scene, even if they have been changed. Prefab instances can break their connection to the parent prefab, resulting in complete autonomy. Any type of GameObject may be made into a prefab within a game. Players' avatars, non-player characters, and inanimate objects may all be instantiated from prefabs.

Prefabs also work as expected when the designer expects dimensional or parametric differences in the instances. For example, a designer may create a prefab of a generic, symmetrical box which represents a building, and then create a dozen instances of that prefab within the scene. He may then change the scale and rotation attributes of the building without "breaking" its connection to the prefab. Then, if the designer later decides to change the color of the building prefab, doing so will change the color of all instances of the prefab without erasing the custom dimensions that were already

[11] Details on all methods within the MonoBehaviour class can be found here: http://docs.unity3d.com/ScriptReference/MonoBehaviour.html

applied. Only the element of the prefab which was changed resonates through to the prefabs' instances.

Assets

The term "assets" in the context of Unity refers to basically any developed Unity accepted file type which can be transferred between users. This may include GameObjects, scripts, developer's tools, entire scenes, or developer-specific files such as "dll" files which are required to enable a plug-in's functionality. Unity provides a user-friendly container, known as a Unity package, for assets that are sold in the Unity Asset Store or exchanged between users. Assets are best kept in a central location and imported into a project only when needed.

Project

The Unity project is the container, represented as a folder, which holds all GameObjects, scripts, meshes, materials, prefabs, scenes, assets and other elements that are typically needed to produce a game. The project also defines the scope of the prefabs and scripts used in the project. This is important to note because any changes made within any part of a project to a specific prefab, script, or other asset will affect all instances of that item within the project. This is not necessarily limiting, as a project can be duplicated, altered to experiment with some new or altered behavior, and then discarded if a developer doesn't want to risk breaking other dependencies of a specific item. Version control systems also provide protection from unintended consequences, and are widely used in the Unity developer community.[12]

Scenes

Each game level in Unity is contained within a scene, which is a virtual environment in which the game is played. Scenes have their own world space and origin. Scenes may be linked together with a script that specifies their end conditions and order as in most games, or they may stand alone. An individual scene may contain everything

[12]For information on using Git, a source control standard for Unity development, visit this website: https://unity3d.com/learn/tutorials/topics/cloud-build/creating-your-first-source-control-repository

needed to prototype a set of behaviors, including terrain, obstacles, friendly and enemy units, and elements that enable the virtual environment, such as lighting.

Scenes enable the use of the Sand Box Pattern as discussed in section 4.4.3, as they may be completely self-contained. This allows behavior developers to conduct rapid testing and try to break their behaviors before they are used in a production-level environment.

5.1.2 The Unity Editor

The basic Unity concepts are all brought together in the Unity developer's user interface (UI), known as the "editor", which is a customizable, panel-based user interface. Unity is shipped with a set of standard windows in a default layout, but developers can build custom windows to make the editor more suitable for their specific work flow. The most common out-of-the-box windows in the editor are the scene, game, console, hierarchy, project, and inspector windows, as shown in Figure 5.1. These windows enable the basic functionality for development in Unity.

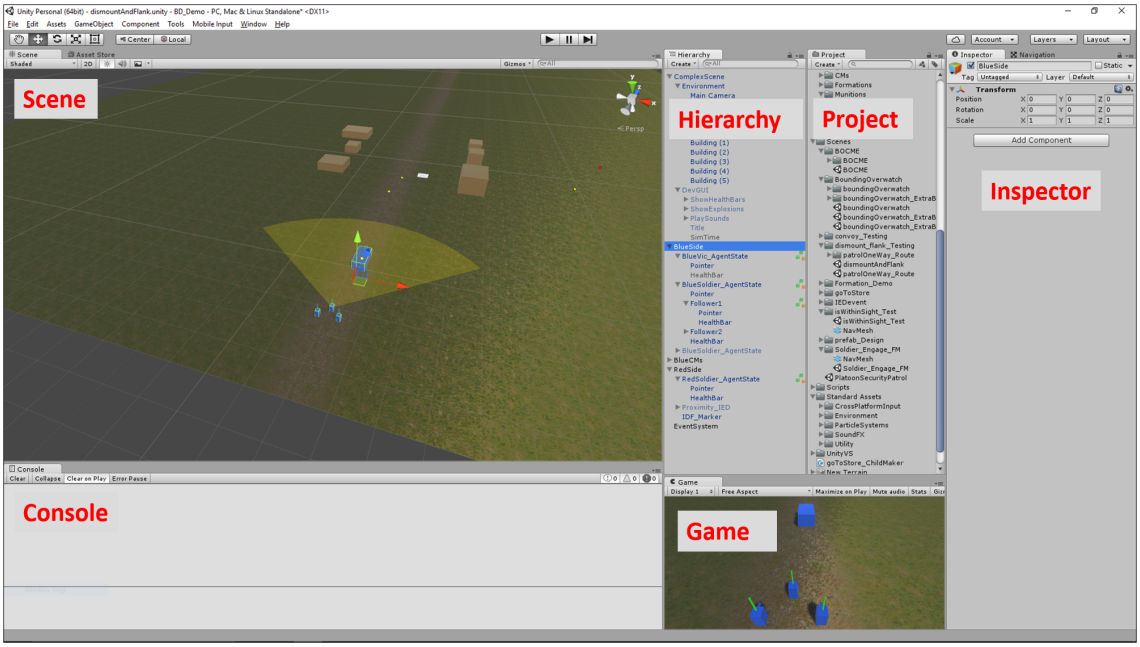

Figure 5.1: A generic Unity editor configuration

Starting with the upper left of the window in Figure 5.1, the scene window is one

of two visual representations of the scene. The scene window is not attached to any GameObject, and allows developers to view the scene from any position and orientation within the virtual environment. Through mouse and keyboard controls, users are able to virtually fly around the scene to examine the spatial relationships of various objects, or to zoom in on a specific part of the game. The scene window's counterpart, the game window, is depicted in the bottom right. The game window is tied to a specific camera's location and frame rendering parameters, and is typically used to provide game consumers with their character's perspective, or to provide a static overview of the scene.

The right half of the screen is occupied by the hierarchy, project, and inspector windows. The hierarchy window holds all GameObjects contained within a scene. The objects are organized by their transform relationships that create a hierarchy of GameObjects. Transform information is composed of a GameObject's position, rotation, and scale. Changes to a GameObject's transform are passed down to its children by default, but changes to a GameObject's child transforms do not necessarily have any effect on the parent's transform.

The project window shows all files contained within a project. This may include scenes, prefabs, 3D models, images, scripts, and other elements typically used to build a commercial game. It is similar to what is seen when opening the project folder from Windows Explorer.

The inspector window depicts detailed information about whatever specific GameObject is selected by the developer. This window is useful for run-time debugging, as developers can see the current values of public variables in scripts attached to GameObjects. Developers can also change those variables while running the game or simulation, allowing them to see how various behaviors change with parametric changes. The inspector window also serves to organize all elements of GameObject, such as scripts, animations, sounds, and navigation components.

The console window is similar to consoles in other development environments. Unity allows scripts to print to the console using general messages or error messages, which are highlighted in red.

The Unity editor also allows users to play, pause, and progress the game frame-by-frame by using the three buttons centered at the top of the editor, as seen in Figure 5.2. This is an excellent tool for allowing developers to see how their variables are changing frame-by-frame. This functionality also helps to debug many potential time-step simulation errors, where the precise time of detection or other events may be important [38].

Figure 5.2: The Unity pause, play, and frame-by-frame buttons

5.2 Scripting with Unity

Although Javascript and Boo can be used to write scripts in Unity, C# was used for this thesis due to the author's familiarity and an abundance of resident expertise within the MOVES Institute. This section discusses scripting specifically as it relates to the use of C# in Unity.

5.2.1 Using MonoBehaviours

Scripts written in C# typically derive from the MonoBehaviour class, and execute a series of methods inherited from that class with each update cycle. A full visual representation of the MonoBehaviour and the update cycle it represents can be seen in Appendix B. For the purposes of this thesis, the term "scripts" may be assumed to mean a script which inherits from the MonoBehaviour class.

Although there are many methods which may be called during the MonoBehaviour lifecycle, the two most important are Start() and Update(). The Start() method is called only once at the beginning of the script's life, and is typically used to gather and set variables for use in the rest of the script. The Update() method is called once each frame, and is used to drive the script using the game loop pattern.[13]

[13]Unity3D.com provides a full range of scripting tutorials in C# from beginner to advanced levels. They can be found here: https://unity3d.com/learn/tutorials/topics/scripting

Activating scripts

Unity scripts will only take effect when they are attached to a GameObject which is active in a scene. GameObjects may be in a scene, yet deactivated. This is an important distinction for behavior developing. Developers may include many agents in a scene, and turn them "off" when debugging to simplify the scene and localize issues. These deactivated GameObjects will not be visible in the scene, and will not affect any execution within the scene. They can be turned on or off manually in the editor, or dynamically within a script.

Class Hierarchy

Using C#, scripts attached to entities may be structured into an inherited class structure. For example, an "Agent" class may hold basic attributes like "health" and methods like "doDamage()" that all game entities will need. A similar approach may be taken for munitions or other non-entity objects which share a baseline of attributes and functionality. This structure is not strictly necessary, but it is helpful to reduce complexity by leveraging the OOP paradigm.

Public and Private Variables

Unity scripts may have both public and private variables and methods. Only public variables will appear in the Unity editor's inspector window, and these variables can be monitored or changed at run time by the developer. As would be expected, only public variables and methods are accessible from outside the referenced script. Private variables can be accessed by pausing execution of the scene and visiting the paired development environment, as discussed in section 5.2.2.

Custom Methods

As with most other languages, custom methods can be built into a C# script. These methods will only be called when there is a reference to them from a MonoBehaviour method. For example, a custom "Die()" method may be built into a generic "Agent" script. Then, when the Update() method is called each frame update cycle, some code may check to see if the agent's health variable is less than or equal to 0. If it is, then it would call the Die() method, and it would execute accordingly, possibly

removing the agent from the scene unless specifically programmed to include a marker for where an entity died.

5.2.2 Scripting Extensions

Due to Unity's massive popularity, major third party software companies have partnered with Unity to integrate its development with existing integrated development environments. Microsoft's Visual Studio (VS) was used for editing and examining scripts for this thesis, as it offers full integration with the Unity editor.[14] VS allows Unity developers to select scripts from within the Unity Editor, and they will open in VS. VS can then be "attached" to that specific instance of the Unity Editor, and variables can be explored within the script at run-time. Breakpoints may also be identified in the script, allowing developers to automatically pause the execution of the script to examine variables and function execution. Unity is shipped with a Unity-specific integrated development environment (IDE) called MonoDevelop. Much of the same functionality is achieved with MonoDevelop; the decision of which IDE to use is largely a matter of preference.

5.2.3 Scripting Without MonoBehaviours

Unity allows developers to leverage the full breadth of C#, making it feasible to write scripts that do not use the MonoBehaviour structure. Although this is possible, the intention of a behavior prototyping environment is to enable quick and dynamic changes to behavioral logic, and the MonoBehaviour construct is not limiting in this regard. Thus, it is typically best to stick to inheriting from the MonoBehaviour in order to prevent confusion with other Unity developers and to avoid unnecessary complexity.[15]

[14]Microsoft's Visual Studio tools for Unity can be found here: https://www.visualstudio.com/en-us/features/unitytools-vs.aspx

[15]Behavior Designer, the commercial off the shelf asset discussed in more detail in chapter 6, does not inherit from the MonoBehaviour class. However, it does explicitly leverage many of the methods found in the MonoBehaviour.

5.3 Limitations of Unity

Although Unity is an excellent game engine, it is not a one-stop-shop for game or simulation development. It is not intended for detailed 3D model construction, nor is it particularly well-suited for use as a high-fidelity analytic simulation tool.

However, myriad tools exist for building models, including the free and open-source Blender project.[16] As with most game engines, Unity has specific file formats which it supports, and conversion methods do exist for most formats not directly supported.[17]

As previously discussed, simulations executed in Unity are inherently time-step simulations, which leads to imprecise determinations of event timing. This is entirely acceptable for most behavior prototyping. The frame-by-frame debugging methods previously discussed can be used to identify and accept these artifacts, which will not likely transfer to a discrete event simulation.

Another complication with Unity is that the execution of scripts is somewhat arbitrary. For example, if multiple scripts utilize the Update() method–which is called once per frame–the order of execution in which these scripts will be updated is somewhat arbitrary and unpredictable. Although a specific order can be set for execution of scripts within a scene, this doesn't necessarily solve the problem of determining the proper order of execution.[18] Consider a situation where an "agent" script with a "doDamage()" method is attached to two soldiers engaged in battle. These soldiers have the same range of weapon, and fire at each other at the same time. Whether or not the order of execution is arbitrary or specified, the order will be incorrect because they both fired at the same time. This typically doesn't matter for behavior prototyping, but it is a limitation that prevents Unity from being used as a full-scale discrete event simulation for analysis. It is important that the behavior developer be aware of this difference from the target simulation.

However, for the non-analytic purposes of prototyping behaviors, Unity performs exactly as needed. The artifacts that we mention here will be eliminated when the

[16] Blender can be downloaded from the following website: https://www.blender.org/

[17] Information on formats supported by Unity can be found here: http://docs.unity3d.com/462/Documentation/Manual/3D-formats.html

[18] Details on specifying the script order of execution can be found here: http://docs.unity3d.com/Manual/class-ScriptExecution.html

behavior is transferred to a discrete event simulation.

5.4 Unity as a Surrogate Development Environment

The use of Unity provides many of the ideal features of a behavior prototyping environment discussed in Chapter 4. Unity revolves around 3D virtual environments, which helps behavior designers better understand the implications of the logical structure of behaviors. It has a flexible, intuitive, and extensible user interface, allowing for customized applications for behavior prototyping. This dynamic development prototyping environment helps to build dynamic behaviors in a timely manner, ideally shortening the iteration cycles for development.

5.4.1 Leveraging 3D Virtual Environments

The built-in 3D virtual environment that exists in a game development environment such as Unity allows designers to test their agent behaviors in a wide variety of circumstances within a short period of time. For example, developers are able to quickly adjust the geometry of the buildings in a MOUT scene and see how that change affects the emergent behavior of a complex behavior tree. Sometimes, simply watching a behavior play out can provide insight as to that behavior's room for improvement.

For example, recall the GoToStore example provided in section 2.1.5, shown again here in Figure 5.3. This HTN appears logical, and simply reading output from a console in a simulation like COMBATXXI would not likely provide insight as to its shortcomings. However, consider a situation where the keys are in the house, and the agent is in the house, but the agent is not holding the keys. Following the logic in this HTN, the agent would then walk to the store, as seen in Figure 5.4. It would not be difficult to add another branch that checks to see if the key is within sight or within range, and proceeds to "pick up" the key if it is. Then the agent would get into the car and drive to the store. This may seem obvious, but isn't always so without the use of at 3D environment.

In this example, it is easy also to move the various objects around and see how it affects the execution logic.

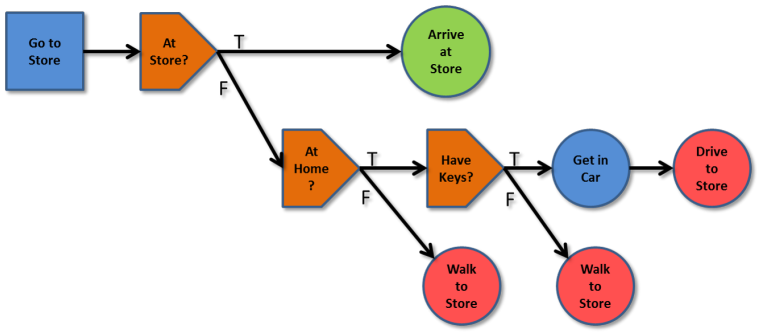

Figure 5.3: An HTN to get to the store.

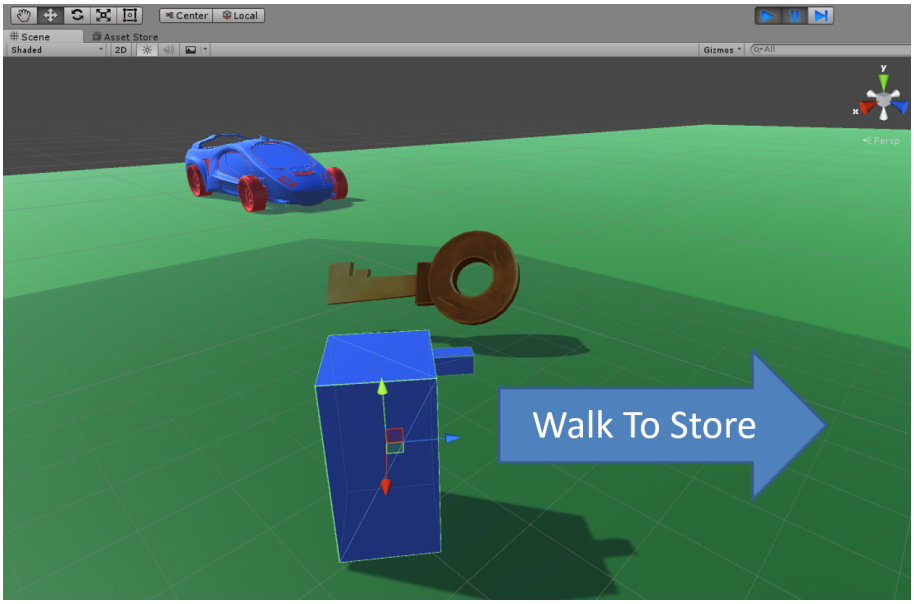

Figure 5.4: The agent walks past the key within plain sight.

5.4.2 Availability of Low-Cost Tools

Unity's large developer base has led to a significant number of models and development tools which can be leveraged by any other developer. The Unity Asset store is the primary distributor of Unity assets. For example, there are over 130 assets for sale in the "Scripting and Artificial Intelligence" section of the Asset Store, ranging in price from $0 to $300. The overall market, however, extends beyond the formal Unity Asset Store. Many open source projects choose to distribute their tools either exclusively via other sites, such as GitHub, or via GitHub and the Asset Store. Regardless of the source, an aspiring behavior developer has many choices when looking for potential time-saving commercial-off-the-shelf solutions. This section provides an

overview of several assets that were considered for potential use in this thesis.

Open Source Projects

Open source projects should always be considered first, as they are free and offer the best opportunity for collaboration. Free and open source projects may be hosted on the Unity Asset store, but they are also found on popular open source websites such as sourceforge.net and github.com. Open source projects should always be approached with caution, and properly vetted within formal security channels of the using organization. The primary benefit to open source code in the context of behavior prototyping is that the code is readable. This allows designers to examine exactly how certain tasks are executed, and change or extend them as needed.

The first resource that was found is an open source project titled *CHP: C# HTN-Planner*, which is an HTN implementation written in C# specifically for Unity. However, it was originally authored for Unity 4, and while promising, seemed to be limited to environments which do not require replanning, unlike COMBATXXI. A more complete tool with more out of the box functionality was required.[19]

Another project which was discovered late in the production of this thesis is titled *behaviac*, which supports BTs, FSMs, and HTNs. This tool was not sufficiently evaluated, but could be used for future behavior prototyping.[20]

Unity Asset Store

Three projects were pursued that exist or have existed within the Unity Asset Store: Stagpoint's HTN Planner, Rival Theory's RAIN AI, and Opsive's Behavior Designer.

A development company called Stagpoint had produced a now unsupported HTN implementation within Unity. The description and online demos show an HTN implementation which appear similar to the HTNs in COMBATXXI.[21] Unfortunately, it was removed from the Asset Store prior to the start of this thesis, and efforts to contact the development team were unsuccessful.

[19]This HTN implementation can be found here: https://sourceforge.net/projects/chpplanner/
[20]*Behaviac* can be found here: https://github.com/TencentOpen/behaviac
[21]Stagpoint's HTN Planner description can be found here: http://stagpoint.com/planner/

Rival Theory's RAIN Artificial Intelligence platform is free in the Unity Asset Store. As a result, it is widely used and there are plenty of user-created tutorials. RAIN has a unique implementation of BTs that is relatively easy to use and potentially useful for creating HTNs in Unity.

However, the source code is proprietary, and users doesn't have a way to know how a certain automated behavior works under the hood. For example, RAIN allows for easy creation of a visual sensor that is based on a line of sight (LOS) algorithm, but the user can only guess if the determination is based on a raycast to the center mass of the target, or some other point. Without specific knowledge of how a given algorithm works, some level of confidence in the ability to translate that code to COMBATXXI HTNs is lost. Furthermore, there is no current support for extending RAIN AI, so custom behaviors meant to prototype COMBATXXI HTNs could not be built.

The final tool which was explored for use as a prototyping tool was Opsive's Behavior Designer.[22] Behavior Designer costs $105 per seat license in the Unity Asset Store, including all three add-on packs which were experimented with for this thesis. While not free, this cost is minimal in comparison to the in-house development costs of building a similar tool. Furthermore, the online forums and responsive support team ease the use and troubleshooting of Behavior Designer behavior trees.

Behavior Designer provides an extension of the Unity editor which allows designers to build custom behavior trees in a separate editor window and watch them execute at run-time. A simple example of a behavior tree in Behavior Designer is shown in Figure 5.5. Each node in the tree has an associated script, and each script can easily be opened in the users IDE of choice. Scripts allow the user to examine exactly how a given task is being executed, and to improve upon that task if they desire. New custom tasks can also be built and integrated into the Behavior Designer development environment. It is the flexible and extensible nature of Behavior Designer which led to its selection for use in this thesis.

[22] An overview and documentation of Opsive's Behavior Designer can be found here: http://www.opsive.com/assets/BehaviorDesigner/

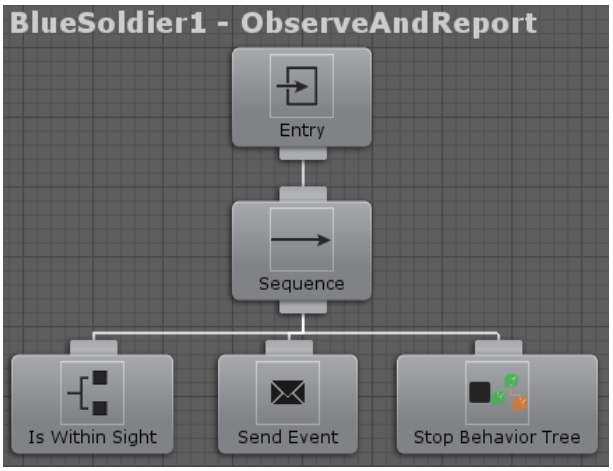

Figure 5.5: A behavior tree which sends an event after detecting an enemy.

5.4.3 Proceeding with Unity and Behavior Designer

Although no HTN implementation exists for Unity which is similar enough to the COMBATXXI HTNs to provide one-to-one mapping of behaviors, the combination of Unity and Behavior Designer provides behavior designers a sufficiently flexible environment to experiment with behaviors that are transferable to COMBATXXI. Chapter 6 provides a detailed description of the development environment that was built to prototype behaviors for this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6:
Using Behavior Trees in Unity to Emulate HTNs

This chapter provides additional examples of behavior trees that are used to emulate COMBATXXI HTNs. While the basic example of the GoToStore behavior was demonstrated in section 2.2.2, another behavior from a well-known COMBATXXI HTN tutorial is used here to demonstrate a more complex behavior. Lastly, attempts made at prototyping new behaviors within Unity for translation to COMBATXXI are detailed.

6.1 Utilizing the Prototyping Environment

Details on the construction and layout of the prototyping environment have been encapsulated in Appendix A. This section discusses the methodologies and concepts used for building the prototyping environment, and following sections will discuss the theory and practice of structuring behavior trees to emulate COMBATXXI HTNs.

6.1.1 A Model of a Model

Although some commercial off the shelf (COTS) tools were used, the majority of this prototyping environment was built from empty scenes within Unity. Building it was a piecemeal process; objects were built as needed during the prototyping process. The tools needed to replicate the GoToStore behavior, for example, were smaller and less robust than those needed to prototype the more complex behaviors that actually transfer to COMBATXXI. However, Unity's ease of use can be a siren's call to develop interesting tools or behaviors that might not actually transfer to the target system. After much trial and error in transferring behaviors to COMBATXXI, the "model of a model" approach became the cornerstone of the methodology used in this thesis. This means that no object or behavior should be built in the prototyping environment which will not transfer to the target simulation. In general, designers should be cognizant of the representational space of the target simulation when constructing the prototyping environment.

Not all aspects of the target simulation can or should be emulated in Unity, but that doesn't mean that an abstraction of an abstraction isn't still useful. For example, the basic action of engaging an enemy target in COMBATXXI is a complex process. It involves target selection, weaponeering,[23] starting the engagement, operating the weapon, creating the munition object, determining the munition accuracy and fly out, assessing the impact, and assessing the damage [39]. An adequate target engagement process in Unity might include target selection, raycasting to verify line of sight, and assessing damage. In this case, the behavior designer must be aware of the nuance of the Engage Functional Module in COMBATXXI and how it might affect the validity of the behavior. Behavior designers should emulate the least amount of detail as is necessary for their particular behavior. It is easy to get bogged down in wanting to perfectly emulate the functional module as it exists in COMBATXXI, but doing so would defeat the purpose of leveraging the time savings that are inherent in using a surrogate development environment for prototyping purposes.

Another example of when less resolution in Unity is appropriate is the observation process. As with many "off the shelf" tools, Behavior Designer has built-in tasks for checking a field of view out to a given range. The task may then return whether or not an enemy is within the field of view. In COMBATXXI, the observation process is driven by a field of view and a field of regard.[24] While the ability to model a field of regard may be necessary for some high-fidelity military simulations, setting the field of regard equal to the field of view for the purpose of prototyping behaviors in Unity is completely acceptable.

Whereas an entity in COMBATXXI may take some time to "find" an enemy in its field of view, the detection of a given object in the field of view of an entity in Unity would happen within one "tick", or the time it takes the computer to iterate through the graphics pipeline–typically sixty times per second. This means that the time to detect an entity in COMBATXXI will have far more variability than in Unity. If we are modeling the observation process, this is not acceptable. However, if we are modeling a given reaction to an observation event, this abstraction probably will be

[23]Weaponeering tables are used in COMBATXXI in order to select the best available weapon and ammunition for a given target at a given range.

[24]Field of view is the area within the field of regard that the entity is currently focused on.

acceptable. It is the duty of the behavior designer to use good judgment on what level of detail to model. Typically, it will be the same behavior designer who reaps the rewards or pays the punishment of wasted time if this is not done judiciously.

6.2 Emulating COMBATXXI HTNs with Behavior Trees

This section combines the fundamentals of HTNs detailed in section 2.1.3 with the lessons learned in building behavior trees for use as HTNs. Using the "model of a model" concept, behavior trees that do not translate to COMBATXXI are ultimately useless for our purposes. Developing a baseline of understanding for how behavior tree components translate to HTN components will help ensure developers build behaviors which are more likely to transfer smoothly to COMBATXXI.

6.2.1 Additional Considerations Regarding the Anatomy of an HTN

Most components discussed in this section were identified in the "Anatomy of an HTN" section, section 2.1.3. However, some additional concepts are included here which were found to be important aspects of HTN design.

Initializing the HTN

Variables are managed in several ways within an HTN, but they are almost always gathered when the HTN is initialized. This can be modeled in Behavior Designer's behavior trees explicitly by the use of an "initiation" branch of a tree, or implicitly by the use of the "shared variables" tab, which allows behavior trees to store variables in one place for use throughout the tree. These variables can be provided as input if the tree is instantiated via a behavior tree reference (see section 6.2.1), procedurally from within the tree itself, or manually by a designer entering values. There is no incorrect way to do this, so long as the designer knows how the values will translate to the initialization of the HTN.

The isCommander Pattern

When architecting behaviors for COMBATXXI, it is often easier to give a behavior to all entities in a unit, and screen the execution of that behavior or a portion of that behavior with the "isCommander" pattern. This pattern simply uses a constraint node to check whether or not an entity is the commander of his unit at a given echelon. This can be emulated by including a public Boolean "isCommander" to the AgentState class script, as detailed in Appendix A.

Modeling Interrupt Goal Tasks

The intent behind interrupt goal tasks in COMBATXXI HTNs is to prevent the planner from planning too far into the future and to represent tasks that take some time complete [18]. The HTN pauses at this node, and then begins again once the interrupt goal task is complete and an appropriate replan event occurs. This helps to prevent too much computational waste by planning too far into the future when the world state will likely necessitate a change to the plan after the execution of the interrupt goal task.

This is not easily replicated in behavior trees, because they do not plan. Therefore, an interrupt goal task as modeled in a behavior tree is no different than any other task which takes some time to execute. However, it is important for designers to keep in mind logical stopping points for the HTN to pause. This is usually the last node in a sub-branch of a tree, if that branch is executed as a child of a sequence or selector node. In the GoToStore example, each of the movement nodes demonstrates how to model interrupt goal tasks, as seen in Figure 2.9 on page 22.

Replanning with Model or Goal Tracker Events

In order to force a behavior tree to emulate replanning as an HTN would, it needs to start again from the top. This is accomplished in one of several ways. Either a node which is part of a sequence node fails, or a node which is part of a selector node succeeds. This returns execution up to the next highest node. If this happens to send execution all the way to the top of the tree, and the tree is set to repeat upon completion, then this will effectively emulate a replan trigger in a COMBATXXI HTN. Alternatively, a repeater node may be placed at the top of the tree, under

the entry node, for an explicit visual reminder that the tree is set to repeat. This is demonstrated in the GoToStore example in Figure 6.1

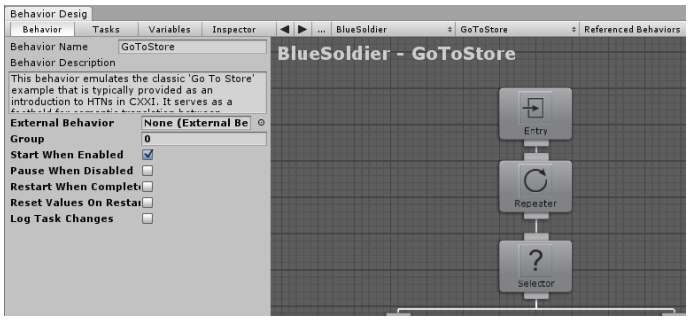

Figure 6.1: Placing a repeater node at the top of the tree or selecting the "Restart when complete" box will allow for emulation of a replan event.

Goal Nodes in Behavior Trees

The use of repeater nodes in a behavior tree means that it will not stop unless explicitly told to do so. However, the "stop behavior tree" node works nicely for this. Although the "stop behavior tree" node does not itself represent the goal, it is a way for the designer to implicitly mark that the goal has been achieved. For example, if a soldier's task is simply to observe and report, then after he has reported visual contact of the enemy, the goal has been achieved. This example is demonstrated in Figure 5.5.

Adding COMBATXXI-like Goals in Behavior Trees

One of the more common ways to add a goal to an entity in COMBATXXI is to use the addGoal() method. This allows a designer to add an HTN to any entity, and to enter the parameters directly from within this function call. There are two main ways this can be emulated in Behavior Designer's behavior trees. First, a designer can use the behavior tree reference node. At run time, this node imports a common behavior tree from the Unity project as a new branch where that node had existed. Variables can be synchronized and passed to the tree, or the tree can rely on existing shared variables from within the parent tree.

The second way to emulate the addGoal() function call is to use the start behavior tree node. This node requires that the behavior tree to be used already exists, and it

should have a unique "group" number associated with it. This behavior tree should be turned off initially, but attached to the entity that the designer expects to need it later.

Combining these concepts allows designers to prototype basic HTNs within Unity. There are likely other ways to accomplish this, and no claim is made that this method is optimal. However, subsequent examples will help to demonstrate that it does work.

6.3 Bounding Overwatch

The Bounding Overwatch tutorial is used to introduce new HTN designers to the necessary concepts and user interfaces.[25] This section describes the Bounding Overwatch scenario as recreated in Unity.

The Agents

There are 8 soldiers, comprising a single infantry squad. There is a squad leader and an assistant squad leader. Otherwise, all soldiers are identical, and are instances of the Soldier prefab as identified in Appendix A.

The Environment

This scenario is built on a flat plane as terrain, and there are six buildings along a simulated road. The plane includes a navigation mesh, as is required in Unity in order to allow the NavMeshAgent component of the prefab instances to move about the scene easily. The road is marked by the path between the squad's starting location and the last way point.

The Behavior Goal

This behavior is used in COMBATXXI to dynamically determine whether a squad should split up and move in bounding overwatch or stick together and move in a single formation. Bounding overwatch is similar to a leapfrog scheme of maneuver, where one unit provides cover while the other moves. Each unit alternates movement

[25]The Bounding Overwatch tutorial can be found at the NPS "Tutorials and Examples" page on the WSMR wiki website: https://cxxi.wsmr.army.mil/confluence/display/NPS/Tutorials+and+Examples

until the maneuver is complete. The determination between whether or not the squad will move in bounding overwatch is made based on the number of buildings along a road. This number is retrieved in COMBATXXI using a Python function that simply returns the number of buildings along a road. In order to emulate this in Unity, a custom task was built which counts the number of GameObjects with a certain tag within a certain range and sector relative to the using entity. Once this number is retrieved and stored, the behavior tree shown in Figure 6.2 is able to direct the execution flow to the appropriate branch. If there are four or more buildings along the road, the flow of execution is directed to the left side of the tree, which starts the bounding overwatch behavior for the squad leader and assistant squad leader.

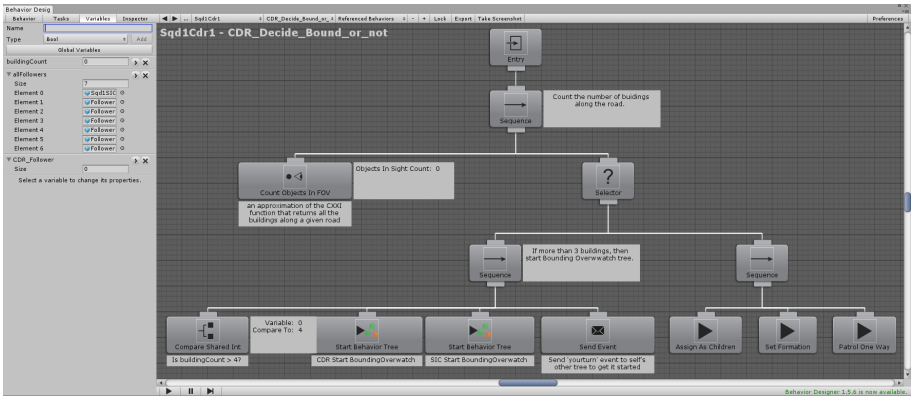

Figure 6.2: This behavior tree enables the squad leader to count the number of buildings as a means of determining whether or not to move by bounding overwatch or by a squad wedge formation. The node labeled "Compare Shared Int" in the bottom left, used as the leftmost node in a set of sequence child nodes, emulates a basic constraint node as in COMBATXXI HTNs. The two "Start Behavior Tree" nodes initiate the behavior trees for the squad leader and assistant squad leader. These trees are detailed in Figure 6.3.

If the squad moves in bounding overwatch, execution moves to the two trees started by the "Start Behavior Tree" nodes that are seen in the bottom of Figure 6.2. The squad leader's subsequent behavior is similar to the assistant squad leader's behavior, as shown on the left and right side of Figure 6.3, respectively.

If the number of buildings returned is less than or equal to three, then the squad will simply move in a single wedge down the middle of the road to the final way point.

This scenario, built in Unity, enables developers to quickly change the orientation

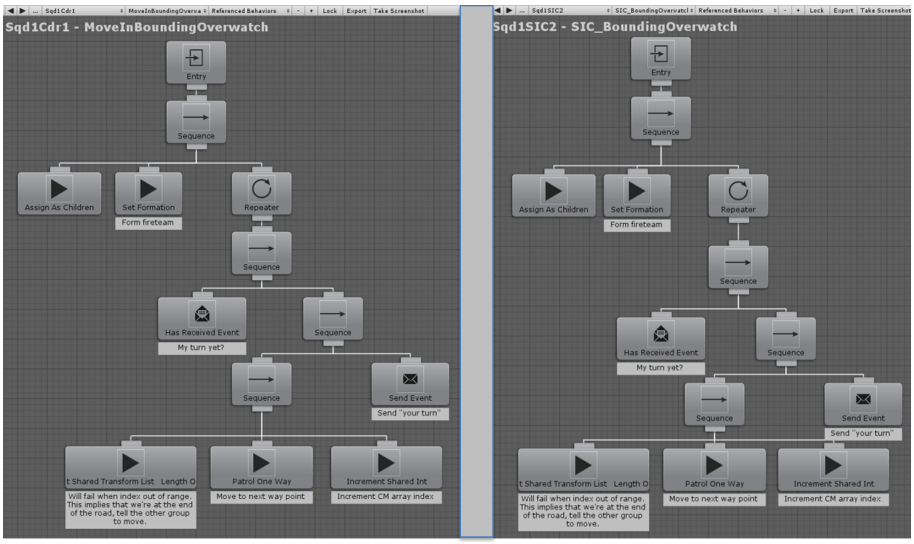

Figure 6.3: These two behavior trees are identical, except for their variables. The behavior trees for the squad leader and the assistant squad leader (i.e. second-in-command, or SIC) to take their respective halves of the squad along the appropriate side of the road using the bounding overwatch tactic. This provides an example of the flexibility of behavior trees.

and number of buildings, the balance in the number of buildings on each side of the road, as well as the size and movement route of the squad. This rapid testing can help to identify weak points in the behavior logic and find the most appropriate way to make the behavior more robust to changes in the simulation world state.

6.4 Prototyping New Behaviors

The Platoon Security Patrol is a COMBATXXI scenario that has been used in the past by OAD, but for which no current implementation exists that uses HTNs. For this reason, it was selected as a target for potential revision as part of this thesis.

6.4.1 The Scenario in COMBATXXI

The Platoon Security Patrol scenario involves a platoon of Marines that conduct a mounted patrol, moving south to north along a main supply route, or road. At some point during their patrol, one of the vehicles strikes an improvised explosive device (IED), and the platoon comes under small arms fire from enemy positions

to the west. The Marines then call for supporting fire in the form of IDF from a nearby 60mm mortar team and close air support (CAS) from nearby Cobra attack helicopters. The platoon then maneuvers to the north and flanks the enemy positions. The scenario, as it exists in COMBATXXI, includes 60 blue forces and 77 red forces.

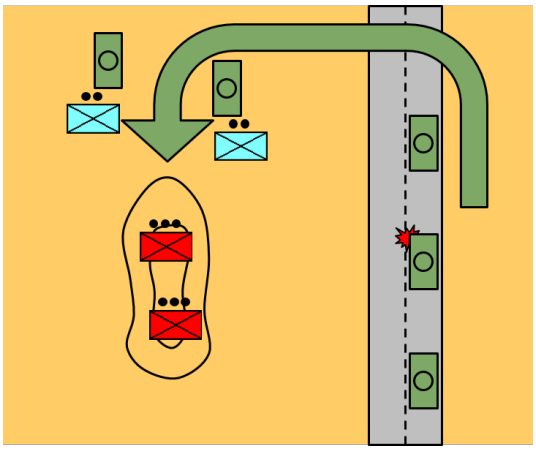

Figure 6.4: An overview of the Platoon Security Patrol provided by OAD.

In order to simplify the scenario, the blue forces were reduced to 33, and the red forces were reduced to 30. Additionally, the CAS assets were removed, so the Marines in the reduced scenario would have to rely only on the on the IDF assets. This simplification allowed for increased focus to be placed on the development of behaviors. In keeping with the qualities discussed in section 4.2.1, the scenario was analyzed and distilled into modular actions which needed to be modeled in the prototyping environment. That distillation is specified in the following two subsections, divided by agent affiliation (e.g. red or blue) and ordered chronologically.

Blue Agent Actions

1. Load dismounted soldiers into their assigned vehicles.
2. Order vehicles to patrol along main route.
3. React to IED-induced catastrophic kill of the lead vehicle by taking a pre-specified secondary route which will be used to flank the enemy position.
4. Submit call for fire to nearby forward observer.
5. Call to cease fire when in position to assault the enemy.
6. Dismount troops when within appropriate range of the enemy.

7. Conduct frontal assault through enemy positions.

Red Agent Actions

1. Engage enemy with small arms fire when IED detonates.
2. Take cover when enemy fire support assets begin firing on friendly positions.
3. Move to secondary fighting positions when force strength is reduced to 60%.
4. Defend secondary fighting positions to the death.

6.4.2 Platoon Security Patrol in Unity

After the scenario was reduced and the agent actions outlined, development began in Unity. A similar scene was constructed, and a subset of agents were created to build a minimal working example of the scheme of maneuver depicted in Figure 6.4.

Once the tasks were specified, development began in Unity, without periodically updating the corresponding behavior in COMBATXXI. This turned out to be problematic, as is discussed in section 7.1.1.

The Agents

There are a total of five entities in this scenario. Three are blue soldiers, one is a blue vehicle, and the last entity is a red soldier.

The Environment

The environment is similar to the Unity version of the bounding overwatch scene, with the exception of being larger and allowing for more maneuver possibilities. There is a graphically depicted road, and buildings based on the prefabs specified in Appendix A. The scene is depicted in Figure 6.6.

The Behavior Goal

The behavior built in this scene was used as a demonstration of the capabilities of the prototyping environment, and does not include all components of the actual platoon security patrol scenario. The blue soldiers begin by loading into the vehicle and patrolling north. They then dismount just short of reaching an enemy IDF zone

where mortar rounds are actively falling. Once they have dismounted, they follow a scripted route by way of several way points in order to conduct a frontal assault on the enemy position. The corresponding behavior tree, which belongs to the squad leader, is depicted in Figure 6.5. This point in the scenario is depicted in Figure 6.6.

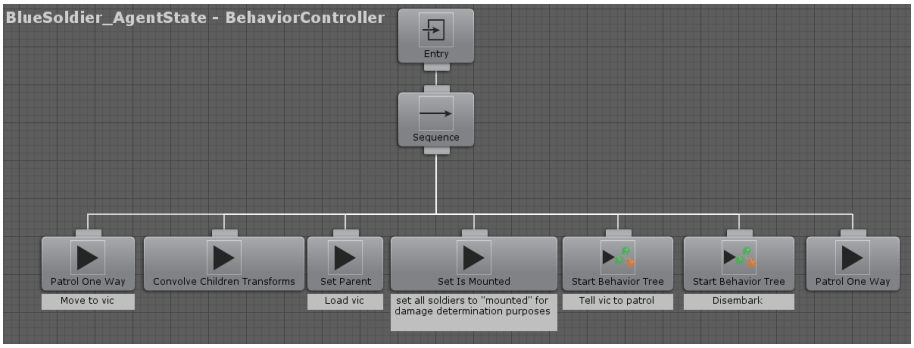

Figure 6.5: This linear behavior tree is simply a starting point for developing original behaviors for the Platoon Security Patrol. Note that "vic" is short for "vehicle."

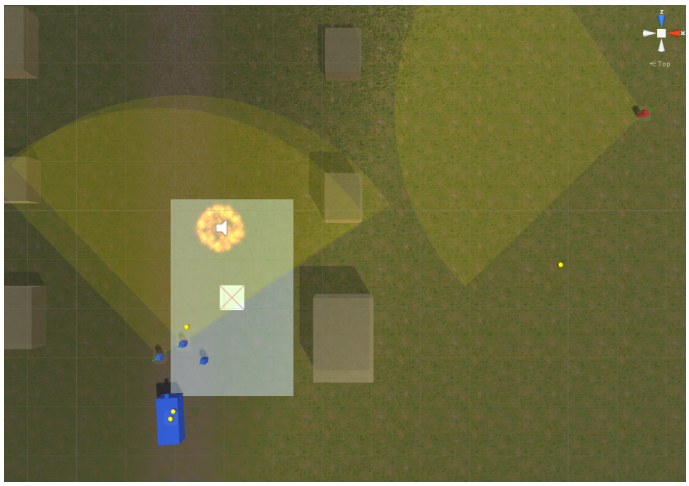

Figure 6.6: A Unity scene built to prototype the Platoon Security Patrol scenario behaviors

Meanwhile, the enemy red soldier is calling for fire on a location near the blue patrol route. This behavior was built as a custom task called "call for fire", which places a red "X" at a center point and initiates 60mm mortar round detonations within a specified box around that "X". This behavior and its associated prefabs are specified in more detail in section A.2.2 on page 93.

All soldiers are able to engage their enemies with direct fire once they are within range, and once either all blue soldiers are dead or all red soldiers are dead, the scenario is over. There is an inactive IED in the scene as well, which could be used for further prototyping of reactions to IEDs.

6.4.3 Translating to COMBATXXI

After a baseline of functionality was developed in Unity, development efforts were shifted to the COMBATXXI scenario. This proved to be the most difficult part of the process, for two main reasons. First, the tasks in Unity do not map to COMBATXXI except on a conceptual level. Second, the development environments both have their own nuance that tend to thwart novice developers. For example, a mounted patrol seemed to not be moving in the COMBATXXI scenario, and significant development time was spent trying to find a reason for this within the HTN. The problem was resolved by turning off the modeling of "detailed ground formation."

However, with assistance from experienced COMBATXXI developers, a partially complete scenario was developed that does implement indirect fires, an IED detonation, and the use of a secondary route as a reaction to the IED detonation. No red soldier actions were programmed, and the blue soldiers stop short of dismounting to conduct the frontal assault. This scenario in action is depicted in Figure 6.7.

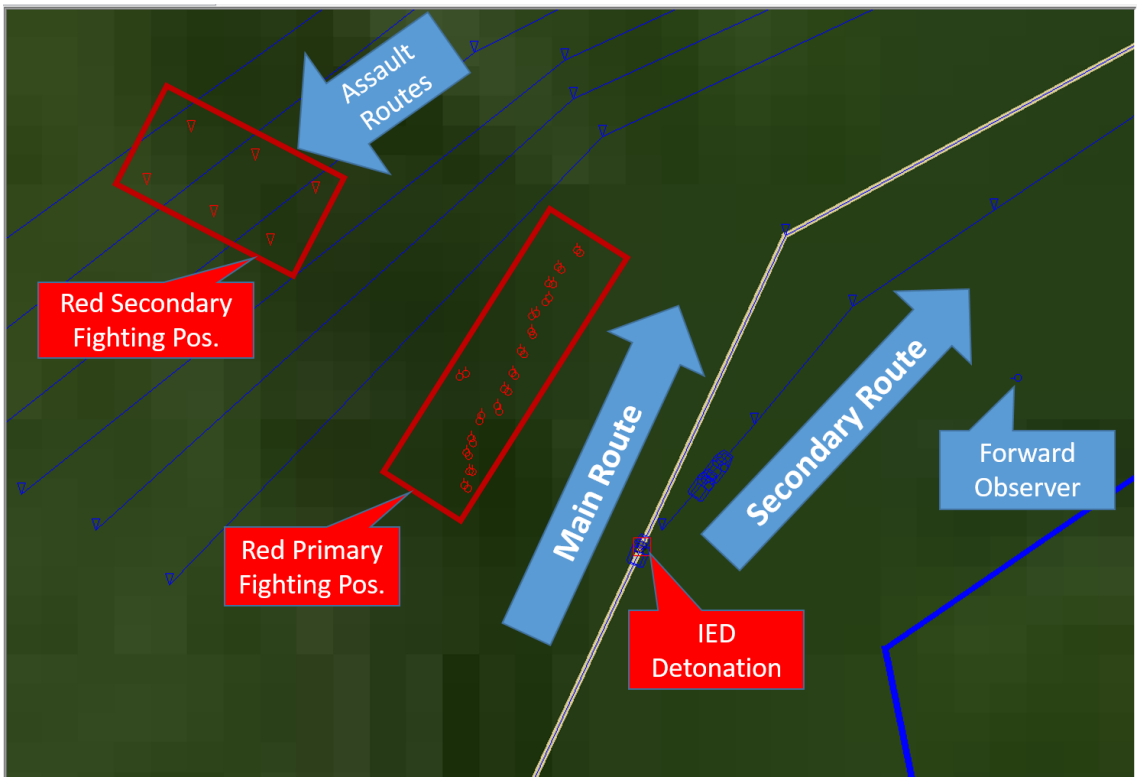

Figure 6.7: In the Platoon Security Patrol scenario, a three vehicle convoy patrols north along the "Main Route" until the lead vehicle is killed by an IED. The remaining two vehicles then take the "Secondary Route," and call for fire support from the nearby forward observer. The flanking behavior, frontal assault via the "Assault Routes," and the red soldier behaviors were left as future work.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 7:
Recommendations and Conclusions

Developing behaviors for COMBATXXI scenarios is the current bottleneck for development of scenarios for analysis. Finding ways to more efficiently build and test behaviors will continue to be the lowest hanging fruit for those seeking to improve the scenario development process. Using visually intuitive tools such as game engines is a previously unexplored methodology which may provide benefit inherent in working with a 3D environment. This thesis involved building a behavior authoring and preliminary testing environment using the Unity3D game engine. The differences in development environments between the prototyping environment and the target simulation created unforeseen barriers. Specific work flows were explored out of necessity, and suggestions on their utility are provided in this chapter. Building upon the already robust COMBATXXI development environment is explored as potential future work. Additionally, there are other ways in which game engines could be used to improve the scenario development process, such as leveraging existing interoperability standards to use the game engine as a replay tool for behavior testing.

7.1 Recommendations

Developing behaviors in a surrogate environment has many potential benefits. It prevents developers from being tempted or directed into including prototyped behaviors in production-level simulation environments. It also enables developers to leverage gap-filling technologies from outside the target simulation, or to evade the whole-model complexity that may exist in the target simulation. However, developing in a surrogate environment requires that the developer be intimately familiar with the capabilities and limitations of both environments. Additionally, if there is not an automated transfer process, then the developer must manually map code from the surrogate environment over to the target simulation. While this may prove worth while, it is worth considering up front the approach that will best fit the surrogate and target simulation pair as well as the experience of the developer.

7.1.1 Choosing a Development Work Flow

Developers have a choice when working with a surrogate environment as to how frequently they transfer prototyped behaviors to the target simulation. There are benefits and detractors at both extremes. Developing an entire scenario in the surrogate environment before visiting the target simulation allows the developer to immerse their thoughts in the code and model nuances of the surrogate environment. Contrastingly, frequently visiting the primary development environment helps to ensure that the behaviors built in the surrogate environment are feasible in the target simulation. Both approaches are potentially valid, and are discussed in more detail below.

Developing in Series

When working in a surrogate environment that is significantly different from the primary development environment, it is easy to become immersed in that environment. This is particularly true if the surrogate environment is easier to use. It is a natural inclination to not want to go back to programming in the more complex environment with a more complicated model, and this represents a possible pitfall for the inexperience developer.

However, for an experienced programmer, this may be the most appropriate approach. If they are confident that what they have prototyped will transfer, the developer could develop whole behaviors or even entire scenarios in the surrogate environment before turning back to the target simulation. This process is visualized as a UML sequence diagram shown in Figure 7.1. This method is similar to the waterfall method of software development, except that it is typically one person conducting each phase of development [40, p. 329].

The inherent risk with this approach is that development efforts in the surrogate environment that do not transfer to the target simulation are wasted. This waste ties back to the "model of a model" concept which was discussed in section 6.1.1. If a non-mapping element of a behavior exists fairly high up in the behavior tree, for example, the entire behavior tree may need to be refactored. This was the approach initially taken for this thesis.

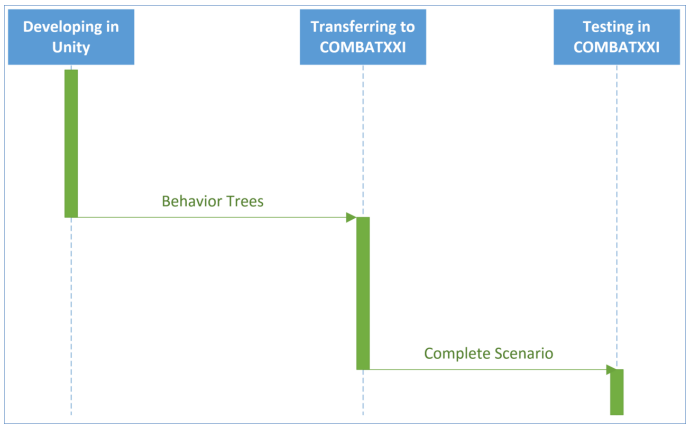

Figure 7.1: Developers may choose to build entire scenarios in the surrogate environment first, incurring the risk that some behaviors won't transfer. Ideally, this process is conducted once per scenario.

Developing with an Iterative Approach

Alternatively, developers may prototype individual behaviors or even small portions of individual behaviors, and immediately transfer them to the primary development environment. This process is visualized as a UML sequence diagram shown in Figure 7.2. This method helps to ensure transferability, and prevents significant development time in the surrogate environment from being wasted. This can be viewed as an implementation of the spiral development process [41, p. 64].

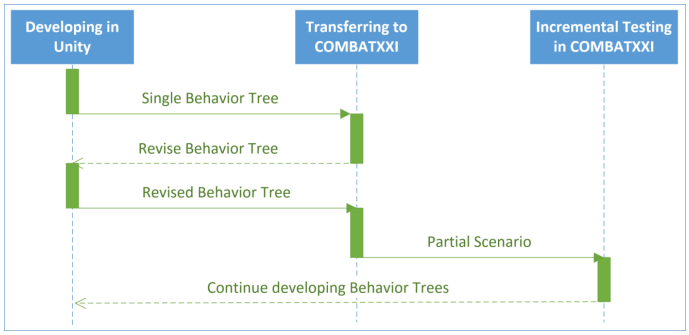

Figure 7.2: Developers may choose to frequently transfer existing behaviors to the target simulation in order to prevent excessive refactoring. This process is repeated as many times as is necessary, depending on the number of behaviors required.

For an experienced developer, this method of repeatedly visiting the primary development environment may be wasteful. Going back and forth between two differ-

ent domains that likely use different programming languages and require familiarity with various model specifics can be confusing. For example, developers in Unity should be familiar with the MonoBehaviour lifecycle depicted in Appendix B, while COMBATXXI developers need to thoroughly understand discrete event simulations. Even an experienced developer will take some time to move local variables, frequently used functions, and scenarios specifics into their short-term working memory. For inexperienced developers who do not have a thorough understanding of what is possible in COMBATXXI, this may be the only feasible approach.

These two methodologies exist on opposite ends of the spectrum. On one side, an ideal developer who does not make mistakes could build an entire scenario in the prototyping environment before transferring anything to the target simulation. On the other end of the spectrum, a complete beginner in the target simulation or the prototyping environment may have to use much shorter iterations to ensure transferability. However, the reality is that any single developer will likely evolve at some point between these two extremes. For example, our beginner developer may begin to gain familiarity with function calls, modules, or libraries from the target simulation that map to the functionality of the prototyping environment. This will allow the beginner developer to spend less time bouncing back and forth between environments. Contrastingly, our experienced developer may undertake a project which requires him to learn a new portion of the target simulation with which he was not yet familiar. In this case, he will need to return to the target simulation more often when prototyping, until he becomes expertly familiar with said portion of the target simulation.

7.1.2 Building an HTN Implementation in Unity

A significant portion of the cognitive effort required to model HTNs in Unity is dedicated to translating between the semantics of behavior trees and the semantics of HTNs. Behavior Designer, the Unity asset used for this thesis, was built by a third party development group. It may be feasible to develop an HTN implementation in Unity which more directly translates to HTNs in COMBATXXI. Certain inherent limitations, such as the discrete event nature of COMBATXXI and the time-stepped nature of game engines, would difficulty in creating a direct translation. However, if the semantic structures of the behaviors are the same, then this would be less of a

concern. Additionally, implementing discrete event simulations in Unity is not outside the realm of possibility, as indicated by current work at the MOVES Institute [42].

7.2 Future Work

There are other ways in which the development process may be improved. COMBATXXI and its associated development suite may be extended to enable faster authoring, testing, and debugging. Game engines and other visualization methods may be explored to provide a visual component to the testing and debugging processes within the current COMBATXXI development cycle. Behavior standards across the DoD M&S community could be developed in order to ensure better transferability of behaviors between existing simulations. Additionally, Unity or other game engines could be explored as potential platforms for analytic simulations.

7.2.1 Extending COMBATXXI

Section 3.3.1 suggests options for extending COMBATXXI as an alternative to the prototyping methodology suggested in this thesis. Although this thesis did not explore any specific in-roads for extending the COMBATXXI development environment, the difficulties experienced when transferring behaviors from the Unity to COMBATXXI demonstrated that prototyping the behaviors within the COMBATXXI framework could be preferable. Specifically, the main benefits of using Unity are the 3D virtual environment and the ability to debug behaviors at run-time. If either of these were achievable within the COMBATXXI framework, that would likely be a more preferable solution to prototyping in a surrogate environment.

7.2.2 Using Unity3D as a COMBATXXI Playback Tool

Several tools exist that allow Unity developers to integrate DIS functionality into their simulations, provided they are willing and able to build out the DIS enumerations that would be sent from COMBATXXI.[26,27] COMBATXXI can provide DIS output

[26]Calytrix's LVC Game, built for Unity 4.x, provides a commercial off the shelf solution, and can be found here: http://www2.calytrix.com/products/lvcgame/introduction/

[27]Karl Jones' DIS-Unity provides an open source solution, also for Unity 4.x, and can be found here: https://sourceforge.net/projects/unitydis/

files which could be played back over a network and visualized in Unity. This would allow developers to visualize their scenario output in a way that is not currently possible. This method would have to overcome the typical array of interoperability concerns, such as DIS enumeration configuration and terrain matching. However, the payoff could well be worth the effort.

7.2.3 Unity3D as an Analytic Tool

When developing scenes which had stochastic processes in them, such as the platoon security patrol's indirect fires, it became evident that Unity could be used as a tool for analysis. Although there are known problems with the use of time-stepped simulations for analytic purposes, there are still analytic simulations that are time-stepped. For example, Map Aware Non-uniform Automata (MANA) uses time arbitrary steps to mark simulation time.[28] NetLogo, a popular tool for building large scale agent-based simulations which have been used to model complex social systems, uses "ticks" to mark its time. If Unity were used for analysis under the same precepts which guide the use of other time-stepped simulations, then it would be equally valid. This thesis focuses on prototyping behaviors for use in a Discrete Event Simulation (DES). Perhaps the use of Unity as a tool for analysis would be more appropriate for simple, agent-based simulations. These simulations often accept the implicit issues of a time-step simulation, as evidenced by the continued use of MANA and NetLogo. In these simulations, developers aim to leverage the first theme of complexity research, as identified by Ilachinski, that "surface complexity can emerge out of a deep simplicity" [3, p. 6].

7.2.4 Standardization of Behavior Specification

One way in which M&S professionals could more reliably leverage behavior development surrogate environments would be to develop a standardized framework for behaviors. Simulation Interoperability Standards Organization (SISO) is the "simulation interoperability through reuse and standards" [43]. SISO currently has no such standard, likely because behaviors are often authored in different programming

[28] Map Aware Non-uniform Automata (MANA) is a simulation tool developed by the New Zealand Defence Technology Agency (DTA).

languages and using different structures, as discussed in Chapter 2. The same logic that guides good transferability between simulations as embodied in other SISO standards could also guide behavior development in surrogate development environments to target simulations, ensuring better transferability.

7.2.5 Conclusions

Although the complete goal of this thesis was not accomplished within the specified time frame, a workable environment for building COMBATXXI HTNs in Unity was completed and the basic premise that such an approach is viable was validated. That environment was used to emulate existing HTNs and to prototype new HTNs. However, there is still much more that can be done to create better behaviors for use in military simulations. These improvements may involve additional work with game engines as surrogate environments or visualization tools, or they may involve extending the current suit of COMBATXXI development tools.

Regardless, agent behaviors will continue to be a primary enabler of entity-level combat simulations, and developers' ability to build reusable and robust behaviors in a timely manner will play a key role in determining the timeliness with which the DoD analytic community can provide M&S-enabled insight to decision makers.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
The Prototyping Environment

This appendix specifies the prototyping environment which has been built in pursuit of the completion of this thesis. Unity3D version 5.3.1 and Opsive's Behavior Designer version 1.5.5 were used in its construction.

To explore this tool, a user must have purchased Behavior Designer and have downloaded the appropriate packages specified here in order to use the custom behavior tree tasks which were built for this project. Then, download the Unity project at the linked repository.[29] There is a complete repository maintained by the MOVES Institute which holds the fully functional Unity project as well as the appropriate seat licenses. Subsequent theses can leverage this repository.[30] The user should then set aside the files titled "Behavior Designer Custom Tasks", so that it does not interfere with the installation of Behavior Designer. Once Behavior Designer is installed, locate the appropriate subfolder within the file hierarchy, and insert the base folder titled "CustomCXXI" tasks there. This will ensure proper integration with the Behavior Designer architecture.

A.1 Behavior Designer Behaviors

Although Behavior Designer has a good deal of out-of-the-box functionality, some extensions were made specifically to allow for behaviors which are more similar to COMBATXXI behaviors. This section specifies how those behaviors are organized and what their intended purposes are.

A.1.1 Custom Tasks

In order to keep the integration of these custom tasks with the larger Behavior Designer system as simple as possible, all tasks were kept in a single folder. This is not

[29]The following bitbucket repository holds the partial project, with all proprietary Behavior Designer components removed: https://bitbucket.org/nps_millerthesis/unity_htn_prototype

[30]To request access for and download the full Unity project from within the nps.edu domain: http://pogy.ern.nps.edu/gitbucket/dmmiller/unity-htn-prototyping

strictly necessary, but doing otherwise would make integration of these scripts needlessly difficult for subsequent users. However, some tasks may inherit functionality from other tasks, and therefore will not all show up in one place within the Behavior Designer tool, as shown in Figure A.1.

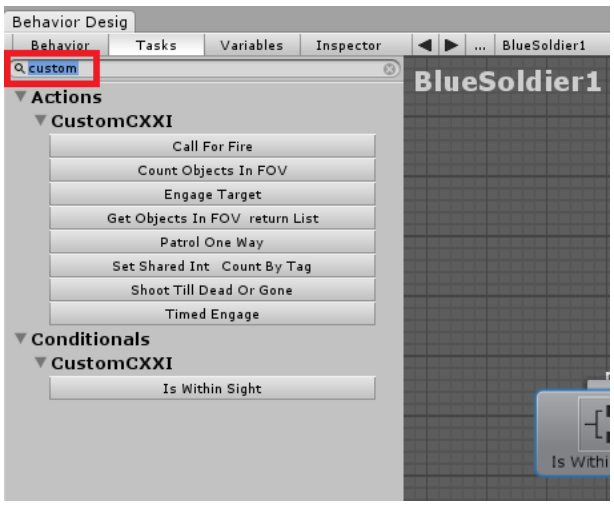

Figure A.1: Type "custom" into the search bar of the task tab within the Behavior Designer editor to find all uploaded tasks built for this thesis.

Each task is contained within a C# script, and documentation is included for each task within the script.

A.1.2 Custom Behavior Trees

Custom behavior trees were used whenever possible, as they allow for modularity and code reuse. Repetitive compound tasks, such as getting in a vehicle, can be wrapped into a custom behavior tree and used from within a larger behavior tree to simplify the behavior design process.

The behaviors which were built into behavior trees for this project are as follows:

EmbarkVic

This behavior, shown in Figure A.2, is attached to a unit leader who may or may not have subordinate agents within the unit. It must be given a GameObject which represents the parent vehicle. This emulates the Transport Group concept from

COMBATXXI. The squad leader then moves the squad to the vehicle in a default wedge formation, and loads them into the vehicle by convolving all dismounts to the center point of the parent vehicle. This is accomplished in Unity by moving the position of each soldier to the center point of the vehicle.

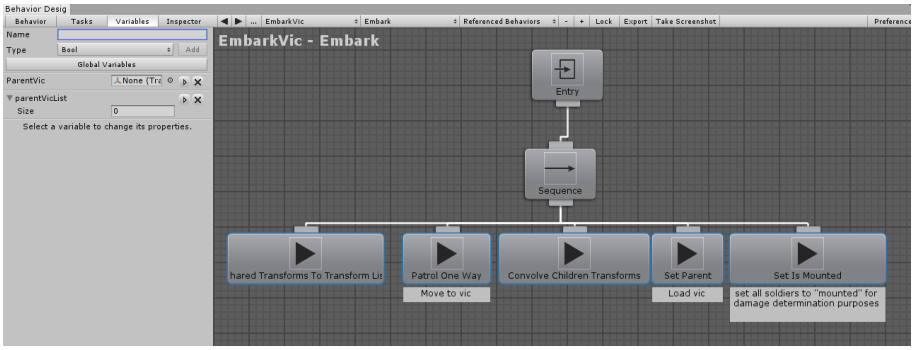

Figure A.2: The "EmbarkVic" behavior tree loads a squad into a vehicle.

EngageUntilDeadOrGone

This behavior, shown in Figure A.3, can be given to all entities in order to allow them to engage enemies within a certain range. It simply detects objects within line of sight, checks to see if any are enemies, and targets the nearest enemy. The enemy type is set to a default, as is the range, but both values can be overridden from the parent tree. If the enemy moves out of range, the behavior is reset.

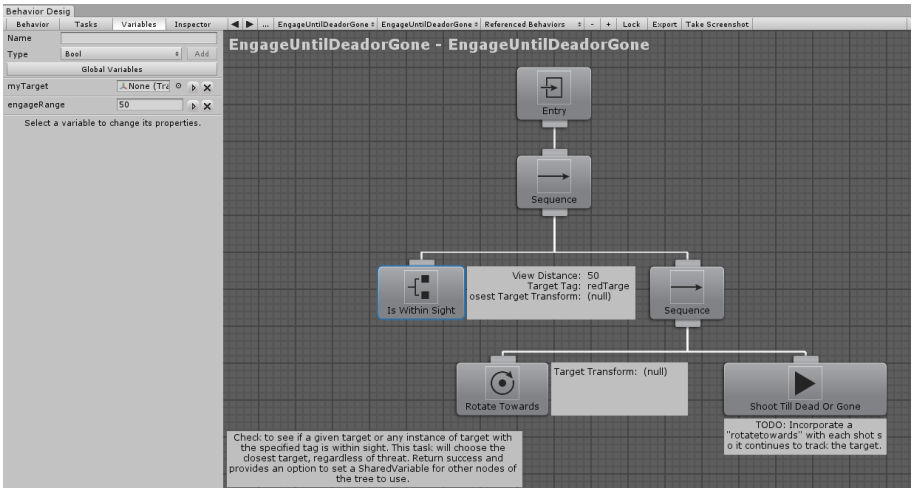

Figure A.3: The "EngageUntilDeadorGone" behavior engages the nearest enemy within range.

DetectCloseWithEngage

This behavior, shown in Figure A.4, was the behavior originally built to emulate engagement in COMBATXXI. It would be added to every combatant in a scene, and turned on or off as needed. However, the "pursuit" portion of this behavior is not similar to engagement in COMBATXXI, which is separate from movement. This led to the development of the similar "EngageUntilDeadorGone" behavior, which does not include a pursuit element. However, this behavior was kept in this collection because it serves as an example of modularity. It is triggered by line of sight detection of an enemy, after which point the two subsequent behavior trees are run in parallel. This allows developers to change one of the subordinate trees–called "behavior tree references" in Behavior Designer–and simply plug it in here without changing the structure of the parent tree. The use of a "behavior tree reference" in Behavior Designer is most similar to the use of the AddGoal() method in the COMBATXXI implementation of HTNs, although this Behavior Designer method is limited to adding a goal to the tree's owning agent.

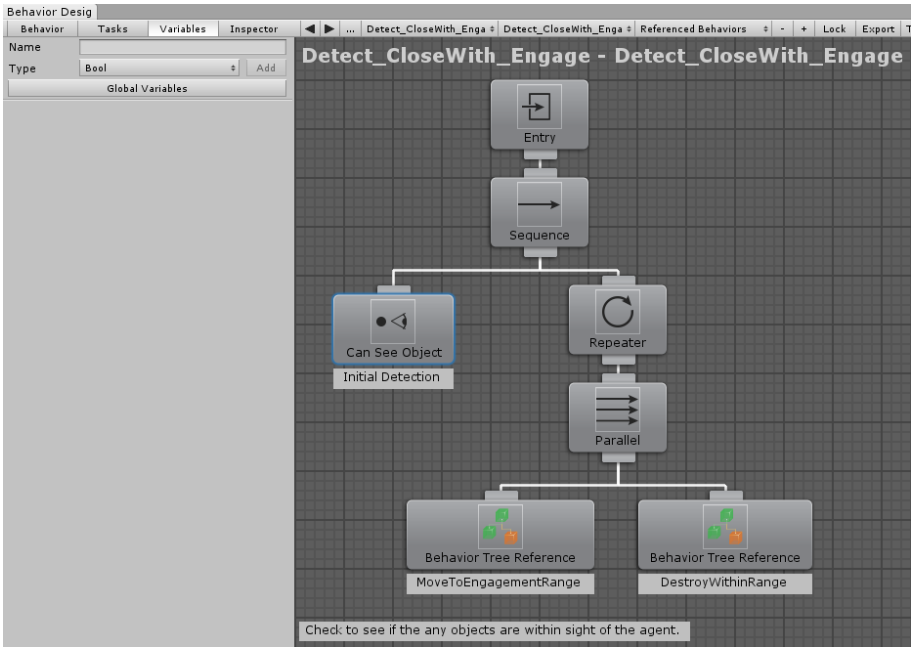

Figure A.4: The "DetectCloseWithEngage" behavior results in a search and destroy type of behavior after detecting an enemy.

MoveToEngagementRange

This behavior, depicted in Figure A.5, closes the gap between an agent's detection range and its engagement range. For example, it may be capable of detecting an agent at 600 meters, but needs to move to within 500 meters to engage. This behavior allows for modeling of that behavior. Ultimately, this is not the default behavior in COMBATXXI, so this behavior was not included in any of the final scenes.

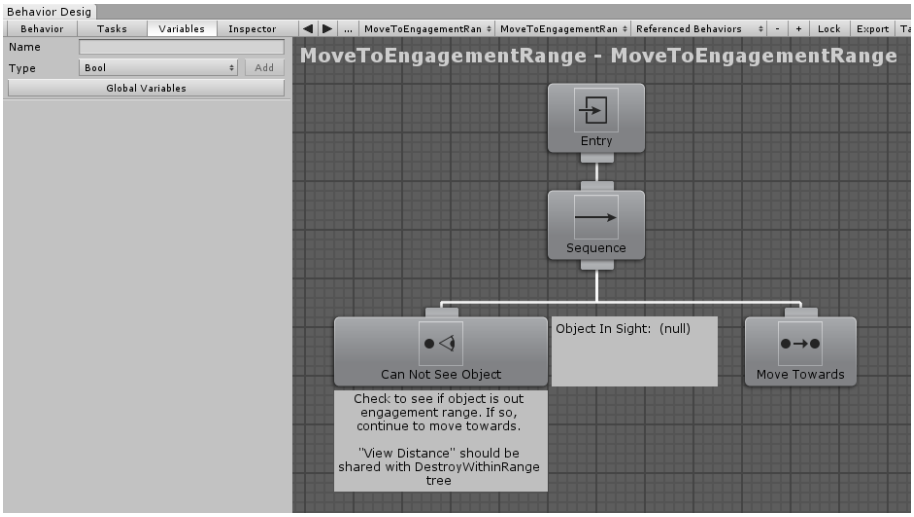

Figure A.5: The "MoveToEngage" behavior moves the agent to within engagement range.

DestroyWithinRange

This behavior, depicted in Figure A.6, is the "engage" portion of the "DetectCloseWithEngage" behavior. It engages the nearest target using a public method from a script attached to the target that reduces the target agent's health.

It is important, when using modular behaviors, to understand their originally intended use and scope. Opening the custom behavior tree and examining the "shared variables" will help to do this. Shared variables in Behavior Designer are similar to a data map in COMBATXXI HTNs. These variables may need to exist in the parent tree if they are not procedurally collected within the child, or referenced tree. While behaviors that can gather their own variables are sometimes more convenient, they are also more likely to be misused. The custom trees that were built early in this

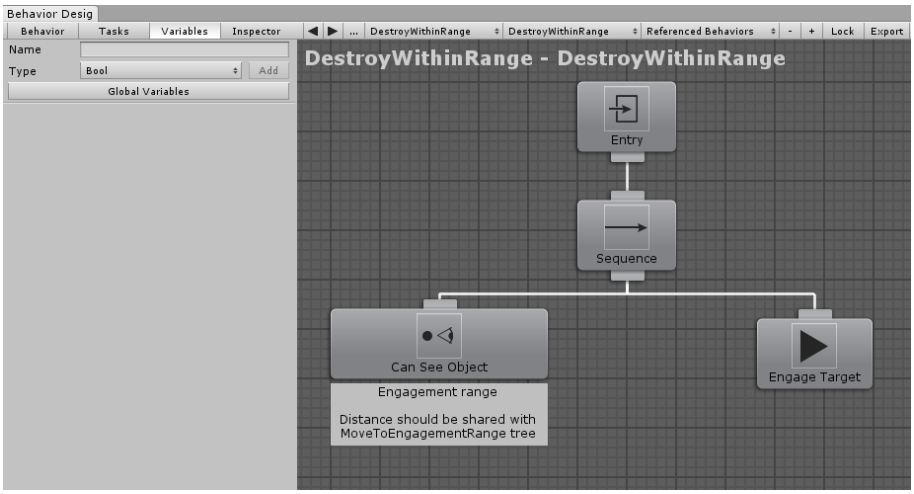

Figure A.6: The "DestroyWithinRange" behavior engages the nearest enemy within range.

process did not leverage shared variables, while those that were created later did use them.

A.2 Prefab GameObjects

Prefabs within Unity are useful for implementing large numbers of similar agents or objects. In commercial games, enemy characters are often built as prefabs and then "spawned" at various points on a map dynamically. For our purposes, prefabs were used to build out units of soldiers, statically placed or dynamically generated munitions, and other scene objects such as terrain, buildings, and control measures.

A.2.1 Agents

Several agent prefabs were built for this project which helped to increase code reuse. Each agent is represented by a large box that is subject to the Unity physics engine, and a smaller box for a "nose" which indicates the direction the agent is looking. Each agent also has a script attached to it, and the agent scripts are organized into a class hierarchy as depicted in Figure A.7. The base class provides the majority of functionality, such as health tracking, receiving damage, and removing the agent from the scene after it is killed. The subclasses only add or override functionality as

is needed for their specific functionality, such as an "isMounted" Boolean to track the mounted or dismounted status of a soldier.

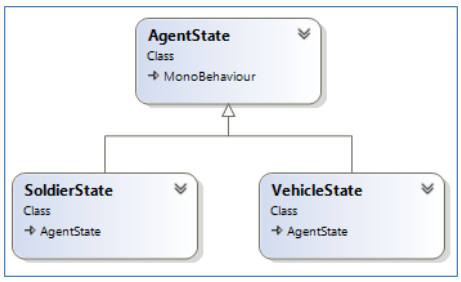

Figure A.7: The class hierarchy of scripts attached to soldiers and vehicles.

The titles of the following sections match the name of the prefab in the Unity project.

BlueSoldier_AgentState

This prefab, the visualization of which is shown in Figure A.8, is a simple model of a soldier. It is a box shaped Figure with a health bar floating overhead and a "nose" to indicate the direction it is facing. The health bar shrinks and turns progressively red as the agent's health is reduced from 100 to 0. This prefab has the "SoldierState" script attached to it, which inherits from the "AgentState" base class. Each soldier also has a line renderer attached to it which allows for visualization of shots to enemy targets.

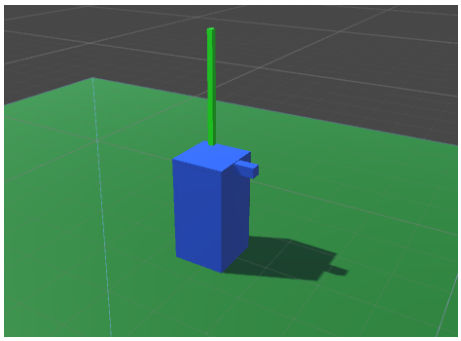

Figure A.8: The "BlueSoldier_AgentState" prefab

There is a "BlueDeath" prefab associated with this prefab, so that when the agent dies, a partially transparent blue "X" is placed where they died. This is simply for reference, and there is no physics-based component to the marker which would interfere with the scene. This can be turned on or off from the SoldierState script attached

to all instances of the BlueSoldier_AgentState prefab. The suffix "AgentState" of this prefab was added to the prefab that has the SoldierState script attached to it.

RedSoldier_AgentState

The red soldiers in this project are identical to the blue soldiers with the exception of their color and their tag. Red soldiers have a tag titled "redTarget", whereas blue soldiers have a "blueTarget" tag. Similar to "BlueDeath", there is a "RedDeath" that marks the spot of where red soldiers die.

BlueVic_AgentState

This prefab is based on the BlueSoldier_AgentState prefab, and its attached script, "VehicleState", inherits from the "AgentState" base class. Only a basic level of functionality was programmed into the vehicle prefab in order to allow it to carry soldiers around the battlefield. Instances of the vehicle prefab can be re-sized as needed to emulate a certain type of vehicle.

A.2.2 Munitions

There are two ways that agents can be damaged in this project. The first method is by direct fire, which is achieved by the firing agent calling the publicly available ReceiveDamage() method from the target agent's AgentState or appropriate sub class script.

The second method, which is achieved by the munition class, is through explosions. Explosions are modeled with the Munition class or one of its subclass scripts, which finds all enemy agents within a specified effective killing radius, and determines how much damage should be applied to each enemy agent within that radius. This is determined by a function which applies maximum damage within some range and then linearly dissipates damage the farther the enemy is away from the detonation point. Zero damage is applied beyond the effective killing radius. The munition base class and its subclasses are depicted in Figure A.9. Each munition-type prefab has a script attached to it which fits within this hierarchy.

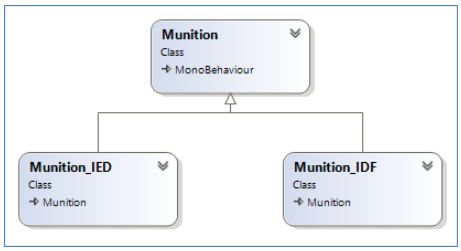

Figure A.9: The class hierarchy of scripts attached to munitions

Proximity_IED

This prefab utilizes the Munition_IED class script, which inherits from the Munition base class. The IED is represented by a small red box. The small box has an empty box around it which is meant to emulate a sensor area of interest in COMBATXXI, which can be used to set off an IED. This empty box in Unity is used as a trigger, which leverages the OnTriggerEnter() method within the Unity physics engine. The associated script detects any agent entering the box, and if it is an enemy agent, it detonates, mimicking a closely-monitored command wire IED. The prefab also includes a detonation sound and particle system, which are initiated at the same time as the damage function, for debugging purposes. The Proximity_IED prefab is depicted in Figure A.10.

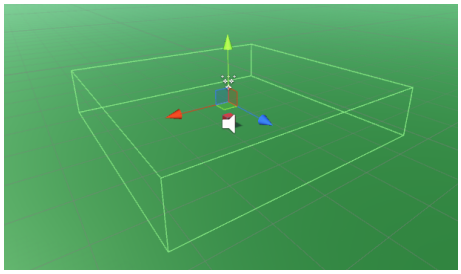

Figure A.10: A proximity IED which detonates when an enemy agent enters the empty box

60mmMortar

The second type of munition within this project is intended to emulate IDF. This a simple model of IDF; there is no ballistics modeling, and the impact area is symmetrical. There are no actual artillery agents, nor is there a fire direction center (FDC). Instead, rounds of 60 millimeter mortars are simply spawned and immediately detonated at set intervals at random points within a user-defined box. This prefab was

built in conjunction with the custom "call for fire" task, which allows a user to input these parameters, and build a call for fire functionality into a behavior tree.

This prefab, shown in action in Figure A.11, is used in conjunction with the "IDFZone" and "IDFMarker" prefabs. The latter marks the center point of the area which is to be targeted. The "IDFZone" is spawned based on the center location of the "IDFMarker", and this dictates the space in which the IDF rounds can land. Each round's location is determined by a random draw from a uniform distribution in both the length and width of the gray plane. An action shot of this behavior is depicted in Figure A.11.

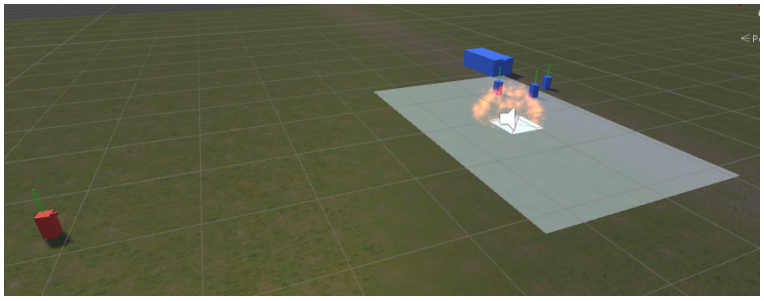

Figure A.11: The red agent on the left is calling for fire, using the 60mm-Mortar prefab, while blue agents maneuver through the impact zone.

A.2.3 Scene Elements

When prototyping behaviors using the sand box pattern, it is useful to have a prefab which includes a baseline of items needed to prototype a behavior. When a user creates a new scene in Unity, it will be empty. One could simply "save as" a scene which is already in use, but this may incur unnecessary baggage from that scene. Sometimes it is useful to start from a clean, yet basically functional slate. Similarly, some objects, such as buildings and control measures, are largely uniform, and rebuilding them each time one is needed is wasteful. This section describes some prefabs that were built for this project and were used multiple times. Each of these objects is passive, meaning no behaviors were given to them.

Control Measures

Waypoints were implemented as empty, transparent disks which allow agents to use the "patrol" task to move along a series of waypoints. These waypoints light up with a bright yellow sphere within the scene window when they are actively being used as a patrol route. This is a part of the built-in debugging features that Opsive created for Behavior Designer. The waypoints were the only control measures implemented for this project, but others, such as phase lines, could be implemented using a thin box and the OnTriggerEnter() method.

Buildings

Buildings were used for two purposes in this project. First, they are used to obstruct an agent's line of sight. Even if an enemy agent is within detection range, they will not be detected if they are on the other side of a building. With this in mind, buildings can also be used to roughly model terrain obstructions such as hills or ridges. Secondly, buildings were given a tag identifying them as buildings, so that behaviors could be built which count the buildings and make a decision based on the number of buildings. This is similar to a common scripted COMBATXXI method, which returns the number of buildings along a road.

BasicScene

The basic scene, depicted in Figure A.12, includes a 15 x 15 terrain, six buildings, area lighting, and a camera which provides an overview for the entire scene.[31] In order to use the basic scene prefab, start with a new scene, and drag the prefab in. Then, remove the original lighting and camera that came with the new scene, and reset the origin of the basic scene prefab to the world space origin. Then, users can drop agents and control measures onto the scene and they will "snap" to the terrain at the appropriate elevation.

ComplexScene

The complex scene was built in order to accommodate larger schemes of maneuver, and is depicted in Figure A.13. Otherwise its basic elements are the same as the basic

[31]Size in Unity is arbitrary. However, most Unity applications that map to the real world use a convention that one Unity unit equates to one meter.

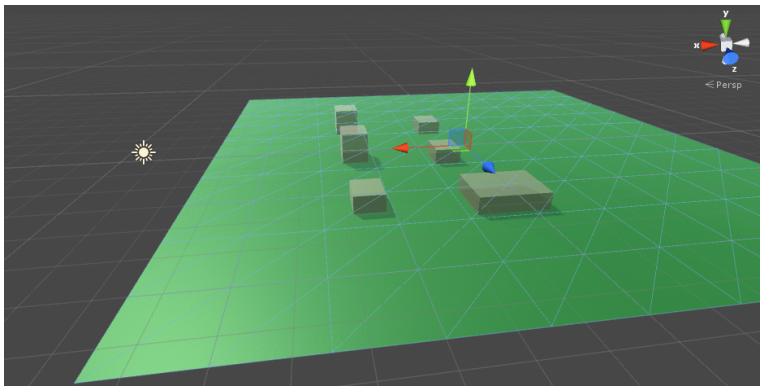

Figure A.12: The basic scene includes only terrain and several buildings.

scene. The primary addition is the debugging user interface. This UI was built using a Unity UI, and is shown in the camera view at run time. It can be turned off by deactivating it in the scene hierarchy.

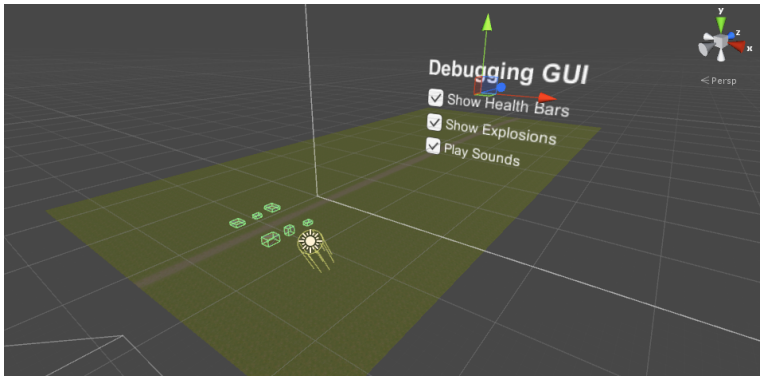

Figure A.13: The complex scene is larger than the basic scene, and includes a debugging user interface built into the game window.

A.3 Scenes for Testing Basic Behaviors

The functions of the scenes that are included in this project fall into three categories: basic behavior testing, COMBATXXI behavior replication, and new behavior prototyping for use in COMBATXXI. The basic behavior scenes were used like sand boxes, where small increments of development were conducted that would later be included in one of the more complex scenes. This method allows users to reduce complexity and focus on perfecting basic behaviors without interference from other behaviors.

A.3.1 Basic Combat Engagement Functionality

The scene titled "BasicEngage" was used as a sand box to test various engagement behaviors. It has one blue soldier, a cluster of red soldiers, and a building to impede line of sight. This basic set up allows a developer to test how an engagement behavior will deal with specific cases such as multiple enemies being within range at once, or enemies that go from within range to out of range and back in again. After clicking the "run" button, a developer can just select either the blue entity or the cluster of red entities and move them around. No movement behavior is needed to test the engagement functionality, although that could also be tested here.

A.3.2 Basic IED Functionality

Often in a combat simulation scenario, an IED is not triggered until later in the scenario. It may be more time effective to test the behavior of the IED in a small scenario, such as the scene titled "IED_test". As discussed in the munition section, munitions should induce less damage to targets that are farther away. An easy way to test that this functionality is working is to march a formation directly into an IED blast zone. The blue soldiers in this scene have a behavior which first puts them in a formation, and then marches them towards the IED. Using the health bars on top of each of the agents and noting the marker where any agents may die, it is easy to see that the agents that were closer to the blast received more damage. The console and inspector windows can be used to confirm this as well.

A.4 Scenes for Implementing Existing Behaviors

Several behaviors from existing HTN literature or COMBATXXI tutorials were built first in order to gain familiarity with the transferability of HTNs and behavior trees.

A.4.1 GoToStore

The classic example that is used to introduce new users to HTNs is the GoToStore behavior, discussed in detail in section 2.1.5. An example of this behavior implemented as a behavior tree is included in the scene titled, not surprisingly, "goToStore".

A.4.2 BOCME

The Basic Observe, Move, Communicate, and Engage, or BOCME scenario, is used to introduce novice COMBATXXI scenario designers to the interfaces needed to build new or interpret existing scenarios.

A.4.3 Bounding Overwatch

The bounding overwatch scene is similar to the the tutorial by the same name which is used to introduce students to more complex HTNs. In this scenario, a squad leader recognizes that there is sufficient cover for bounding overwatch. He then starts to move the squad down the road alternating in fire teams, using buildings as cover. If the user selects the buildings in the scene hierarchy and deactivates them, the squad leader will then determine that there is no cover. The squad will then move in a squad wedge down the middle of the road.

A.5 Scenes for Building New Behaviors

The ultimate goal of this thesis was to build new behaviors in Unity which could then be translated to COMBATXXI. The primary scene used in this endeavor is titled "platoonSecurityPatrol". When this scene is played, a fire team of three agents patrol in a wedge to a vehicle. They load into the vehicle, and the vehicle begins moving to a designated drop-off location, at which point the fire team dismounts. The fire team then maneuvers in a wedge along a specified route in order to attack a red soldier. Meanwhile, this red soldier is calling for fire, resulting in 60mm mortar rounds being dropped on the area where the blue fire team is maneuvering. This scene, part of which is shown in Figure A.11, is meant to demonstrate a more complex scheme of maneuver which can be built in Unity.

APPENDIX B:
MonoBehaviour Script Lifecycle

Figure B.1 depicts the MonoBehaviour lifecycle, which is the heartbeat of the Unity game engine. Most of the lifecycle revolves around the frame update sequence. Frame rates vary based on scene complexity and hardware capability.

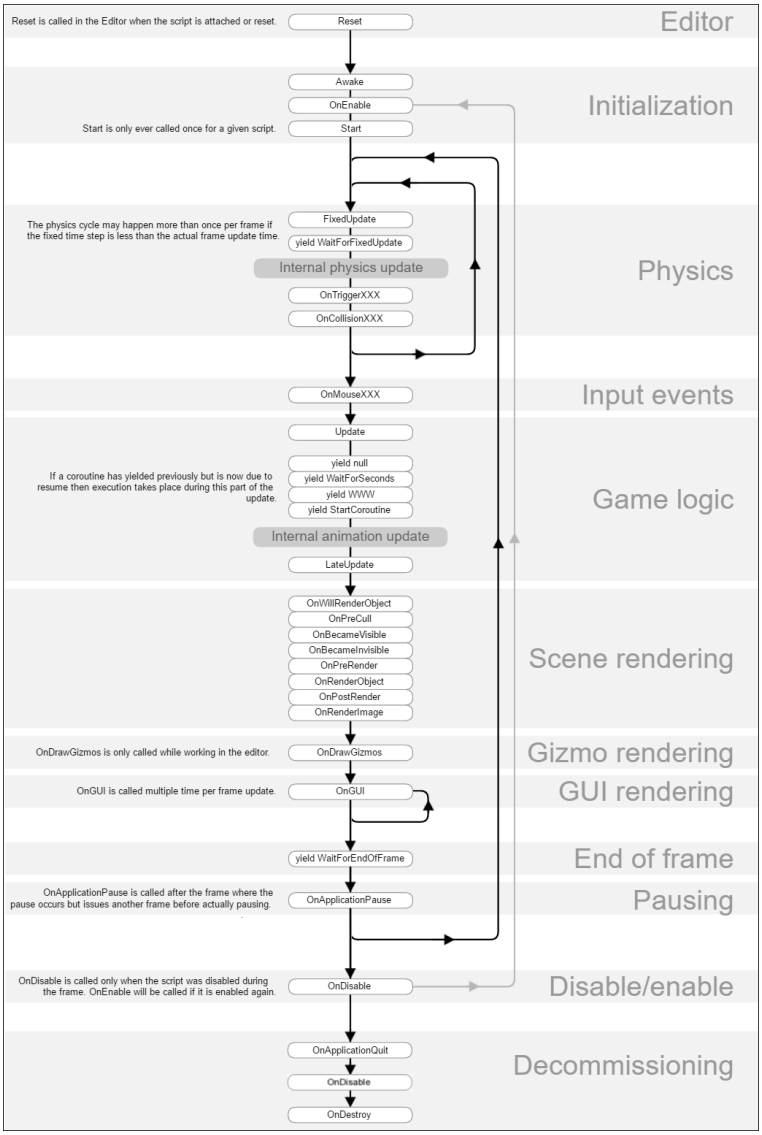

Figure B.1: MonoBehaviour Script Lifecycle from [44]

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] A. Tolk *et al.*, *Engineering principles of combat modeling and distributed simulation*. Wiley Online Library, 2012.

[2] T. W. Lucas, W. D. Kelton, P. J. Sánchez, S. M. Sanchez, and B. L. Anderson, "Changing the paradigm: Simulation, now a method of first resort," *Naval Research Logistics (NRL)*, vol. 62, no. 4, pp. 293–303, 2015. [Online]. Available: http://dx.doi.org/10.1002/nav.21628

[3] A. Ilachinski, *Artificial War: Multiagent-Based Simulation of Combat*. World Scientific Press, 2004.

[4] Operations Analysis Division, Analysis Directorate, DC CD&I. (n.d.). United States Marine Corps, Deputy Commandant, Combat Development and Integration, Analysis Directorate, Operations Analysis Division. [Online]. Available: http://www.mccdc.marines.mil/Units/Analysis/OAD.aspx. Accessed Dec. 29, 2015.

[5] Combatxxi, defined. (n.d.). United States Army, Training and Doctrine Command Analysis Center, White Sands Missile Range (TRAC-WSMR). [Online]. Available: https://cxxi.wsmr.army.mil/confluence/display/CXXIDOC/Home/. Accessed Nov. 13, 2015.

[6] United States Marine Corps, *Warfighting: Marine Corps Doctrinal Publication 1*. Washington, D.C.: DEPARTMENT OF THE NAVY, Headquarters United States Marine Corps, 1997.

[7] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 25–34, Aug. 1987.

[8] R. Nystrom, *Game programming patterns*. Genever Benning, 2014.

[9] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Pearson Education, 1999.

[10] M. J. Donaldson, "Modeling Dynamic Tactical Behaviors in COMBATXXI using Hierarchical Task Networks," M.S. thesis, Naval Postgraduate School, Monterey, 2014.

[11] H. Wallach and D. O'Connell, "The kinetic depth effect." *Journal of experimental psychology*, vol. 45, no. 4, p. 205, 1953.

[12] T. Sando, M. Tory, and P. Irani, "Effects of animation, user-controlled interactions, and multiple static views in understanding 3d structures," in *Proceedings of the 6th Symposium on Applied Perception in Graphics and Visualization*, ser. APGV '09. New York, NY, USA: ACM, 2009, pp. 69–76.

[13] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: Theory and Practice*, ser. Morgan Kaufmann Series in Artificial Intelligence. Elsevier/Morgan Kaufmann, 2004.

[14] A. J. Champandard. (2013, Mar. 8). Planning in games: An overview and lessons learned. [Online]. Available: http://aigamedev.com/open/review/planning-in-games/

[15] I. L. Balogh, D. Reeves, and C. Fitzpatrick, "Using hierarchical task networks to create dynamic behaviors in combat models," unpublished.

[16] A. J. Champandard. (2007, Sep. 4). On finite state machines and reusability. [Online]. Available: http://aigamedev.com/open/article/fsm-reusable/

[17] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," in *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, ser. IJCAI'71. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1971, pp. 608–620.

[18] "Hierarchical task networks and dynamic maneuver," class notes for Simulation Development Practicum, MOVES Inst., Naval Postgraduate School, Monterey, CA, fall 2015.

[19] A. J. Champandard. (2012, Mar. 28). The #fsmgate scandal and what it means for your AI architecture. [Online]. Available: http://aigamedev.com/open/editorial/fsmgate-scandal/

[20] A. Kyaw, C. Peters, and T. Swe, *Unity 4.x Game AI Programming*, ser. Community experience distilled. Packt Publishing, 2013.

[21] D. Isla. (2005, Mar. 11). Understanding behavior trees. [Online]. Available: http://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php

[22] S. Rabin, *AI Game Programming Wisdom 4*, ser. AI Game Programming Wisdom. Course Technology, Cengage Learning, 2008.

[23] A. J. Champandard. (2007, Sep. 6). Understanding behavior trees. [Online]. Available: http://aigamedev.com/open/article/bt-overview/

[24] Behavior designer. (n.d.). Opsive Software Development. [Online]. Available: http://www.opsive.com/assets/BehaviorDesigner/. Accessed Dec. 29, 2015.

[25] *Standard for Distributed Interactive Simulation–Application Protocols*, IEEE Std. 1278.1, 2012.

[26] Platoon scenario builders guide. (n.d.). United States Army, Training and Doctrine Command Analysis Center, White Sands Missile Range (TRAC-WSMR). [Online]. Available: https://cxxi.wsmr.army.mil/confluence/display/CXXIDOC/Platoon+Scenario+Builders+Guide. Accessed Nov. 13, 2015.

[27] T. DiChristopher. (2016, Jan. 26). Digital gaming sales hit record $61 billion in 2015. [Online]. Available: http://www.cnbc.com/2016/01/26/digital-gaming-sales-hit-record-61-billion-in-2015-report.html

[28] K. Cox. (2014, Jun. 9). It's time to start treating video game industry like the $21 billion business it is. [Online]. Available: https://consumerist.com/2014/06/09/its-time-to-start-treating-video-game-industry-like-the-21-billion-business-it-is/

[29] J. Skrebels. (2016, Apr. 1). Final fantasy 15 has to sell 10 million copies to make back its investment. [Online]. Available: http://www.ign.com/articles/2016/04/01/final-fantasy-15-has-to-sell-10-million-copies-to-make-back-its-investment

[30] J. Appleget, R. Burks, and M. Jaye, "A Demonstration of ABM Validation Techniques by Applying Docking to the Epstein Civil Violence Model," *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, pp. 403–411, 2013.

[31] S. B. Van Hemel, J. MacMillan, G. L. Zacharias *et al.*, *Behavioral Modeling and Simulation:: From Individuals to Societies*. National Academies Press, 2008.

[32] A. Noonchester, "AI in the Awesomepocalypse," presented at the Eleventh Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 2015. [Online]. Available: https://www.youtube.com/watch?v=ZIAmoRsu3Z0

[33] Z. Parrish. (2014, May 12). Introduction to blueprints. [Online]. Available: https://www.unrealengine.com/blog/introduction-to-blueprints

[34] K. Beck, *Test-driven Development: By Example*, ser. A Kent Beck signature book. Addison-Wesley, 2003.

[35] D. H. Hanson. (2014, Apr. 23). TDD is dead. Long live testing. [Online]. Available: http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html

[36] J. Brodkin. (2013, Jun. 3). How unity3d became a game-development beast. [Online]. Available: http://insights.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/

[37] The leading global game industry software. (n.d.). Unity3D. [Online]. Available: https://unity3d.com/public-relations. Accessed Nov. 13, 2013.

[38] A. H. Buss and P. J. Sánchez, "Simple movement and detection in discrete event simulation," in *Simulation Conference, 2005 Proceedings of the Winter*. IEEE, 2005, pp. 992–1000.

[39] Engagement processes. (n.d.). United States Army, Training and Doctrine Command Analysis Center, White Sands Missile Range (TRAC-WSMR). [Online]. Available: https://cxxi.wsmr.army.mil/confluence/display/CXXIDOC/Engagement+Processes. Accessed Nov. 13, 2015.

[40] W. W. Royce, "Managing the development of large software systems," in *proceedings of IEEE WESCON*, vol. 26, no. 8. Los Angeles, 1970, pp. 1–9.

[41] B. W. Boehm, "A spiral model of software development and enhancement," *Computer*, vol. 21, no. 5, pp. 61–72, 1988.

[42] B. R. Harder, I. L. Balogh, C. J. Darken, "Implementation of an Automated Fire Support Planner," unpublished.

[43] Simulation Interoperability Standards Organization. Overview of SISO: Who we are and what we do. (n.d.). https://www.sisostds.org/AboutSISO/Overview.aspx. Accessed 08-February-2016.

[44] Execution order of event functions. (n.d.). Unity3D. [Online]. Available: http://docs.unity3d.com/Manual/ExecutionOrder.html. Accessed Mar. 13, 2016.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California